TAMPEREEN TEKNILLINEN YLIOPISTO

Petteri Aimonen

Videokuvan keskihajonnan laskennan nopeusoptimointi SSE2-käskykannan avulla

Kandidaatintyö

Tarkastaja: Erno Salminen

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Signaalinkäsittelyn ja tietoliikennetekniikan koulutusohjelma
PETTERI AIMONEN: Videokuvan keskihajonnan laskennan nopeusoptimointi SSE2-käskykannan avulla
Kandidaatintyö, 51 sivua, ei liitesivuja
Toukokuu 2010
Pääaine: Digitaali- ja tietokonetekniikka
Tarkastajat: Erno Salminen
Avainsanat: SSE2-käskykanta, OpenMP, optimointi, nopeusoptimointi, konvoluutio, keskihajonta, prosenttipiste

Tämä kandidaatintyö käsittelee videokuvan käsittelyyn liittyvän algoritmin nopeuttamista. Työn pohjana olevasta algoritmista on valmis prototyyppi, johon saavutettua nopeutta verrataan. Algoritmin tarkoitus on mitata videokuvassa tapahtuvien muutosten suuruutta. Algoritmi laskee videokuvasta pikselikohtaisen keskihajonnan edellisten kuvien ajalta, ja keskihajonnan arvoista prosenttipisteen. Lisäksi kuvat suodatetaan konvoluutiolla ennen laskentaa.

Optimoinnissa käytettiin prosessorin SSE2-käskykantaa (engl. Streaming SIMD Extensions 2), joka on eräs SIMD-periaatteen (engl. Single Instruction Multiple Data) toteutus. SSE2-käskyt suorittavat saman operaation usealle luvulle, jolloin prosessori pystyy käsittelemään tietoa nopeammin kuin yksi lukuarvo kerrallaan. Lisäksi laskenta rinnakkaistettiin usealle prosessorille ja laskentatapaa parannettiin tutkimalla algoritmia matemaattisesti.

Mittaustulosten perusteella algoritmin nopeus noin kymmenkertaistui verrattuna aiempaan prototyyppiin. Tämän vertailun lisäksi mittausten avulla tutkittiin nopeuden riippuvuutta kuvien koosta, liikkeen määrästä ja muista tekijöistä. Yhteenvedossa on esitetty projektin eteneminen, työmäärä ja eri optimointitapojen vaikutus.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Signal Processing and Communications Technology
PETTERI AIMONEN: Using SSE2 instruction set for speed optimizing the calculation of standard deviation from video image.
Master of Science Thesis, 51 pages, no appendix pages
May 2010
Major: Digital and Computer Systems
Examiner: Erno Salminen
Keywords: SSE2 instruction set, OpenMP, optimizing, speed optimizing, convolution, standard deviation, percentile

This Master of Science Thesis focuses on speeding up a video processing algorithm. There is an existing prototype of the algorithm in question, which is used for comparing the achieved performance. The purpose of the algorithm is to measure the amount of movement in video image. The algorithm calculates the per-pixel standard deviation over a number of past frames in video image, and then calculates a specified percentile of the deviation results. Additionally, the images are filtered with convolution prior to other calculations.

The primary method used in optimizing is the SSE2 (Streaming SIMD Extensions 2) instruction set, which is an implementation of the SIMD (Single Instruction Multiple Data) principle. The SSE2 instructions perform the same operation to multiple values, so that the processor can process information more efficiently than when calculating a single value at a time. In addition, the calculation was parallelized to multiple processor cores and the algorithm was modified using mathematical insight.

Based on measurement results, the speed of the algorithm increased tenfold compared to the earlier prototype. Measurements were also used to study the effect of image size, amount of movement and other factors on the speed. The conclusion presents the progress of the project, amount of work involved and the effect of various optimization methods.

SISÄLLYS

TERMIT JA SYMBOLIT

c Kuvan värikanavien määrä.

$f(x,y)$ Algoritmille syötetyn kuvan pikseliarvo kohdassa (x,y).

$g(x,y)$ Konvoluution tulos pikselille, joka on kohdassa (x,y).

h Kuvan korkeus.

i Kuvapuskurin indeksi keskihajonnassa.

k Konvoluutiokernelin koko, sama kumpaankin suuntaan.

n Keskihajonnan laskennassa käytettävä kuvapuskurien lukumäärä.

N Prosenttipisteen laskennassa käsiteltävien lukujen lukumäärä, eli tämän algoritmin tapauksessa $w \cdot h$.

$O(\ldots)$ Algoritmien vertailussa käytetty merkintä, joka kuvaa ajan- tai muistinkulutuksen verrannollisuutta käsiteltävän tiedon määrään.

s Kokonaislukulaskennassa käytettävä skaalauskerroin konvoluution ja keskihajonnan yhteydessä.

w Kuvan leveys.

fps

Frames Per Second eli kuvaa sekunnissa, käsittelynopeudelle käytetty mittayksikkö.

GCC

GNU C Compiler, GNU-projektin C-kääntäjä, joka on saatavilla usealle alustalle

L2-välimuisti

Prosessorin toisen tason välimuisti. Intelin nykyisissä prosessoreissa se on samalla myös viimeisen tason välimuisti, eli siitä puuttuvat muistihaut viedään ulkoiselle muistille.

Microsoft Visual C++

Microsoftin C-kääntäjä Windows-alustalle

MPix

Kuvan koon yksikkö, megapikseli eli miljoona pikseliä.

MPix/s

Käsittelynopeudelle käytetty mittayksikkö, megapikseliä sekunnissa.

OpenMP

Open Multi-Processing, usean kääntäjävalmistajan tukema rajapinta rinnakkaislaskentaan

SIMD

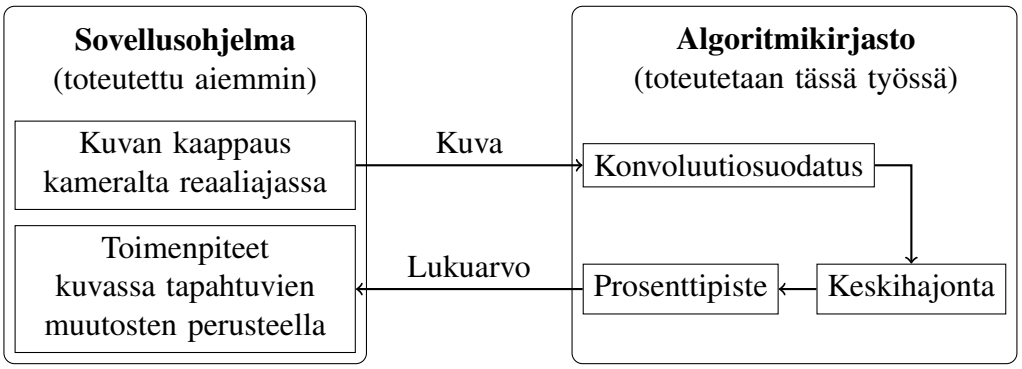
Single Instruction Multiple Data, prosessorien käskytyyppi jossa sama operaatio suoritetaan usealle luvulle yhdellä käskyllä

SSE2

Streaming SIMD Extensions 2, Intelin kehittämä käskykantalaajennos SIMD-laskentaa varten

1. JOHDANTO

Tämä kandidaatintyö käsittelee videokuvan käsittelyyn liittyvän algoritmin nopeuttamista. Työn pohjana on aiemmin suunniteltu algoritmi, josta on toimiva prototyyppi jonka toimintaa halutaan nopeuttaa. Käsiteltävän järjestelmän rakenne on esitetty kuvassa 1.1.



Kuva 1.1: *Työssä toteutettavan algoritmikirjaston toimintaympäristö ja perusrakenne.*

Algoritmi koostuu kolmesta pääalueesta: kuvan konvoluutiosuodatuksesta, pikselikohtaisen keskihajonnan laskennasta sekä keskihajonnan halutun prosenttipisteen laskennasta. Laskenta on luonteeltaan helposti rinnakkaistuva, joten käytettäväksi menetelmiksi soveltuvat rinnakkaiseen laskentaan perustuvat SSE2-käskykanta (engl. Streaming SIMD Extensions 2) ja monen prosessoriytimen käyttö.

Aiempi prototyyppi on toteutettu erityistä signaalinkäsittelyn laskentaohjelmaa käyttäen. Tässä työssä toteutettavasta versiosta halutaan yhteensopiva eri alustojen kanssa, joten toteutuskieleksi valittiin C. Eri toteutusalustasta huolimatta järjestelmän muihin osiin tehtävät muutokset on tarkoitus pitää vähäisinä, joten algoritmin rajapinta tehdään samanlaiseksi kuin prototyypissä.

Algoritmia käytetään perustasoisella uudella tietokoneella, jonka prosessori on esimerkiksi Intel Pentium Dual-Core E5300 2,60 GHz. Aiemman prototyypin nopeus on ollut harmaasävykuvilla n. 10 MPix/s (megapikseliä sekunnissa). Tavoitteena on saavuttaa optimoinnin jälkeen 40 MPix/s laskentanopeus.

Luvussa 2 esitellään lyhyesti optimoinnissa käytettävät menetelmät ja luvussa 3 käsitellyn algoritmin rakenne. Kunkin algoritmin osa-alueen optimointi on esitelty luvuissa 4, 5 ja 6. Luvussa 7 on arvioitu toteutusta nopeusmittausten perusteella ja luvussa 8 on yhteenveto tuloksista ja projektin etenemisestä.

2. KÄYTETTÄVÄT MENETELMÄT

Tässä luvussa on esitelty käytettäväksi harkitut työkalut, kuten kääntäjä ja lisäkirjastot, sekä käytetyt optimointimenetelmät. Lisäkirjastoja tai näytönohjainlaskentaa ei lopulta käytetty toteutuksessa.

2.1 Kääntäjät

Algoritmi toteutetaan C-kielellä ja kääntäjinä käytetään Microsoft Visual C++:aa sekä GCC:tä (engl. GNU C Compiler). Näin saavutetaan hyvä siirrettävyys eri alustoille pienellä työmäärällä, sillä kummassakin kääntäjässä on tuki OpenMP-rinnakkaislaskennalle (engl. Open Multi-Processing) sekä SSE2:n käytölle. Eroja on lähinnä joissakin syntaksiseikoissa.

2.2 Apukirjastot

On olemassa valmiita kirjastoja, jotka sisältävät optimoituja aliohjelmia yleisiin tarpeisiin. Tällaisia ovat esimerkiksi Intel Integrated Performance Primitives ja liboil. Valmiiden kirjastojen käyttö helpottaa ohjelmointia, ja on useissa tapauksissa järkevä vaihtoehto.

Tässä työssä ei kuitenkaan käytetty valmiita kirjastoja, sillä niistä olisi voitu hyödyntää lähinnä pelkkä konvoluutiolaskenta. Muilta osin valmiiden kirjastojen käyttö olisi rajoittanut saavutettavaa nopeutta, sillä esimerkiksi kokonaislukulaskentaan tai liukuvan ikkunan käyttöön perustuvat menetelmät olisi ollut vaikea toteuttaa tehokkaasti.

2.3 Kokonaislukulaskenta

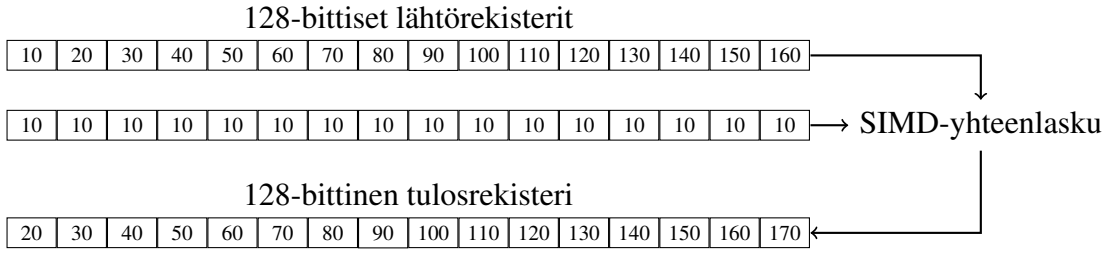
Kaikki nopeuskriittinen laskenta suoritetaan toteutuksessa kokonaisluvuilla. Kokonaislukuoperaatiot ovat bittileveydestä ja käskystä riippuen noin 2–8 kertaa nopeampia kuin vastaavat liukulukuoperaatiot [1, s. C-9, C-18].

Kokonaislukulaskenta soveltuu tähän algoritmiin hyvin, sillä käsiteltävät pikseliarvot ovat joka tapauksessa kokonaislukuja. Joissakin kohdissa kokonaislukujen käyttö vaatii kuitenkin erityistä huomiota lukualueiden ja pyöristyksen osalta.

2.4 SSE2-käskykanta

SSE2-käskykanta on työpöytätietokoneiden *x86*-prosessoreiden käskykannan laajennos, joka sisältää SIMD-käskyjä (engl. Single Instruction Multiple Data). Nämä käskyt ovat käytettävissä Intelin prosessoreissa vuoden 2000 Pentium 4-mallista alkaen [2, s. 2-4], sekä AMD:n prosessoreissa vuoden 2003 Opteron- ja Athlon 64 -malleista alkaen [3] [4].

SIMD-käskyissä prosessori suorittaa yhdellä käskyllä saman operaation monelle lukuarvolle, SSE2:n tapauksessa enimmillään 16 luvulle. Käsiteltävät luvut on pakattu samaan 128-bittiseen rekisteriin, ja niiden bittimäärä voi olla kokonaislukujen tapauksessa joko 8, 16 tai 32. Lisäksi SSE2 sisältää käskyjä 32- ja 64-bittisten liukulukujen käsittelyyn. Käsiteltävien lukujen määrä riippuu suoraan niiden bittimäärästä, esimerkiksi 16-bittisiä lukuja voidaan käsitellä 8 luvun ryhmissä. Periaate on esitetty kuvassa 2.1.



Kuva 2.1: *SIMD-laskennan periaate. SIMD-käskyt suorittavat saman operaation usealle luvulle. Käsiteltävät luvut on pakattu yhteen rekisteriin, jonka leveys on SSE2-käskykannassa 128 bittiä. Kuvassa luvut ovat 8-bittisiä, joten niitä mahtuu rekisteriin 16 kappaletta.*

2.4.1 Nopeusedut

Laskenta on nopeampaa, kuin jos samat operaatiot suoritettaisiin jokaiselle luvulle erikseen. Nopeusetu syntyy kahdesta tekijästä: laskennan rinnakkaisuudesta ja pienemmästä määrästä käskyjen tulkintaa. SIMD-käskyissä laskutoimitusten rinnakkaistaminen on tehokkaampaa kuin yksittäisillä käskyillä, koska niiden riippuvuussuhteita ei tarvitse erikseen selvittää.

Saavutettava nopeusetu on merkittävä, sillä SSE2-käskyjen suoritusnopeus on verrattavissa yksittäisten laskutoimitusten nopeuteen. Esimerkiksi 16-bittisen kertolaskun läpäisy (engl. throughput) on SSE2-käskyllä yksi käsky 4:ssä kellojaksossa ja yksittäisoperaationa yksi käsky 2:ssa kellojaksossa [1, s. C-30]. SSE2-käsky suorittaa saman kertolaskun 8 luvulle, joten nopeus on nelinkertainen.

2.4.2 Ongelmakohdat

SSE2-käskyissä käytettävissä olevia operaatioita on vähemmän kuin yksittäisillä luvuilla laskettaessa. Esimerkiksi jakolasku puuttuu, joten se täytyy tarvittaessa suorittaa erikseen tai vakiojakajan tapauksessa korvata kertolaskulla.

Käsiteltäessä useita lukuja kerrallaan ei voida myöskään käyttää tavanomaisia päätösrakenteita lukujen vertailuun, sillä osa rekisterissä olevista luvuista voi täyttää ehdon ja osa ei. Ellei päätösrakenteita voida karsia, täytyy kummankin haaran operaatiot suorittaa kaikille luvuille ja yhdistää sitten lopulliseen tulokseen tulevat luvut erityisellä käskyllä.

Lukujen bittileveys määrää sen, montako lukua voidaan käsitellä yhtäaikaisesti. Laskenta on nopeinta pienimmällä 8 bitin bittileveydellä, mutta joissain kohdissa käsiteltävien lukujen arvoalue vaatii 16 tai 32 bitin leveyden. Parhaan nopeuden saavuttamiseksi voi olla tarpeen muuttaa luvut esimerkiksi 16-bittisiksi vasta keskellä laskutoimitusta, jolloin alkuosa voidaan suorittaa nopeammin 8-bittisillä käskyillä. Useat muunnokset kuitenkin monimutkaistavat ohjelmakoodia.

2.4.3 Kääntäjän tuki

SSE2-käskyjä käytetään tässä työssä kääntäjän sisäänrakennettujen funktioiden (engl. intrinsic functions) kautta. Tämä vähentää ohjelmoijan työmäärää verrattuna assembler-koodiin, koska kääntäjä pystyy itse hoitamaan muuttujien sijoittelun rekistereihin sekä tekemään muita optimointeja koodille. Ohjelmointi on silti huomattavasti työläämpää kuin tavallisen C-koodin kirjoittaminen. Listauksessa 2.1 on vertailtu tavallista ja SSE2-käskykantaa käyttävää ohjelmaa.

Jokainen funktio vastaa yhtä SSE2-käskyä, ja niitä kutsutaan vastaavasti kuin tavallisia funktioita. Funktioiden kanssa käytetään erityistä `__mm128i`-tietotyyppiä, joka kuvaa SSE2-rekistereitä. Tätä tyyppiä ei voi suoraan käsitellä tavallisilla C-kielen operaatioilla, vaan tulokset täytyy tallentaa muistiin erillisellä käskyllä.

2.5 Rinnakkaislaskenta

Nykyaikaisissa prosessoreissa on usein kaksi tai useampi suoritinydin. Näiden hyödyntämiseksi laskenta täytyy rinnakkaistaa useampaan säikeeseen. Tämän algoritmin yhteydessä se on yksinkertaista, sillä operaatiot ovat suurelta osin toisistaan riippumattomia.

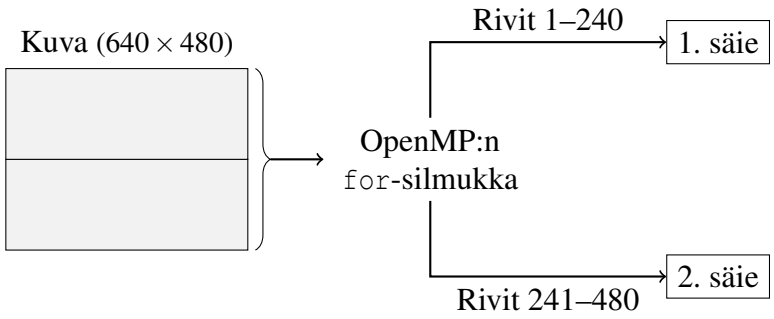
Algoritmi on rinnakkaistettu kuvan 2.2 mukaisesti siten, että jokainen kuva jaetaan eri säikeissä käsiteltäviin vaakasuoriin osiin. Kahden prosessoriytimen tapauksessa kumpikin ydin käsittelee siis puolet kuvasta.

OpenMP:n käyttämä jakotapa voidaan valita `schedule`-määreellä. Tässä työssä ja esimerkissä käytetään määrettä `schedule(static)`, joka jakaa silmukan arvoalueen

```
// Tavallinen C-toteutus
void summaa_rivi(uint8_t *rivi1, uint8_t *rivi2, uint8_t *tulos, int leveys)
{
    int x;
    // Summataan rivi1:n jokainen pikseli vastaavaan pikseliin rivi2:ssa.
    for (x = 0; x < leveys; x++)
    {
        tulos[x] = rivi1[x] + rivi2[x];
    }
}

// SSE2-toteutus
void summaa_rivi(uint8_t *rivi1, uint8_t *rivi2, uint8_t *tulos, int leveys)
{
    int x;
    // Käsitellään pikseleitä 16 tavun lohkoissa.
    for (x = 0; x <= leveys - 16; x += 16)
    {
        // Lähtöarvojen lataaminen muistista
        __m128i lohko1 = _mm_loadu_si128((__m128i*)(rivi1 + x));
        __m128i lohko2 = _mm_loadu_si128((__m128i*)(rivi2 + x));

        // Yhteenlasku
        __m128i tuloslohko = _mm_add_epi8(lohko1, lohko2);

        // Tuloksen tallennus
        _mm_storeu_si128((__m128i*)(tulos + x), tuloslohko);
    }

    // Jos leveys ei ole jaollinen 16:lla, käsitellään rivin loppuosa
    // erikseen.
    for (; x < leveys; x++)
    {
        tulos[x] = rivi1[x] + rivi2[x];
    }
}
```

Listaus 2.1: *Sama funktio toteutettuna tavallisena C-kielisenä ohjelmana sekä SSE2-käskyillä. Esitetty funktio summaa yhden rivin pikseleitä toiseen riviin, ja tallentaa tuloksen annettuun kohtaan muistissa. Käsiteltävät pikseliarvot ovat 8-bittisiä. SSE2-käskyillä tehty toteutus on huomattavasti monimutkaisempi, ja vaatii lisäksi erillisen toteutuksen rivin loppuosaa varten.*

samankokoisiin osiin ja käsittelee yhden osan kussakin säikeessä [5, s.43]. Koska rivin käsittelyaika ei riipu merkittävästi rivin sisällöstä, kaikki säikeet valmistuvat yhtäaikaisesti. Jos käsittelyaika vaihtelisi enemmän, voitaisiin käyttää jakotapaa `schedule(dynamic)`, joka jakaa rivejä ajon aikana sitä mukaa kun säikeet ovat saaneet edelliset rivit käsiteltyä.



Kuva 2.2: *Rinnakkaislaskennan toteutustapa tässä työssä. Kukin kuva jaetaan eri prosessoreilla käsiteltäviin vaakasuoriin osiin.*

Rinnakkaistus on toteutettu kääntäjän OpenMP-tuella [6] [7]. Etuna on hyvin helppo käyttö sekä siirrettävyys eri alustoille. OpenMP:tä käytetään kääntäjän `#pragma`-lauseiden kautta, joilla määrätään esimerkiksi `for`-silmukka jaettavaksi eri säikeisiin. Ohjelma toimii suoraan myös ilman OpenMP-tukea käännettäessä, mutta käyttää tällöin vain yhtä prosessoria. Listauksessa 2.2 on esimerkki OpenMP:n käytöstä.

```
void summaa_kuvat(uint8_t *kuva1, uint8_t *kuva2, uint8_t *tulos,
                  int leveys, int korkeus)
{
    #pragma omp parallel for schedule(static)
    for (int y = 0; y < korkeus; y++)
    {
        // Kussakin säikeessä käydään läpi osa y:n arvoista.
        // Rinnakkaistus ei vaadi tässä tapauksessa koodiin mitään muutoksia,
        // sillä rivit ovat täysin riippumattomia toisistaan.
        summaa_rivi(kuva1 + y*leveys, kuva2 + y*leveys, tulos + y*leveys);
    }
}
```

Listaus 2.2: *Esimerkki OpenMP:n käytöstä `for`-silmukan rinnakkaistamiseen. Tässä tapauksessa rinnakkaistus onnistuu pelkällä yhden rivin lisäyksellä. Usein samoja muuttujia käytetään kuitenkin eri säikeistä, jolloin tarvitaan enemmän koodia.*

Säikeistyksen voi toteuttaa myös muilla tavoin, kuten *pthreads*-kirjastolla, mutta tämä on työläämpää ja virhealtista. On olemassa myös erityisiä rinnakkaisohjelmointikieliä, joista on etua monimutkaisempien algoritmien yhteydessä. Tämä algoritmi rinnakkaistuu kuitenkin hyvin yksinkertaisesti.

2.6 Näytönohjainlaskenta

Näytönohjaimessa on suuri määrä rinnakkaisia prosessoreita, jotka soveltuvat erityisen hyvin tässä käsitellyn algoritmin kaltaiseen helposti rinnakkaistuvaan laskentaan. Tällä hetkellä ongelmana on kuitenkin huono siirrettävyys eri alustoille, kuten eri valmistajan näytönohjaimille.

Toimivuuden varmistamiseksi algoritmista täytyisi olla myös pääprosessorilla ajettava versio, mikä kaksinkertaistaisi työmäärän. Nopeustavoitteet saavutettiin ilman näytönohjainlaskennan hyödyntämistä, joten tähän ei käytetty aikaa.

Tilanteeseen on tulossa parannusta OpenCL-kielen myötä [8]. Sen avulla sama ohjelma pystytään ajamaan sekä eri näytönohjaimilla että tarvittaessa myös tietokoneen pääprosessorilla. Kielen kehitys on kuitenkin vielä alkuvaiheessa, ja SiSoftwaren [9] mittaustulosten mukaan suorituskyky pääprosessorilla on merkittävästi huonompi kuin SSE2:a suoraan käytettäessä.

2.7 Tarkistukset ja mittaukset

Algoritmin aiempi prototyyppi on integroitu kiinteästi sovellusohjelmaan, joten sen käyttö optimoidun version tarkistamisessa olisi ollut työlästä. Tästä syystä algoritmista tehtiin aluksi C-kielinen referenssitoteutus, jonka avulla voitiin varmistaa optimoidun version oikea toiminta. Referenssitoteutusta verrattiin aluksi muutamassa testitapauksessa algoritmin matemaattiseen kuvaukseen, ja sitten usealla automaattisesti generoidulla testitapauksella optimoituun versioon. Ideana on se, että virheiden esiintyminen selkeässä ja yksinkertaisessa referenssitoteutuksessa on epätodennäköisempää kuin optimoidussa versiossa.

Sovellusohjelman yleisimmin käyttämä tulos on keskihajonnan prosenttipiste, mutta pienet virheet välituloksissa eivät näy niin herkästi siinä. Sen vuoksi algoritmin tarkistuksessa vertailut tehtiin lopputuloksen lisäksi välituloksille, eli konvoluution ja pikselikohtaisen keskihajonnan antamille taulukoille.

Optimoidun version nopeus määritettiin käyttäen C-kielistä testiohjelmaa, joka luki joukon still-kuvia muistiin ja suoritti sitten algoritmin niille. Sen sijaan aiemman prototyypin nopeutta ei pystytty tarkasti määrittämään, sillä sen erottaminen sovellusohjelmasta olisi ollut työlästä. Sen vuoksi vertailussa piti tyytyä pelkkään arvioon aiemmasta nopeudesta.

3. ALGORITMIN ESITTELY

Työssä käsiteltävän algoritmin perustarkoitus on videokuvan muutosten analysointi. Tätä varten algoritmi pitää sisäisessä muistissaan valitun määrän edellisiä kuvia ja laskee niiden perusteella pikselikohtaisen keskihajonnan. Algoritmin toiminta on esitetty esimerkillä kuvassa 3.1.

Algoritmin muut osat liittyvät kuvien esikäsittelyyn ja tulosten jälkikäsittelyyn. Videokuvat voidaan suodattaa ennen laskentaa konvoluutiosuotimella, jolla voidaan toteuttaa esimerkiksi sumennus tai reunojen tunnistus. Tuloksena saaduista keskihajonnan arvoista puolestaan voidaan laskea prosenttipiste tai tietyn kynnysarvon ylittävien pikselien määrä.

3.1 Osien merkitys

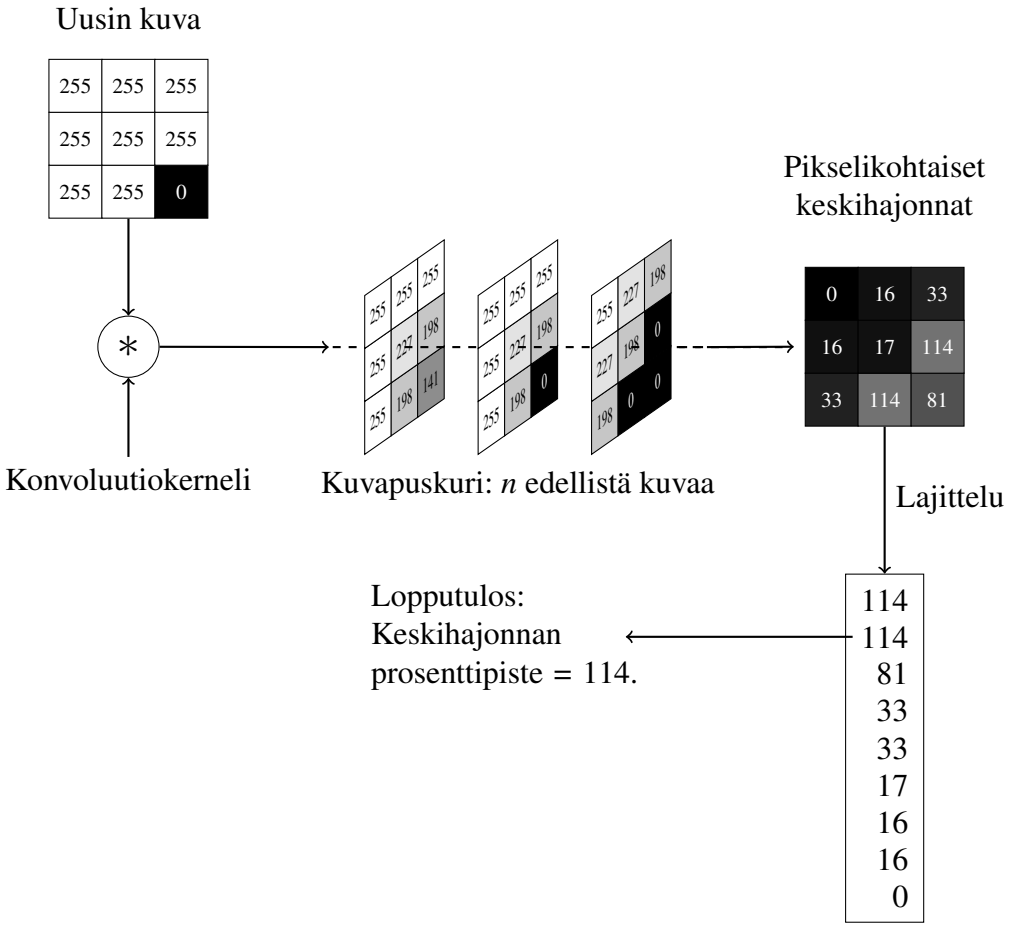
Työssä ei muutettu olemassaolevan algoritmin toimintaa, jotta sitä käyttäviin sovellusohjelmiin ja parametreihin ei tarvitse tehdä säätöjä. Alla on kuitenkin lyhyt kuvaus siitä, mikä algoritmin kunkin osan tarkoitus on.

Konvoluutio: Kuvat esikäsitellään konvoluutiolla, jonka kernelin sovellusohjelma voi vapaasti määrätä. Tähän mennessä on käytetty sumennuskerneliä pienten häiriöiden, kuten kamerasta aiheutuvan kohinan, poistoon kuvasta. Konvoluution joustavuuden vuoksi samaa menetelmää voitaisiin käyttää esimerkiksi reunantunnistukseen tai joidenkin piirteiden korostamiseen kuvassa.

Keskihajonta: Keskihajonta toimii algoritmissa muutosten suuruuden mittarina. Se on laskennallisesti raskaampi kuin esimerkiksi vaihteluvälin laskeminen minimin ja maksimin avulla. Toisaalta keskihajonta ei ole yhtä herkkä yksittäisille virhetuloksille. Nopeutuksen kannalta tulevaisuudessa kannattaisi arvioida, voisiko minimi- ja maksimilaskennalla korvata keskihajonnan.

Prosenttipiste: Keskihajonnan tuloksista lasketaan useimmiten esimerkiksi 99 % prosenttipiste. Tulokset siis järjestetään suuruusjärjestykseen, ja valitaan se tulos, joka on 99 %:n kohdalla pienimmästä tuloksesta laskien. Tarkoitus on vähentää yksittäisissä pikseleissä olevien häiriöiden vaikutusta koko tulokseen.

Kynnysarvo: Keskihajonnan tuloksista voidaan laskea myös tietyn kynnysarvon ylittävien pikselien määrä. Tarkoitus on vastaava kuin prosenttipisteen tapauksessa, mutta kynnysarvossa haluttu muutosten suuruus on määrätty ennakolta ja algoritmi laskee, montako pikseliä on muuttunut.



Kuva 3.1: *Algoritmin periaate: videokuvan kukin kuva suodatetaan konvoluutiolla, jonka jälkeen lasketaan pikselikohtainen keskihajonta n edellisen kuvan ajalta ja siitä prosenttipiste.*

3.2 Algoritmin rajapinta

Algoritmin sovellusohjelmalle näkyvä rajapinta on koostuu pääosin neljästä funktiosta:

Alustus: Alustusfunktiolle kerrotaan kuvan leveys, korkeus, kuvatyyppi, puskurin pituus sekä käytettävä konvoluutiokerneli. Se varaa tarvittavat muistipuskurit ja esikäsittelee konvoluutiokernelin. Paluuarvona on osoitin, jonka muut funktiot ottavat parametrina. Kutsumalla alustusfunktiota useita kertoja voidaan samanaikaisesti käsitellä useaa kuvavirtaa.

Kuvan syöttö: Kuvan lisäysfunktiolle annetaan osoitin uuteen kameralta tai tiedostolta saatuun kuvaan. Funktio suorittaa konvoluution kuvalle ja tallentaa tuloksen sisäiseen puskuriin, sekä päivittää keskihajonnassa käytetyt summataulukot.

Keskihajonnan laskenta: Keskihajonnan laskeva funktio käy läpi muistissa pidetyt summataulukot, joiden perusteella se laskee keskihajonnan kullekin pikselille. Tuloksista voidaan laskea joko prosenttipiste, kynnysarvon ylittävien pikselien määrä tai palauttaa jokaisen pikselin keskihajonta taulukossa.

Muistin vapautus: Purkamisfunktio poistaa alustusfunktion varaamat puskurit ja vapauttaa muistin.

Virhetilanteet käsitellään palauttamalla funktiosta virhekoodi, joka kertoo virhetilanteen nimen. Yksinkertaisuuden vuoksi virhekoodi on pelkkä osoitin merkkijonoon, jonka sisältö voi olla esimerkiksi `"OUT_OF_MEMORY"`. Jos virhettä ei tapahtunut, virhekoodi on nolla.

3.3 Tuetut kuvatyypit

Aiemmassa prototyypissä on käytetty pelkkiä harmaasävykuvia. Tähän versioon toteutettiin kuitenkin tuki yhteensä kolmelle erilaiselle kuvatyypille. Harmaasävykuvista tuetaan 8-bittisiä harmaasävyarvoja, ja värikuvista 24- ja 32-bittisiä RGB-formaatteja. RGB32-tyypin tapauksessa myös jokaisen pikselin neljäs kanava, jota käytetään joskus läpinäkyvyyden ilmaisemiseen, käsitellään vastaavasti kuin värikanavat. Läpinäkyvyyttä ei varsinaisesti käytetä sovelluksissa, mutta toteutuksen kannalta se ei eroa värikanavista.

Kanavien järjestyksellä kuvissa ei ole varsinaisesti merkitystä algoritmin kannalta, sillä kaikki kanavat käsitellään samalla tavalla. Tämän vuoksi sama toteutus sopii yhtä hyvin myös esimerkiksi ajoittain käytetyille BGR-kuville, joissa kanavat ovat vain eri järjestyksessä. Sen sijaan videokäytössä yleisessä YUV-formaatissa kirkkaus- ja värikanavat on erotettu, eikä tämä toteutus siksi sovi niille.

Toteutus tukee kuvia, joissa värisyvyys on 8 bittiä jokaista värikanavaa kohti. Tulevaisuudessa voi olla tarvetta tukea myös 16-bittisiä kuvia, joita käyttäen kamera pystyy käsittelemään laajempaa valoisuusaluetta eli suurempia kontrasteja. Tämä vaatii kuitenkin kaikkien bittileveyksien kaksinkertaistamista, joten sitä ei toistaiseksi toteutettu.

4. KONVOLUUTIOLASKENTA

Konvoluutiolaskentaa käytetään algoritmissa sisääntulevien kuvien suodatukseen. Koska konvoluutio suoritetaan jokaiselle kuvalle, on se yksi tehokkuuden kannalta tärkeimmistä osuuksista.

4.1 Konvoluution matemaattinen kuvaus

Kuvankäsittelyssä kaksiulotteinen konvoluutio tarkoittaa kuvan pikseliarvojen kertomista kerneliksi kutsutun matriisin mukaisilla painokertoimilla. Matemaattisesti tuloskuvan pikselin määräävä funktio $g(x,y)$ riippuu lähtökuvan pikseleistä $f(x,y)$ ja kernelin arvoista $k(x\prime,y\prime)$ seuraavasti:

$$g(x,y) = \sum_{x\prime}\sum_{y\prime} f(x+x\prime,y+y\prime)\cdot k(x\prime,y\prime) \tag{4.1}$$

Määritelmän voi esittää myös matriisimuodossa, mutta käytetty esitystapa on kuvankäsittelyn näkökulmasta havainnollisin. Käytettävien kuvan ja kernelin koot rajoittavat funktioiden määrittelyalueet ja summien välit äärellisiksi. Tavallisesti kernelin koko on pienempi kuin kuvan, esimerkiksi 5×5 arvoa, jolloin summat ovat välillä $x\prime,y\prime \in [-2,2]$. Laskentaa on havainnollistettu esimerkillä kuvassa 4.1.

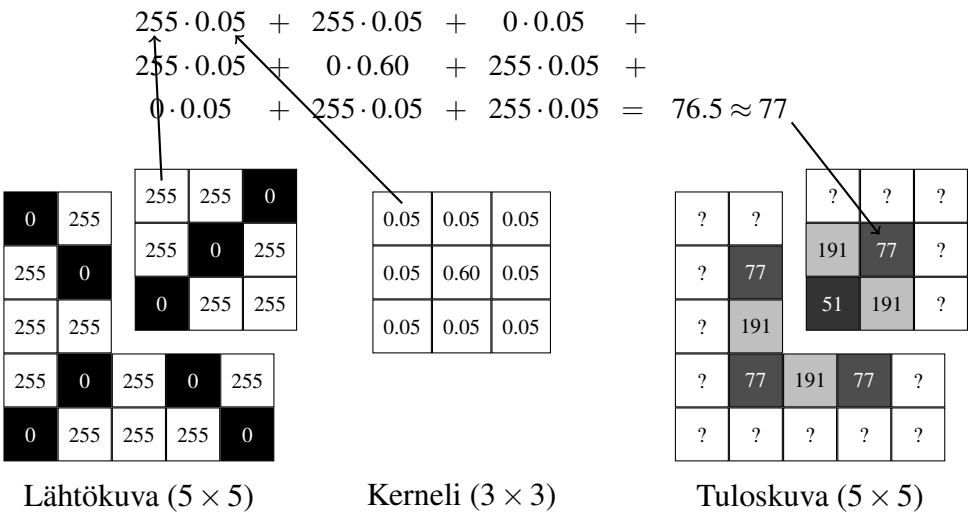
Konvoluutio vaatii perusmuodossaan yhden kertolaskun jokaista pikseliä ja kernelin arvoa kohti. Värikuvalle konvoluutiolaskenta suoritetaan erikseen kullekin värikanavalle. Kokonaisuutena kertolaskujen määrä on siis $w\cdot h\cdot k^2\cdot c$, jossa w ja h ovat kuvan leveys ja korkeus, k kernelin koko kuhunkin suuntaan ja c värikanavien määrä. Yhteenlaskujen määrä on sama, koska jokainen kertolaskun tulos lisätään summamuuttujaan.

4.1.1 Reuna-alueiden käsittely

Kuvan reuna-alueilla lähtökuvassa ei ole kaikkia kernelin painokertoimia vastaavia pikseleitä. Näiden pikselien käsittelyyn on useita tapoja, jotka on esitetty myös kuvassa 4.2:

a) Kuvan ulkopuolisten pikseliarvojen olettaminen nollaksi eli täyttäminen mustalla. Sumennuksessa tämä menetelmä aiheuttaa virheen reuna-alueille.

b) Kuvan reuna-alueiden laskematta jättäminen, jolloin syntyvä tuloskuva on lähtökuvaa pienempi.

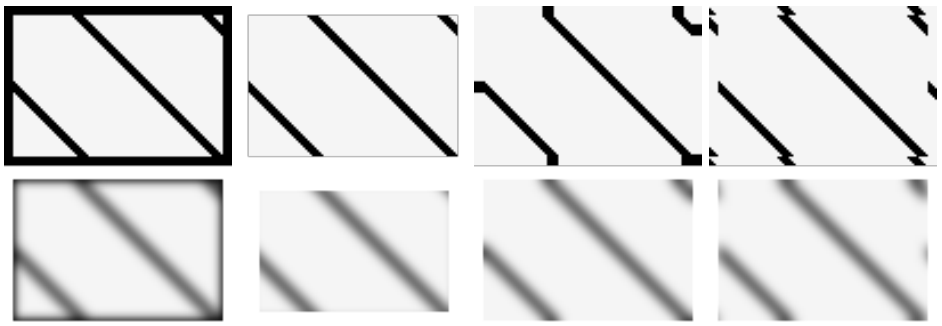


Kuva 4.1: *Konvoluution periaate. Tuloskuvan oikean yläkulman pikseliarvon laskemiseksi jokainen kernelin arvo kerrotaan vastaavalla lähtökuvan pikselillä. Kertolaskujen tulokset lasketaan yhteen ja pyöristetään kokonaisluvuksi. Reunimmaiset pikselit riippuvat käytetystä reuna-alueiden käsittelytavasta.*

c) Kuvan jatkaminen kopioimalla reunimmaiset pikseliarvot kuvan ulkopuolelle.

d) Kuvan toistaminen käyttämällä kuvan vastapäisellä reunalla olevia pikseliarvoja.

Näistä toteutukseen valittiin tuettavaksi reuna-alueiden jatkaminen ja reunojen leikkaaminen. Reuna-alueiden jatkaminen tuottaa luonnollisen tuloksen sumennusta käytettäessä, ja on ollut algoritmissa jo aiemmin käytössä. Leikkaamisen toteutus puolestaan on yksinkertainen ja on hyödyllinen käsiteltäessä pienempää aluetta suuresta kuvasta.



a) Täyttö mustalla b) Leikkaaminen c) Jatkaminen d) Toistaminen

Kuva 4.2: *Vertailu konvoluution reuna-alueiden käsittelytavoista sumennuksen yhteydessä. Ylempi kuva on lähtökuva kullakin täydennystavalla käsiteltynä, ja alempi kuva on tulos.*

4.1.2 Separoituvat kernelit

Useat käytännön kannalta olennaiset kernelit, kuten Gaussian-sumennus ja keskiarvosuodatus, voidaan jakaa pystyvektorin $\mathbf{Y}$ ja vaakavektorin $\mathbf{X}$ tuloksi seuraavasti:

$$k(x', y') = \mathbf{X}_{x'} \cdot \mathbf{Y}_{y'}$$

Tällöin konvoluutio voidaan suorittaa kahdessa osassa, eli erikseen vaaka- ja pystysuunnassa. Myöhemmin havaitaan, että toteutuksen kannalta pystysuunnan konvoluutio on hyödyllistä suorittaa ensin. Tulos on yhtenevä aikaisemman konvoluution määritelmän kanssa:

$$\begin{aligned} g_1(x,y) &= \sum_{y'} f(x, y+y') \cdot \mathbf{Y}_{y'} \\ g(x,y) &= \sum_{x'} g_1(x+x', y) \cdot \mathbf{X}_{x'} \\ &= \sum_{x'} \left(\sum_{y'} f(x+x', y+y') \cdot \mathbf{Y}_{y'} \right) \cdot \mathbf{X}_{x'} \\ &= \sum_{x'} \sum_{y'} f(x+x', y+y') \cdot \underbrace{\mathbf{Y}_{y'} \cdot \mathbf{X}_{x'}}_{=k(x',y')} \end{aligned}$$

Kummassakin osiossa tarvitaan nyt jokaista pikseliarvoa kohti k kertolaskua, eli yhteensä vain $2k$ aiemman k^2 sijaan. Ero laskutoimitusten määrässä on esimerkiksi 5×5 kernelillä 2,5-kertainen ja 15×15 kernelillä 7,5-kertainen. Separoituvan kernelin käyttö on esitetty myös kuvassa 4.3.

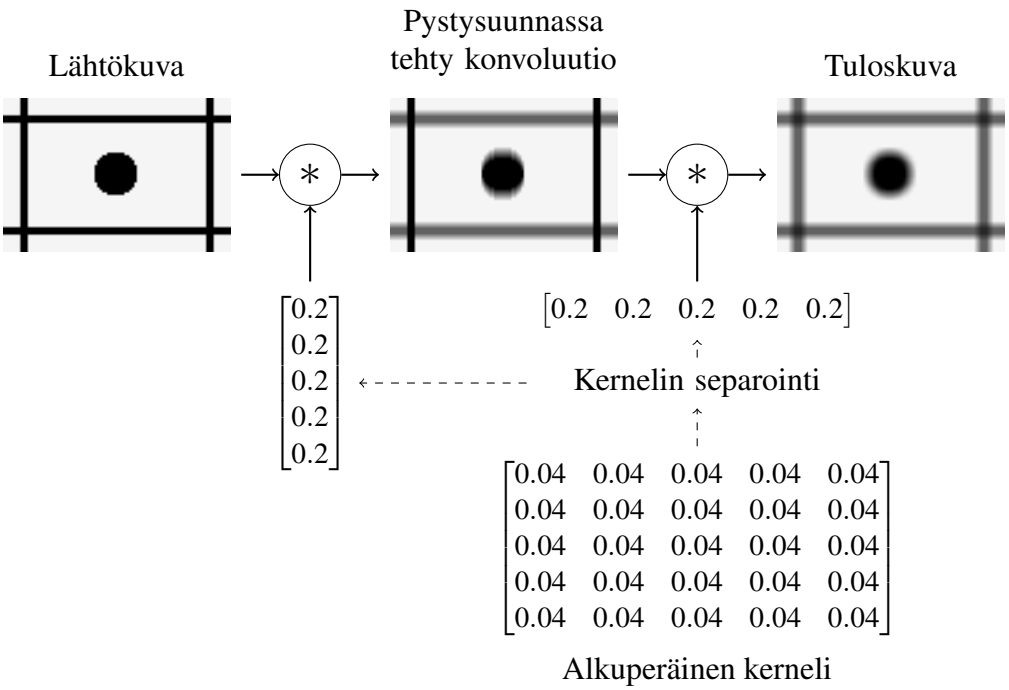
Separoidun kernelin laskeminen

Separoituvuuden hyödyntämiseksi ohjelman täytyy pystyä testaamaan, voidaanko annettu kerneli jakaa pysty- ja vaakavektoreiksi, sekä laskemaan nämä vektorit. Tämä voidaan toteuttaa esimerkiksi pääakselihajotelman (engl. Singular Value Decomposition) avulla [10]. Pääakselihajotelman toteuttaminen ilman ulkoista lineaarialgebrakirjastoa olisi kuitenkin hyvin työlästä, joten alla esitetty yksinkertainen rivin ja sarakkeen valintaan perustuva menetelmä on parempi. Menetelmää ei ole kuitenkaan kuvattu kirjallisuuslähteissä, joten sen todistus on esitetty alla.

Valitaan x_0 ja y_0 siten, että $k(x_0, y_0) \neq 0$. Tällaiset indeksit ovat olemassa ellei kerneli sisällä pelkkää nollaa. Separoidut vaaka- ja pystyvektorit ovat tällöin

$$\mathbf{X}_{x'} = \frac{1}{k(x_0, y_0)} \, k(x', y_0) \tag{4.2}$$

$$\mathbf{Y}_{y'} = k(x_0, y') \, . \tag{4.3}$$



Kuva 4.3: *Separoidun kernelin käyttäminen. Kun konvoluutio suoritetaan kahdessa osassa, kertolaskuja tarvitaan vähemmän. Kuvassa konvoluutiota on merkitty asteriskilla* $*$.

Separoituvuus voidaan testata yksinkertaisesti kertomalla vektorit $\mathbf{X}$ ja $\mathbf{Y}$ ja vertaamalla tulosta alkuperäiseen kerneliin.

Todistus. Oletetaan, että annettu kerneli on separoituva eli on olemassa vektorit $\mathbf{U}$ ja $\mathbf{V}$ siten, että kernelin arvo $k(x\prime,y\prime) = \mathbf{U}_{x\prime} \cdot \mathbf{V}_{y\prime}$ kaikilla indekseillä $x\prime$ ja $y\prime$. Tällöin todistettavina olevien kaavojen 4.2 ja 4.3 tulokset ovat

$$X_{x\prime} = \frac{1}{k(x_0,y_0)}\, k(x\prime,y_0) = \frac{1}{\mathbf{U}_{x_0} \cdot \mathbf{V}_{y_0}}\, \mathbf{U}_{x\prime} \cdot \mathbf{V}_{y_0}$$
$$Y_{y\prime} = \ldots = \mathbf{U}_{x_0} \cdot \mathbf{V}_{y\prime}$$

Tällä kernelillä suoritetun konvoluution tulos on

$$\begin{aligned} g(x,y) &= \sum_{x\prime}\sum_{y\prime} f(x+x\prime,y+y\prime) \cdot \mathbf{X}_{x\prime} \cdot \mathbf{Y}_{y\prime} \\ &= \sum_{x\prime}\sum_{y\prime} f(x+x\prime,y+y\prime) \cdot \frac{1}{\mathbf{U}_{x_0} \cdot \mathbf{V}_{y_0}} \cdot \mathbf{U}_{x\prime} \cdot \mathbf{V}_{y_0} \cdot \mathbf{U}_{x_0} \cdot \mathbf{V}_{y\prime} \\ &= \sum_{x\prime}\sum_{y\prime} f(x+x\prime,y+y\prime) \cdot \mathbf{U}_{x\prime} \cdot \mathbf{V}_{y\prime} \end{aligned}$$

eli sama kuin alkuperäisellä kernelillä saatu. □

4.2 Toteutustapa

Konvoluution toteutuksessa pääpaino on ollut laskennan nopeuttamisessa kokonaislukujen ja SSE2-käskykannan käytöllä. Laskenta suoritetaan 8- ja 16-bittisillä kokonaisluvuilla 16 tavun lohkoissa, mikä suoraan nostaa nopeuden moninkertaiseksi verrattuna jokaisen pikselin laskemiseen erikseen. Lisäksi kuva jaetaan pystysuunnassa osiin eri prosessoreilla käsiteltäväksi.

Tietyillä kerneleillä laskentaa voidaan nopeuttaa. Separoituvat kernelit tunnistetaan automaattisesti, ja tällöin käytetään niille optimoitua toteutusta. Separoituvuudesta saatava etu riippuu kernelin koosta, ja mittausten perusteella esimerkiksi 5×5 kernelillä nopeus puolitoistakertaistuu. Myös yksikkökerneli, jolla tehty konvoluutio ei muuta kuvaa, tunnistetaan ja tällöin konvoluutio ohitetaan. Tämänkin nopeusetu on melko pieni, sillä pullonkaulaksi muodostuu kuvan summaaminen varianssipuskuriin.

Ohjelmakoodia on nopeutettu myös hyödyntämällä kääntäjän automaattisia optimointeja. Esikääntäjämakrojen avulla algoritmista käännetään erillinen versio jokaiselle pikseliformaatille, jolloin ajonaikaiset vertailut vähenevät. Lisäksi reuna-alueiden tarkistuksia ei tehdä kuvan keskiosassa, vaan ainoaan reunimmaisissa lohkoissa.

4.2.1 Kokonaislukulaskenta

Algoritmin käsittelemät pikseliarvot ovat valmiiksi 8-bittisiä kokonaislukuja. Laskentaa varten myös liukulukuina annetut kernelin arvot on skaalattava jollain vakiolla, ja pyöristettävä lähimpään kokonaislukuun.

Kernelin arvot ovat tavallisesti välillä $[0, 1]$, mutta esimerkiksi reunantunnistuksessa arvot voivat vaihdella laajemmalla alueella ja myös negatiivisiksi. Parhaan tarkkuuden saamiseksi skaalauskerroin kannattaa siis sovittaa käytettävään kerneliin.

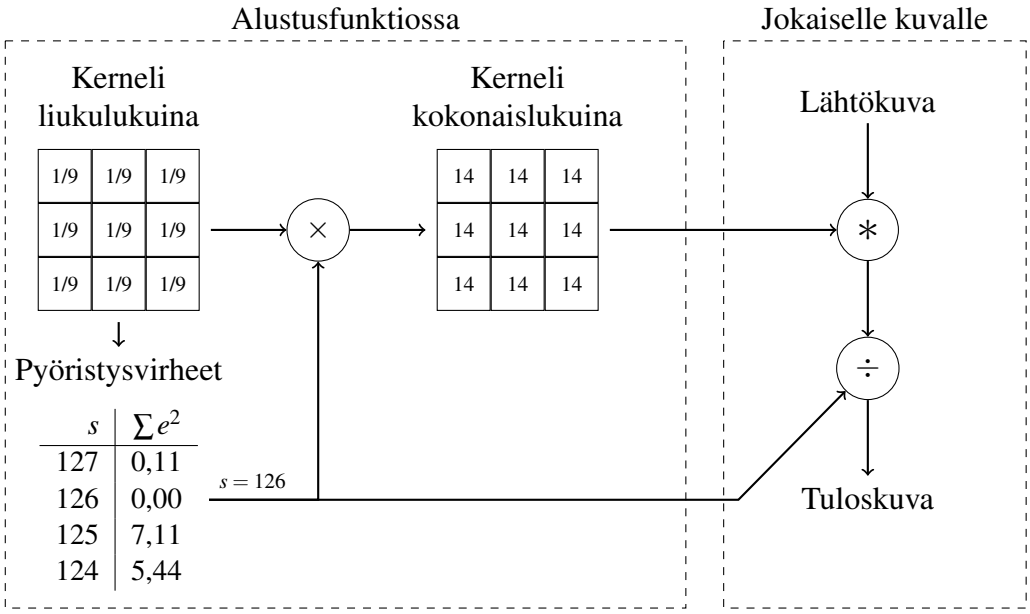
Ehdot skaalauskertoimelle

Konvoluutiota laskettaessa pikseliarvojen summa kerätään etumerkilliseen 16-bittiseen muuttujaan saturoivan yhteenlaskukäskyn avulla. Laskutavasta johtuen kernelin arvoille on tiettyjä rajoituksia joilla vältetään laskutoimitusten ennenaikainen saturoituminen.

Konvoluution 16-bittisen summan maksimiarvo on `0x7FFF`. Jotta tämä vastaisi suurinta pikseliarvoa 255, voi skaalauskerroin olla enintään 127. Summan saturoitumista käytetään hyödyksi siten, että kernelin arvojen summa voi olla yli 127, jolloin liian suuret tulokset leikkaantuvat suurimpaan mahdolliseen arvoon.

Saturoituva yhteenlasku suoritetaan vasta kertolaskun jälkeen, joten kertolaskun ylivuodon välttämiseksi itseisarvoltaan suurin kernelin arvo saa olla skaalattuna enintään ± 127. Tällöin 8-bittisen pikseliarvon kanssa tehtävän kertolaskun tulos mahtuu 16 bittiin.

Saturoitumista ei voida kuitenkaan käyttää jos kernelissä on negatiivisia arvoja. Tällöin summa voisi saturoitua kesken laskutoimituksen ja myöhempi vähennyslasku tuottaisi



Kuva 4.4: *Skaalauskertoimen s käyttö konvoluutiossa. Kuvassa on esitetty skaalauskertoimen valinta pyöristysvirheiden perusteella, kernelin pyöristys sekä tuloksen jakaminen. Pyöristysvirhe $\sum e^2$ on tässä tapauksessa 0, sillä 126 on jaollinen 9:llä.*

väärän tuloksen. Siksi negatiivisia arvoja sisältävien kernelien kanssa täytyy tarkistaa, että skaalattujen positiivisten arvojen summa on enintään 127, ja vastaavasti skaalattujen negatiivisten arvojen summa enintään -127.

Skaalauskertoimen valinta

Skaalauskertoimen valinnalla pyritään minimoimaan pyöristysvirheet kernelin arvoissa. Kerneli on laskennan ajan kiinteä, eikä alustukseen kuluvalla ajalla ole suurta merkitystä. Lisäksi mahdollisia skaalauskertoimen arvoja on enimmillään 127 yllä esitetyistä ehdoista johtuen.

Skaalauskertoimen yläraja $\hat{s}$ saadaan määritettyä suoraan ehtojen perusteella. Alustuksessa käydään läpi kaikki skaalauskertoimen kokonaislukuarvot $s \in [2, \hat{s}]$ ja lasketaan pyöristysvirheistä johtuvan virheen neliösumma. Skaalauskertoimeksi valitaan se arvo, joka tuottaa pienimmän virheen.

Koko ketju skaalauskertoimen valinnasta sen käyttöön on esitetty kuvassa 4.4.

4.2.2 Keskialueen laskenta

Kuvan keskialueella konvoluution laskenta on yksinkertaisin, koska reuna-alueita ei ole huomioitavana. Tuloskuva muodostuu 16 tavun lohkoista, joista kukin lasketaan seuraavasti:

1. Alustetaan muuttujat `sum_low` ja `sum_high` nollaksi.
2. Käydään jokainen kernelin indeksi läpi. Jos indeksiä vastaava kernelin arvo ei ole nolla:
   (a) Ladataan kernelin indeksiä vastaavasta kohdasta 16 tavun lohko lähtökuvaa muistiin.
   (b) Pikseliarvoille ja kernelin kokonaislukuarvolle suoritetaan 16-bittinen etumerkillinen kertolasku.
   (c) Kertolaskun tulos lisätään summamuuttujiin 16-bittisellä saturoituvalla yhteenlaskulla.
3. Summa jaetaan skaalauskertoimella s kertolaskun avulla:
   (a) Summaan lisätään puolet skaalauskertoimesta.
   (b) Summa kerrotaan skaalauskertoimen käänteisluvulla $(256 \cdot 256)/s$ ja tuloksesta jätetään pois 16 alinta bittiä.
4. Summat palautetaan etumerkittömiksi 8-bittisiksi luvuiksi siten, että ylimenevät arvot saturoituvat.
5. Tulos tallennetaan tuloskuvan seuraaviin 16 tavuun, ja paikkalaskuria kasvatetaan 16:lla.

Jokaista tuloskuvan tavua varten kerätään 16-bittinen etumerkillinen summa konvoluution määritelmän 4.1 mukaisesti. Koska lohkon koko on 16 tavua, tarvitaan summien tallentamiseen kaksi 128-bittistä SSE2-muuttujaa. Kaikki värikanavat käsitellään samalla tavalla, joten tässä kohtaa ei tarvitse huomioida käytettävää pikseliformaattia.

Kohdassa 2 ohitetaan suoraan kaikki kernelin sisältämät nollakertoimet, sillä ne eivät vaikuttaisi tulokseen. Näin laskenta on tehokasta, vaikka ajon aikana käytettäisiin pienempää kerneliä kuin käännösvaiheessa asetettu maksimikoko on.

Lähtökuvan lohkon paikka määritetään kohdassa 2a kernelin indeksin ja tuloskuvan lohkon perusteella. Esimerkiksi kernelin arvon $k(-1,-1)$ kohdalla vähennetään tuloskuvan kohdasta yksi rivi ja yksi sarake, ja ladataan lähtökuvan lohko tästä kohdasta. Sarakekoordinaatissa huomioidaan pikseliformaatin koko, eli esimerkiksi RGB24-kuvilla tavuosoitteesta vähennetään 3.

Kertolaskua varten 8-bittiset pikseliarvot on laajennettava 16-bittisiksi. Tämä laajennus tehdään etumerkittömästi, eli lisäämällä nollia eniten merkitseviksi biteiksi, koska pikseliarvot ovat positiivisia. Itse kertolasku suoritetaan kuitenkin etumerkillisenä negatiivisten kernelin arvojen huomioimiseksi.

Kohdassa 2c käytetään saturoituvaa yhteenlaskua, jolloin laskutoimituksen tulos jää maksimi- tai minimiarvoon ylivuodon sijaan. Saturoituvuutta on käytetty hyväksi kernelin skaalauskertoimen valinnassa, jolloin skaalauskerroin voi olla suurempi ja saavutetaan parempi tarkkuus pyöristysvirheiden suhteen. Lisäksi SSE2-käskykannassa on valmis käsky saturoituvalle yhteenlaskulle, joten sen käyttö ei vaikuta nopeuteen.

Oikean lopputuloksen saamiseksi summa pitää jakaa käytetyllä skaalauskertoimella. Hitaan jakolaskun välttämiseksi se tehdään kertomalla summa skaalauskertoimen käänteisluvulla. Käänteisluku voidaan laskea silmukan ulkopuolella, jolloin se ei vaikuta laskennan nopeuteen.

Pyöristystä varten summaan lisätään kohdassa 3a puolet skaalauskertoimesta, jolloin aikaansaadaan pyöristys positiivisille luvuille. Periaatteena on yhteys $x/s + \frac{1}{2} = (x + \frac{s}{2})/s$. Negatiiviset luvut saturoituvat joka tapauksessa tuloksessa nollaksi, joten niiden pyöristyksellä ei ole merkitystä.

SSE2-käskykannassa on kohtaan 3b sopiva erityinen käsky, joka suorittaa 16×16-bittisen kertolaskun ja palauttaa tuloksen 16 ylintä bittiä. Kertolaskun tulos muunnetaan 8-bittiseksi saturoivasti, jolloin negatiiviset arvot pyöristyvät nollaan ja yli menevät arvot 255:een.

4.2.3 Reuna-alueiden käsittely

Pikseliarvojen jatkaminen kuvan reuna-alueilla tehdään ylläolevan listan kohdassa 2a. Jos reunaan osutaan pystysuunnassa, eli $y + y'$ on negatiivinen tai ylittää kuvan alalaidan, koko ladattavan lohkon kohtaa siirretään pystysuunnassa kuvan lähimpään reunaan. Vaakasuunnan käsittely on monimutkaisempi, koska yhtäaikaisesti käsitellään 16 vierekkäistä tavua.

Ennen lohkon lataamista testataan, onko joko $x + x'$ negatiivinen tai $x + x' + 15$ yli kuvan oikean laidan. Jos näin on, ladattavan lohkon kohtaa siirretään vaakasuunnassa siten, ettei lataus mene rivin ulkopuolelle. Muutoin kuvan alussa tai lopussa voisi tapahtua lukeminen varaamattomalta muistialueelta, ja sen seurauksena ohjelman kaatuminen. Lataamisen jälkeen lohkoa siirretään vaakasuunnassa siirto-operaatiolla ja täydennetään kopioimalla reunan pikseliarvoja.

Reuna-alueiden tarkistukset tehdään vain tarvittavissa kohdissa, eli kuvan ja rivien alussa ja lopussa. Yksittäisten lohkojen laskenta on eriytetty toiseen funktioon, jolla on `inline`-määre ja parametri, joka ohjaa ehtojen käsittelyä. Tällöin kääntäjä luo automaattisesti keskialuetta varten koodin, jossa ei ole turhia ehtolauseita hidastamassa.

4.2.4 Separoituvuuden hyödyntäminen

Kernelin separoituvuus testataan alustuksen yhteydessä. Separoinnin tulosvektorit tallennetaan kahteen taulukkoon, joille haetaan kummallekin erikseen skaalauskerroin. Jos kerneli on todettu separoituvaksi, käytetään konvoluutiossa erillistä laskentarutiinia.

Tämä laskentarutiini käyttää osaksi samaa koodia kuin yllä käsitelty. Separoituva konvoluutio lasketaan aluksi pystysuunnassa, sitten tulos tallennetaan, ja sen jälkeen tehdään vastaava vaakasuunnassa tulosriville. Pystysuunnan laskenta tehdään ensiksi jotta sen tulos mahtuu käytettävissä olevalle yhdelle tulosriville. Kummankin suunnan konvoluutio on aiemmin käsiteltyä yksinkertaisempi, koska reuna-alueet ja kernelin läpikäynti täytyy tehdä vain yhdessä ulottuvuudessa.

Separoituvuutta ei kuitenkaan voida käyttää kerneleille, joissa on negatiivisia arvoja. Koska välitulos tallennetaan 8-bittisenä ilman etumerkkiä, negatiiviset välitulokset leikkautuisivat. Ongelman voisi kiertää käyttämällä erillistä puskuria, johon välitulokset tallennettaisiin 16-bittisinä. Algoritmin käyttötarkoituksessa positiiviset kernelit ovat kuitenkin yleisempiä, ja koska separoituvuudesta saatava nopeusetu on muutenkin rajallinen, ei tämän toteutukseen käytetty aikaa.

4.2.5 Rinnakkaistus

Laskenta suoritetaan tuloskuvan rivi kerrallaan. Pystysuunnan silmukka on jaettu eri prosessoreille OpenMP-käskyjen avulla. Koska jokaiseen tuloskuvan kohtaan kirjoitetaan vain yhdessä säikeessä, ei eri säikeiden välille synny riippuvuusongelmia. Jotta prosessorikohtaisen välimuistin käyttö olisi tehokasta, peräkkäiset rivit käsitellään aina samassa säikeessä. Kuva siis jaetaan esimerkiksi kahden prosessorin tapauksessa ylä- ja alaosaan, sen sijaan että rivit käsiteltäisiin vuorotellen eri prosessoreilla.

4.2.6 Kääntäjäoptimoinnit

Kääntäjä pystyy suorittamaan konvoluutioalgoritmille joitain hyödyllisiä optimointeja. Kernelin koko on käännösaikainen vakio, joten kääntäjä avaa (engl. loop unrolling) kernelin läpikäynnissä käytetyt silmukat.

Jotta käännetty kirjasto olisi joustava, pikseliformaattia ei voitu määritellä vakioksi. Sen sijaan kullekin tuetulle pikseliformaatille käännetään erillinen versio konvoluutioalgoritmista, ja suorituksen aikana valitaan mitä näistä kutsutaan. Käytännössä tämä on toteutettu määrittelemällä kääntäjän komentoriviltä esikääntäjämakro `PIXEL_SIZE`, jonka perusteella muodostetaan myös funktion nimi. Eri parametreilla käännetyt funktiot linkitetään yhteen, ja pääfunktiossa, nopeuskriittisen silmukan ulkopuolella, on ehtorakenne valitsemassa oikean version.

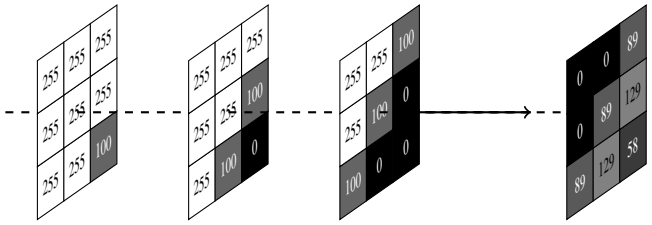
4.2.7 Rajoitukset

Suurin tuettu kernelin koko on 33×33, jolloin indeksi x' on välillä $[-16, 16]$. Rajoitus muodostuu siitä, että reuna-alueiden tarkastelussa huomioidaan vain reunimmaiset lohkot. Jos indeksi x' olisi yli 16, täytyisi tarkastelu laajentaa myös muihin lohkoihin.

Kernelin symmetrisyydellä on separoituvuuden tavoin mahdollista nopeuttaa laskentaa, mutta tätä ei ole toteutettu. Tällaiselle kernelille $k(x', y') = k(\pm x', \pm y')$, jolloin kaikki neljä lähtöpikseliä $f(x \pm x', y \pm y')$ voitaisiin laskea yhteen ennen kertolaskua. Mittausten perusteella kertolaskuihin kuluva aika muodostaa kuitenkin vain yhden neljäsosan konvoluution laskenta-ajasta, joten symmetrisyydestä saatava hyöty olisi rajallinen.

5. KESKIHAJONNAN LASKENTA

Keskihajontaa käytetään algoritmissa kuvan muutosten mittarina. Keskihajonta lasketaan pikselikohtaisesti kaikista puskurin kuvista kuvan 5.1 mukaisesti.



Kuva 5.1: *Pikselikohtaiset keskihajonnat lasketaan kaikista puskurissa olevista kuvista. Tulos kertoo sen, paljonko kukin pikseli on muuttunut.*

5.1 Keskihajonnan matemaattinen kuvaus

Keskihajonta määritellään matemaattisesti keskiarvon μ perusteella. Tämän algoritmin tapauksessa keskihajonta $\sigma(x,y)$ lasketaan puskurissa olevista kuvista $g_i(x,y)$. Kuvan indeksi i on välillä $[0,n-1]$, jossa n on alustuksessa määritelty puskurien määrä.

$$\mu(x,y) = \frac{1}{n}\sum_i g_i(x,y) \tag{5.1}$$

$$\sigma(x,y) = \sqrt{\frac{1}{n}\sum_i (g_i(x,y) - \mu(x,y))^2} \tag{5.2}$$

Joissakin yhteyksissä käytetään myös otoskeskihajontaa, jossa jakaja on $n-1$, mutta tämän algoritmin tapauksessa näin ei tehdä. Suoraan määritelmän perusteella laskettaessa tarvitaan jokaista laskentakertaa kohti $w \cdot h \cdot c \cdot n$ kertolaskua, $w \cdot h \cdot c$ neliöjuurta sekä $2 \cdot w \cdot h \cdot c$ jakolaskua. Laskutoimitusten määrää voidaan kuitenkin vähentää usealla tavalla.

5.1.1 Neliöjuurioperaation välttäminen

Neliöjuuren laskeminen on esimerkiksi kertolaskuun verrattuna huomattavan raskasta. Operaatio voidaan välttää, kun laskennassa käytetään sisäisesti varianssia $\sigma(x,y)^2$.

Rajapinnassa on aiemmin käytetty keskihajontaa, joten siirtyminen kokonaan varianssiin vaatisi muutoksia algoritmia käyttävään sovellukseen. Esimerkiksi käytetyt kynnysarvot täytyisi korottaa neliöön. Tämä muunnos voidaan kuitenkin tehdä suoraan algoritmin rajapinnassa.

Toteutus korottaa syötetyn kynnysarvon neliöön ja vastaavasti ottaa neliöjuuren tuloksena annettavasta prosenttipisteestä. Nämä operaatiot täytyy suorittaa vain kerran jokaiselle kuvalle, joten niiden tehokkuus ei ole kriittinen. Jos sovellus tarvitsee pikselikohtaiset keskihajonnat taulukossa, joudutaan neliöjuurioperaatio kuitenkin suorittamaan jokaiselle pikselille. Normaalissa käyttötilanteessa tätä toimintoa ei kuitenkaan yleensä käytetä.

5.1.2 Laskenta liukuvalla ikkunalla

Keskihajonnan määritelmässä olevat summat voidaan avata käyttäen yhteyttä $var(x) = E\left[x^2\right] - E\left[x\right]^2$ [11, s.32]. Tällöin saadaan seuraava esitysmuoto:

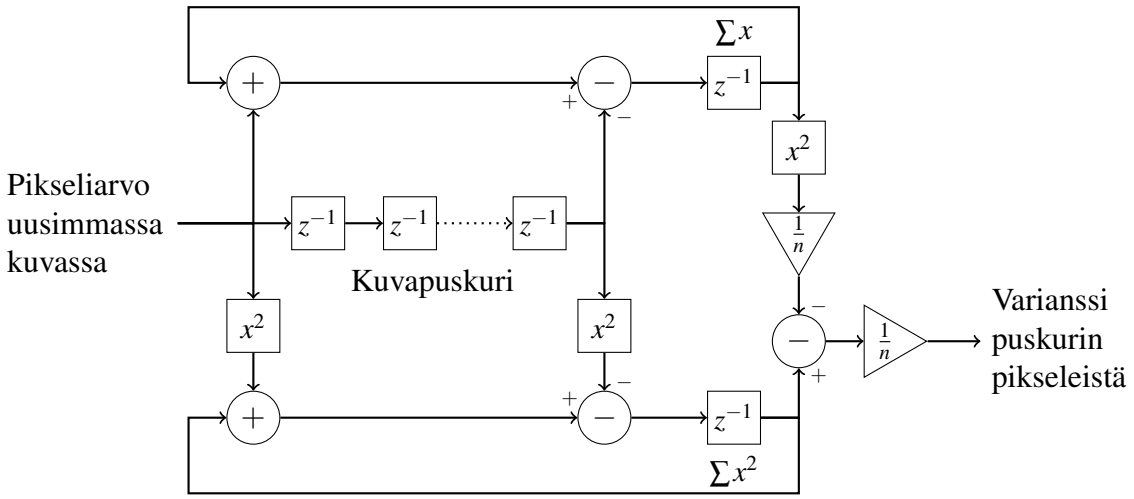
$$\sigma(x,y)^2 = \frac{1}{n}\left[\left(\sum_i g_i(x,y)^2\right) - \frac{1}{n}\left(\sum_i g_i(x,y)\right)^2\right] \tag{5.3}$$

Tässä muodossa on se etu, että summien sisällä olevat termit riippuvat vain kunkin kuvan pikseleistä, eivät kaikista puskureista lasketusta keskiarvosta. Tällaiset summat voidaan laskea liukuvan ikkunan avulla kuvan 5.2 mukaisesti. Jokainen syötetty kuva lisätään summiin, ja vastaavasti jokainen puskurista poistuva kuva vähennetään summista. Algoritmi pitää muistissaan siis kaksi taulukkoa: pikseliarvojen summan, sekä pikseliarvojen neliöiden summan.

Saavutettu etu on se, että laskutoimitusten määrä ei riipu puskurin pituudesta. Liukuvaa ikkunaa käytettäessä tarvitaan jokaista kuvaa varten $2 \cdot w \cdot h \cdot c$ kertolaskua. Lisäksi jokaista keskihajonnan laskentakertaa kohti tarvitaan vielä $w \cdot h \cdot c$ kertolaskua ja $2 \cdot w \cdot h \cdot c$ jakolaskua. Laskutoimitusten määrää on verrattu perustoteutukseen taulukossa 5.1.

Kahden puskurin tapauksessa liukuvalla ikkunalla tarvitaan enemmän laskutoimituksia kuin kohdassa 5.1. Ero on pieni verrattuna algoritmin muiden osien viemään aikaan, joten erillistä toteutusta kahden puskurin erikoistapausta varten ei tehty. Kun puskurien määrä on yli kolme, on liukuvan ikkunan menetelmä selvästi tehokkaampi.

Liukulukuja käytettäessä tässä menetelmässä esiintyy numeerisia ongelmia, sillä toisistaan vähennettävien summien ero on usein pieni [12]. Liukulukujen tarkkuus riippuu luvun koosta, joten suurien summien tapauksessa tarkkuus ei välttämättä riitä



Kuva 5.2: *Liukuvan ikkunan käyttö varianssin laskennassa. Kuvassa on esitetty summamuuttujien päivitys sekä varianssin laskenta summien arvoista yksittäiselle pikselille. Todellisuudessa summat tallennetaan taulukoihin, eli algoritmissa on kaksi summamuuttujaa jokaista pikseliä varten.*

Taulukko 5.1: *Keskihajonnan laskemiseen tarvittujen laskutoimitusten eri määrillä kuvia. Perustoteutus on kohdassa 5.1 esitetty kaava, ja liukuva ikkuna kohdan 5.3 kaava. Neliöjuurta ei kohdan 5.1.1 mukaisesti lasketa.*

	Perustoteutus		Liukuva ikkuna	
Kuvia	Kerto	Jako	Kerto	Jako
(kpl)	(laskutoimituksia per pikseli ja värikanava)			
2	2	2	3	2
3	3	2	3	2
50	50	2	3	2

summien erotuksen tarkkaan laskemiseen. Erityisesti tästä syystä algoritmissa käytetään keskihajonnan laskennassa kokonaislukuja tarkkuuden säilyttämiseksi.

5.2 Toteutustapa

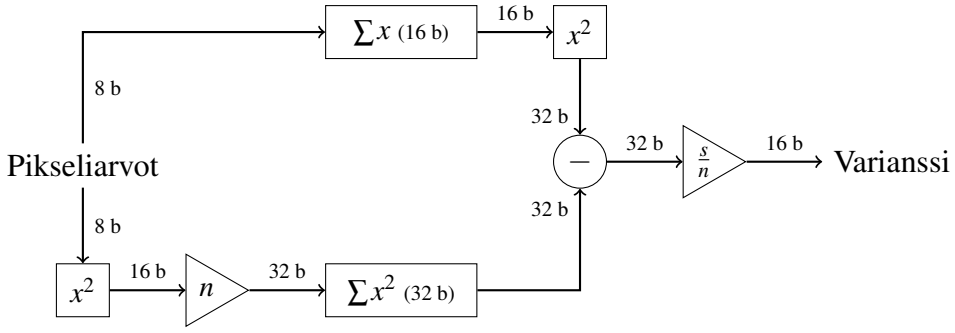
Keskihajonnan laskenta koostuu kahdesta osuudesta: summataulukoiden päivityksestä, sekä keskihajonnan laskemisesta niiden perusteella. Summataulukoiden päivitys tehdään kuvan syötön yhteydessä ja keskihajonnan laskenta erillisessä funktiossa.

Keskihajonnan toteutuksessa hyödynnettiin pääosin samoja optimointeja kuin konvoluution tapauksessa: SSE2-käskykantaa, rinnakkaistusta ja erillistä käännöstä eri pikselityypeille.

5.2.1 Lukualueet

Pikseliarvot ovat 8-bittisiä, joten kaavan 5.3 sisältämät pikseliarvojen neliöt ovat 16 bittisiä. Summataulukoille tarvittu tila puolestaan riippuu halutusta kuvapuskurien enimmäismäärästä. Esimerkiksi pikseliarvojen summa 10 puskurista on enimmillään $10 \cdot 255 = 2550$, jonka tallennus vaatii 16 bittisen muuttujan.

Tehdyssä toteutuksessa pikseliarvojen summalle varataan 16 ja neliöiden summalle 32 bittiä. Tällöin puskurien enimmäismäärä on $2^{16}/2^8 = 256$. Bittileveydet eri kohdissa laskentaa on esitetty kuvassa 5.3. Puskurien enimmäismäärää voitaisiin tarvittaessa nostaa käyttämällä suurempaa bittileveyttä summamuuttujissa.



Kuva 5.3: *Lukujen bittileveydet eri kohdissa varianssilaskentaa. Kertominen ja jakaminen n:llä on kuvassa esitetty kohdassa 5.2.3 kuvatulla tavalla.*

Varianssin tulos voi sisältää murtolukuosan, vaikka itse pikseliarvot ja välitulokset ovat kokonaislukuja. Kokonaislukulaskennassa tämä esitetään skaalauskertoimen avulla. Skaalauskertoimen valintaa varten tarvitaan varianssin suurin mahdollinen arvo, joka saadaan Popoviciuksen epäyhtälöstä [13]. Koska pikseliarvojen vaihteluväli on $[0,255]$, on varianssi enimmillään 128^2. Valitsemalla skaalauskertoimeksi $s = 4$, tulos mahtuu 16-bittiseen muuttujaan.

5.2.2 Summien kerääminen

Summien päivitys suoritetaan konvoluutiolaskennan yhteydessä. Ennen konvoluution laskemista summataulukoista vähennetään vanhin puskurissa oleva kuva, eli kuva jonka päälle konvoluutio kirjoittaa uuden kuvan. Konvoluution jälkeen suodatettu kuva summataan vastaavasti taulukkoihin.

Kumpikin operaatio suoritetaan rivi kerrallaan samalla funktiolla. Päivitysfunktio ottaa parametrin, joka määrää vähennetäänkö vai lisätäänkö annettu kuva summaan.

Laskenta suoritetaan SSE2-käskykannalla 16 tavun lohkoissa. Ladatut pikseliarvot laajennetaan 16- tai 32-bittisiksi, kerrotaan neliösumman tapauksessa itsensä kanssa ja lisätään vastaaviin kohtiin summataulukossa.

Neliösumman tapauksessa jokainen lohko joudutaan käsittelemään neljässä vaiheessa, sillä 32-bittisiä lukuja mahtuu rekisteriin vähemmän kuin 8-bittisiä. Lohko kannattaa kuitenkin ladata kerralla ja jakaa sitten useaan käsiteltävään osaan, sillä näin tarvitaan vähemmän muistioperaatioita.

5.2.3 Jakaminen puskurien määrällä

Kaavassa 5.3 on kaksi jakolaskua puskurien määrällä. Yksinkertaisimmin kokonaislukutarkkuuteen päästäisiin, kun kumpikin jakolasku suoritettaisiin erikseen kertomalla puskurien määrän käänteisluvulla. Parempi tarkkuus saavutetaan kuitenkin suorittamalla jakolaskut vasta laskennan lopuksi. Ero on siis siinä, tuleeko esimerkiksi joukon $\{0,0,1\}$ varianssiksi 0 vai 0.33.

Toteutuksessa kaavan 5.3 laskentajärjestystä on muokattu seuraavasti:

$$\sigma(x,y)^2 = \frac{s}{n^2}\left[\underbrace{\left(\sum_i n \cdot g_i(x,y)^2\right)}_{\text{32-bittinen summa}} - \underbrace{\left(\sum_i g_i(x,y)\right)}_{\text{16-bittinen summa}}^2\right] \tag{5.4}$$

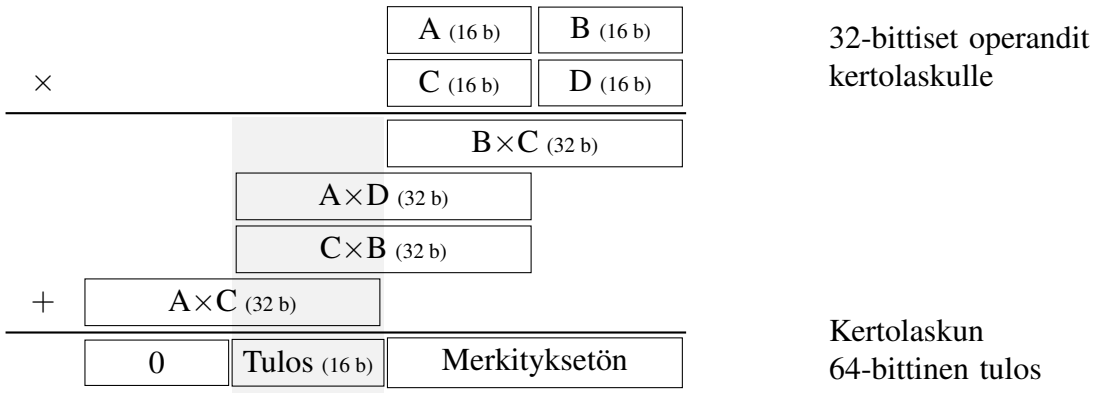
Erona on summattavien arvojen kertominen n:llä ensimmäisessä summassa, jolloin lopuksi voidaan jakaa koko tulos n^2:lla. Kertominen suoritetaan ennen summausta, sillä tällöin luvut ovat vielä 16-bittisiä ja siten nopeampi kertoa.

Tämän muutoksen vaikutus nopeuteen on pieni, mutta todennäköisesti negatiivinen. Ennen summausta tehtävä kertolasku suoritetaan liukuvan ikkunan vuoksi kahdesti jokaiselle pikselille, kun taas summan ulkopuolella se tarvitsisi laskea kerran. Tässä toteutuksessa summan ulkopuolella tarvittaisiin kuitenkin 32-bittinen kertolasku, joka olisi hitaampi kuin kaksi 16-bittistä. Jatko-optimoinnissa voitaisiin arvioida, riittääkö varianssille kokonaislukuarvo. Tällöin jälkimmäisen summan voisi jakaa suoraan n:llä 16-bittisen kertolaskun avulla.

5.2.4 Varianssin laskenta summista

Varianssia laskettaessa summat käsitellään neljän luvun lohkoissa, koska 128-bittiseen SSE2-rekisteriin mahtuu neljä 32-bittistä lukua. Konvoluution yhteydessä kerätyt summat ladataan muistista, ja 16-bittinen summa kerrotaan itsellään neliöönkorotusta varten. Nämä 32-bittiset termit vähennetään toisistaan.

Tulos täytyy vielä jakaa puskurien määrän neliöllä. Tämä tehdään kertomalla käänteisluvulla $2^{32} \cdot s/n^2$, jonka arvo on laskettu silmukan ulkopuolella. Kerrottavat luvut ovat 32-bittisiä ja kertolaskun tulos siten 64-bittinen, mutta tuloksesta jätetään 32 alinta bittiä pois. SSE2-käskykannassa ei ole suoraan sopivaa 32-bittistä kertolaskua, joten tässä käytetään 16-bittisiä kertolaskuoperaatioita kuvan 5.4 mukaisesti.



Kuva 5.4: *Koska SSE2-käskykannassa ei ole viimeisessä jakolaskussa tarvittavaa 32-bittistä kertolaskua, täytyy se suorittaa 16-bittisten kertolaskujen avulla. Tarkoitus on jakaa kertolaskun tulos 2^{32}:lla, jolloin tuloksesta jätetään pois alimmat 32 bittiä.*

5.2.5 Tulosten käsittely

Yleisimmässä käyttötapauksessa ohjelma ei tarvitse koko keskihajontataulukkoa, vaan tietyn prosenttipisteen arvoista tai valitun kynnysarvon ylittävien pikselien määrän. Nämä tulokset kannattaa laskea suoraan keskihajonnan laskevassa silmukassa, jotta tuloksia ei tarvitse välissä tallentaa erilliseen taulukkoon.

Toteutuksessa keskihajonnan laskevalle funktiolle annetaan osoittimet tulosmuuttujiin, joita on kolme: prosenttipisteen arvo, kynnysarvon ylittävä lukumäärä ja koko taulukko keskihajonnoista. Nollaosoittimen `NULL` antamalla voidaan tarpeettomat tiedot jättää laskematta.

5.2.6 Rinnakkaistus

Rinnakkaistus tapahtuu pääosin vastaavasti kuin konvoluution yhteydessä, eli jakamalla kuva vaakasuoriin osiin. Ainoa ero on siinä, että kynnysarvolaskurista ja prosenttipisteen laskentaan käytetystä taulukosta täytyy olla jokaiselle säikeelle erillinen kopio. Muussa tapauksessa säikeet saattaisivat päivittää laskuria yhtäaikaisesti, aiheuttaen väärän tuloksen. Erilliset muuttujat summataan yhteen laskennan lopuksi.

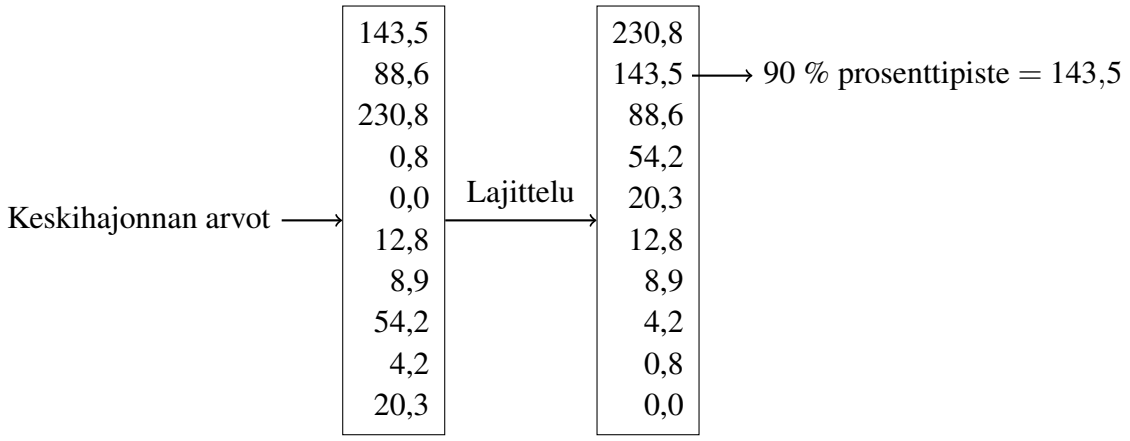
5.2.7 Rajoitukset

Keskihajonta lasketaan yhdelle värikanavalle kerrallaan. Tällöin kuva joudutaan käymään kolmesti läpi kaikkien kanavien laskemiseksi, mikä hidastaa laskentaa hieman. Aiempi prototyyppi on tukenut pelkkiä harmaasävykuvia, joten kanavat haluttiin pitää erillään kunnes tulosten käsittelytapa on selvinnyt. Mahdollisesti jatkokehityksessä kanavien

käsittely yhdistetään, jolloin keskihajonnan prosenttipiste ja muut tulokset lasketaan kaikkien kanavien yhdistetyistä tuloksista.

6. PROSENTTIPISTEEN LASKENTA

Prosenttipiste (engl. percentile) on mediaania muistuttava operaatio, jossa lukujoukko järjestetään suuruusjärjestykseen ja taulukosta valitaan tietyssä kohdassa oleva arvo. Esimerkiksi 99 % prosenttipiste on sellainen arvo, että 1 % taulukon arvoista on sitä suurempia. Periaate on esitetty myös kuvassa 6.1. Tämän algoritmin tapauksessa prosenttipistettä käytetään keskihajonnan tulosten suodatukseen, jolloin häiriöt yksittäisissä pikseleissä suodattuvat pois.



Kuva 6.1: *Prosenttipisteen laskeminen. Keskihajonnan tulokset lajitellaan, ja lajitellusta taulukosta valitaan tietyssä kohdassa oleva arvo.*

6.1 Prosenttipisteen matemaattinen kuvaus

Prosenttipisteelle on esitetty useita vaihtoehtoisia määritelmiä, jotka eroavat siinä, miten prosenttiluku pyöristetään taulukon indeksiksi. Tämän työn tapauksessa käytetään seuraavaa määritelmää. Prosenttipiste $P(p)$ on arvo, jonka kanssa yhtäsuuria tai pienempiä arvoja on $p \cdot N/100$ kappaletta, missä kappalemäärä pyöristetään lähimpään kokonaislukuun. Käsiteltävien lukujen lukumäärä N on tämän algoritmin tapauksessa

pikselien lukumäärä eli $w \cdot h$. Kukin värikanava käsitellään siis erikseen.

$$\begin{aligned}
&\text{Järjestetään luvut } v_0 \ldots v_{N-1} \text{ siten, että } v_0 \leq v_1 \leq \ldots \leq v_{N-1} \\
&R = \text{round}(\frac{p \cdot N}{100}) \\
&P(p) = v_R
\end{aligned}$$

Useimpien järjestysalgoritmien vaatima aika on $O(N\log(N))$. Prosenttipisteen laskemiseen ei kuitenkaan tarvita täyttä lajittelua.

6.1.1 Nopeutus valinta-algoritmeilla

Prosenttipisteen tehokkaasti laskevia algoritmeja kutsutaan yleisesti valinta-algoritmeiksi, ja ne ovat nopeimmillaan $O(N)$ ajankäytöltään ja $O(1)$ muistinkäytöltään [14]. Tavallisin rakenne näissä algoritmeissa on valita taulukosta satunnainen luku kynnysarvoksi, ja jakaa taulukko sitten kahteen osaan sen mukaan ovatko luvut pienempiä vai suurempia kuin kynnysarvo. Saatujen kahden taulukon pituuksista voidaan päätellä, kummassa taulukossa haluttu prosenttipiste on, ja tämän jälkeen jakamista jatketaan rekursiivisesti. Algoritmin tehokkuus määräytyy lähinnä siitä, kuinka hyvin kynnysarvo valitaan. Parhaassa tapauksessa kynnysarvoksi osuu taulukon mediaani, jolloin lajiteltavien lukujen määrä puolittuu jokaisella rekursiotasolla.

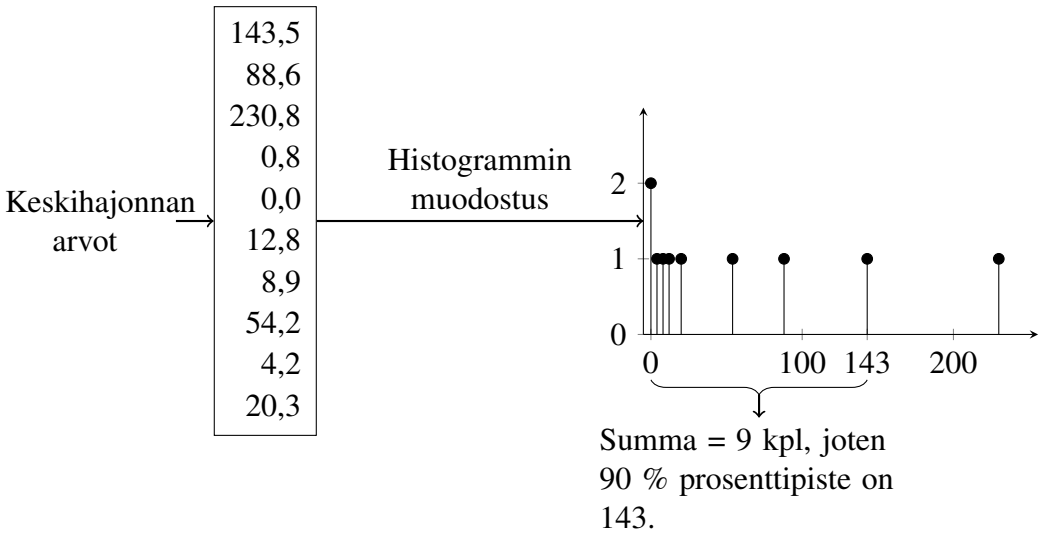
Valinta-algoritmit ovat yleisessä tapauksessa tehokkain tapa prosenttipisteen laskemiseen, mutta tämän työn tapauksessa tulosten arvoalue on rajoitettu. Sen vuoksi laskeminen saadaan vielä nopeammaksi muodostamalla histogrammi tuloksista.

6.1.2 Nopeutus histogrammilla

Histogrammi muodostetaan laskemalla jokaisen arvon lukumäärä taulukossa. Saadut lukumäärät käydään läpi pienimmästä arvosta alkaen ja summataan yhteen, kunnes summa on yhtäsuuri tai ylittää prosenttipisteen määritelmän mukaisen indeksin R. Prosenttipiste on tällöin se arvo, johon summaaminen lopetettiin. Periaate on esitetty myös kuvassa 6.2.

Histogrammin muodostaminen on ajankäytöltään $O(N)$, mutta muistinkäyttö riippuu käsiteltävästä arvoalueesta. Koska varianssin tulokset ovat 16-bittisiä, riittää histogrammitaulukon kooksi 65 536 lukua.

Histogrammin laskeminen on ohjelmallisesti hyvin yksinkertaista: histogrammin muodostamiseen tarvitaan pelkkä taulukon indeksointi ja yksi yhteenlasku jokaista pikseliä varten. Prosenttipisteen laskeminen histogrammista puolestaan vaatii enimmillään 65 536:n luvun yhteenlaskun. Vaikka valinta-algoritmit siis kuluttavat vähemmän muistia ja ovat ajankäytöltään samaa O-luokkaa, vaatii niiden laskenta enemmän käskyjä kuin histogrammi.



Kuva 6.2: *Histogrammin käyttö prosenttipisteen laskennassa. Kunkin keskihajonnan arvon esiintymismäärä taulukossa lasketaan, ja tulokset muodostavat histogrammin. Histogrammi käydään pienimmästä arvosta alkaen läpi ja summataan yhteen, kunnes summa ylittää prosenttipistettä vastaavan indeksin.*

6.2 Toteutustapa

Toteutus jakautuu kahteen osaan: histogrammin keräämiseen, ja prosenttipisteen laskentaan histogrammista. Näistä edellinen suoritetaan jo keskihajonnan laskennan kanssa samassa silmukassa ja jälkimmäinen sen jälkeen.

Konvoluutiosta ja keskihajonnasta poiketen prosenttipisteen laskennassa ei käytetä SSE2-käskykantaa. Eniten aikaa kuluu taulukon indeksointiin histogrammia kerättäessä. Tässä on kyse epäsuorasta muistin osoituksesta, jonka nopeutuksessa SIMD-käskyistä ei ole apua.

6.2.1 Lukualueet

Histogrammitaulukon alkiot tarkoittavat pikselien lukumääriä, joten niiden arvoalue riippuu kuvan koosta. Esimerkiksi 640×480 pikselin kuvassa on 307 200 pikseliä. Jos kuvassa ei ole yhtään muutoksia, on kaikkien pikselien varianssi 0, ja siten histogrammin ensimmäinen arvo on 307 200 ja muut arvot 0.

Sopiva bittileveys histogrammitaulukolle on siten 32 bittiä. Tämä riittää enintään 2^{32} pikselin kokoisille kuville, eli esimerkiksi $65\,536 \times 65\,536$ pikselin kokoon asti.

6.2.2 Histogrammitaulukon kerääminen

Histogrammitaulukko alustetaan nollaksi ennen varianssilaskennan silmukkaa. Varianssilaskennassa tulokset saadaan neljän luvun lohkoissa, joka sitten käydään luku kerrallaan läpi. Jokainen luku vastaa yhden pikselin varianssia, ja on arvoalueeltaan 16-bittinen. Histogrammitaulukkoa indeksoidaan tällä luvulla ja kasvatetaan taulukon arvoa, eli `histogrammi[varianssi]++;`.

Laskennan nopeuden kannalta suurin merkitys on muistioperaation nopeudella. Koska taulukon koko on $65536 \cdot 4 = 262144$ tavua $= 256$ kilotavua, mahtuu se nykyaikaisten prosessorien toisen tason välimuistiin kokonaan. Lisäksi jos kuvassa on vain vähän muutoksia, eniten käytetään histogrammin ensimmäisiä alkioita jotka vastaavat pieniä varianssin arvoja. Tällöin ne mahtuvat ensimmäisen tason välimuistiin, ja niiden käsittely on hyvin nopeaa.

6.2.3 Rinnakkaistus

Varianssi lasketaan rinnakkaisesti, joten samassa silmukassa suoritettava histogrammitaulukon päivitys suoritetaan myös rinnakkain. Jotta eri säikeissä tapahtuvat laskennat eivät rikkoisi toistensa tuloksia, jokaiselle säikeelle varataan oma taulukko.

Säikeiden määrä määräytyy sen mukaan, montako prosessoriydintä laitteistossa on. Tarvittavien taulukoiden määrää ei siksi tiedetä käännösvaiheessa. Taulukot varataan siksi dynaamisesti `calloc()`-kutsulla aina ennen laskentaa. Muistivaraus vie jonkin verran aikaa, ja sen voisi periaatteessa siirtää algoritmin alustukseen. Käytännössä ajankäyttö on testatuilla alustoilla kuitenkin merkityksettömän pieni, enemmän aikaa kuluu histogrammitaulukon nollaamiseen, mikä täytyy tehdä joka tapauksessa aina ennen laskentaa.

6.2.4 Prosenttipisteen hakeminen histogrammista

Prosenttipisteen laskemiseksi histogrammin alkioita summataan järjestyksessä yhteen, kunnes summa ylittää halutusta prosenttipisteestä määräytyvän arvon. C-kielellä esitettynä laskenta tapahtuu näin:

```
int pp_indeksi = (int)roundf(leveys * korkeus * prosenttipiste);
int varianssi;
int summa = 0;
for (varianssi = 0; varianssi < HISTOGRAMMI_KOKO; varianssi++)
{
    for (int saie = 0; saie < saikeiden_maara; saie++)
    {
        summa += histogrammit[saie][varianssi];
```

```
    }

    if (summa >= pp_indeksi)
        break;
}

float keskihajonnan_prosenttipiste = sqrtf((float)varianssi / 4);
```

Taulukon läpikäyntiä voisi ajatella nopeutettavan SSE2-käskyillä, siten että kerralla summattaisiin yhteen kokonainen 16 tavun lohko. Testien perusteella pullonkaula on kuitenkin muistihauissa, sillä lohkoissa tehtävä summaus ei nopeuttanut laskentaa lainkaan.

6.2.5 Rajoitukset

Merkittävin rajoitus prosenttipisteen laskennassa on se, että histogrammitaulukon koko rajoittaa prosenttipisteen tarkkuutta. Tarkkuus riittää hyvin 8-bittisille luvuille, mutta 16-bittisellä värisyvyydellä taulukon kokoa jouduttaisiin kasvattamaan tai pienentämään tuloksen tarkkuutta.

7. NOPEUSMITTAUSTEN TULOKSET

Algoritmin nopeutta arvioitiin eri alustoilla ja testikuvilla suoritettujen nopeusmittausten perusteella. Nopeutta verrattiin aiempaan prototyyppiin, referenssitoteutukseen sekä julkaistuihin mittauksiin joidenkin laskentakirjastojen nopeuksista.

7.1 Mittaustavat

Toteutettua kirjastoa testattiin lyhyen C-kielisen testiohjelman avulla. Aiemman prototyypin nopeus pystyttiin vain arvioimaan, koska sen liittäminen testiohjelmaan olisi ollut vaikeaa. Lisäksi testauksessa käytettiin `perf`-ohjelmaa [15] [16], jolla pystyttiin tarkastelemaan prosessorin toimintaa tarkemmin.

7.1.1 Testikuvat

Testikuvina käytettiin digitaalikameran sarjavalotuksella otettua kuvasarjaa sekuntikellosta. Nämä kuvat muunnettiin kutakin testiä varten oikeaan kokoon ja värimuotoon.

7.1.2 Testiohjelma

Toteutetun optimoidun version sekä referenssitoteutuksen nopeus mitattiin C-kielisen testiohjelman avulla. Testiohjelma lataa muistiin annetut kuvat ja suorittaa sitten algoritmin näille kuville. Algoritmin käyttämä aika mitataan Linux-alustalla `gettimeofday`-funktiolla sekä Windows-alustalla `QueryPerformanceCounter`-funktiolla.

Kummankin mittaustavan tarkkuus on noin mikrosekunnin suuruusluokkaa ja kutsun käyttämä aika joitakin kymmeniä mikrosekunteja. Mittaukseen kulunut aika ei siis vaikuta merkittävästi algoritmin ajankäyttöön, joka on useita millisekunteja. Mittausten tarkkuus riippuu jonkin verran käytetystä alustasta, ja funktioissa voi esiintyä joskus ongelmia [17]. Tämän työn yhteydessä ei kuitenkaan tarvittu kovin suurta tarkkuutta, joten eri mittaustapojen analysointiin ei käytetty aikaa.

Testiohjelmaa käytettäessä käsiteltävät on ladattu valmiiksi muistiin algoritmia varten. Tämä vastaa todellista käyttötilannetta, jossa kuva olisi muistissa kameralta lukemisen jälkeen.

7.1.3 Prototyypin arviointi

Mittausohjelman liittäminen aiempaan prototyyppiin olisi ollut työlästä, koska se on kiinteästi integroitu sovellusohjelmaan. Sen vuoksi prototyypin nopeus piti arvioida sovellusohjelman nopeuden perusteella.

Kuvaa analysoitaessa suurin osa sovellusohjelman ajasta kului algoritmin suorittamiseen. Suurimmaksi nopeudeksi 320×240 pikselin kuvilla havaittiin 120 fps. Esimerkiksi 640×480 pikselin kuvilla sovellus pystyi enää noin 40 fps nopeuteen. Tämän perusteella prototyypin nopeudeksi arvioitiin 10–20 MPix/s. Nopeus pätee 8-bittisille harmaasävykuville, joka on ainoa prototyypin tukema kuvamuoto.

7.1.4 Laskentajärjestyksen tehokkuuden arviointi

Prosessorin välimuistin, liukuhihnan ja muiden osien tehokas käyttö voi vaikuttaa merkittävästi ohjelman nopeuteen. Näitä arvioitiin Linuxin perf-työkalun avulla. Työkalu hyödyntää Intelin prosessoreissa olevia performance monitoring events [1, s. B-1] -laskureita. Nämä laskurit on integroitu suoraan prosessoriytimeen, joten niiden käyttö ei vaikuta ohjelman suoritukseen. Laskurien avulla saadaan tieto esimerkiksi siitä, montako ehdollista hyppyä ennustettiin väärin.

Tärkein tarkoitus näissä mittauksissa oli se, ettei toteutukseen jäisi mitään helposti korjattavaa pullonkaulaa. Jos esimerkiksi jossain kohdassa olisi havaittu suuri määrä välimuistista puuttuvia muistihakuja, olisi se voitu korjata lisäämällä sopiva käsky tiedon esihakemiseksi välimuistiin.

7.1.5 Mittausten toisto

Muiden sovellusten ja kuvadatan vaihtelun vaikutusten poistamiseksi jokainen testi ajettiin n. 200 kertaa, ja tuloksista otettiin mediaani. Keskiarvoa ei käytetty, sillä muiden ohjelmien vaatima aika voi hidastaa algoritmia hetkellisesti paljonkin. Tällöin keskiarvo vääristyisi ylöspäin.

Todellisen käytön kannalta on olennaista, kuinka nopeasti algoritmi toimii kuormitetulla koneella keskimäärin, toisin sanoen myös muiden ohjelmien vaikutus huomioiden. Ympäristöä on kuitenkin hyvin vaikea vakioida juuri tiettyyn kuormaan, joten se jätettiin pois näistä mittauksista.

7.2 Laitteisto

Varsinainen testi- ja käyttölaitteisto on pöytäkone, jossa on Windows XP käyttöjärjestelmänä ja Intel Pentium Dual-Core E5300 2,60 GHz suoritin. Lisäksi testit ajettiin samalla laitteistolla Ubuntu 9.10 -Linux-alustalla.

Liikkeen määrän, kuvan koon ja kääntäjäparametrien vaikutuksen sekä laskentajärjestyksen tehokkuuden arvioitiin käytetyt testit suoritettiin kannettavalla tietokoneella, jonka käyttöjärjestelmä on Debian GNU/Linux 5.0.4 ja suoritin Intel Core2 Duo T7300 2,00 GHz. Tarkemmat tiedot kummastakin laitteistosta ovat taulukossa 7.1.

Taulukko 7.1: *Mittauksissa käytetyt testilaitteistot.*

	Testilaite 1 (pöytäkone)	Testilaite 2 (kannettava)
Suoritin	Intel Pentium Dual-Core E5300	Intel Core2 Duo T7300
Kellotaajuus	2,60 GHz	2,00 GHz
Välimuisti	2048 KiB	4096 KiB
Keskusmuisti	3 GiB	2 GiB
Muistiväylän nopeus	800 MHz	800 MHz

7.3 Tulokset

Mittaustulokset on järjestetty aliluvuiksi tarkastellun ominaisuuden mukaan. Ensimmäisessä osassa verrataan algoritmin eri toteutuksia toisiinsa, ja lopuissa keskitytään tässä työssä toteutettuun versioon.

7.3.1 Toteutukset

Työssä käsitellystä algoritmista on olemassa yhteensä kolme toteutusta: aiempi prototyyppi, sekä tässä työssä tehdyt referenssitoteutus ja optimoitu toteutus. Näiden nopeudet yhdellä testikuvasarjalla on esitetty taulukossa 7.2.

Taulukko 7.2: *Algoritmin eri toteutusten nopeudet. Testissä on käytetty* 640×480 *pikselin kokoisia 8-bittisiä harmaasävykuvia, viittä kuvapuskuria ja* 3×3*-sumennusta konvoluutiokernelinä.*

	Aiempi prototyyppi	Referenssitoteutus	Optimoitu toteutus
Konvoluutio	–	54,4 ms	1,1 ms
Keskihajonta	–	31,2 ms	1,2 ms
Prosenttipiste	–	85,9 ms	0,1 ms
Yhteensä	–	171,5 ms	2,3 ms
	n.10–20 MPix/s	1,8 MPix/s	134 MPix/s

Taulukosta havaitaan, että referenssitoteutus on hitain ja optimoitu toteutus vastaavasti nopein. Erot eri toteutusten välillä ovat kymmenkertaisia, joten järjestys on selkeä vaikkei prototyypin nopeutta tunneta tarkasti.

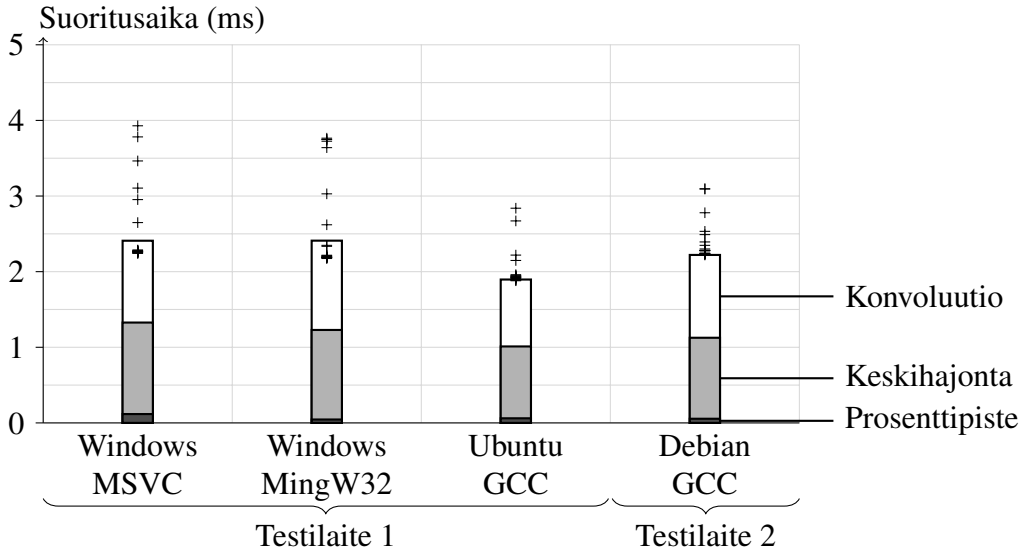
Prototyypin nopeus verrattuna referenssitoteutukseen selittyy sillä, että prototyypin käyttämässä ohjelmointiympäristössä on valmiina nopea konvoluutiototeutus. Lisäksi referenssitoteutuksessa on useita ehtolauseita eri kuvaformaatteja varten, mitä aiempi prototyyppi ei tue.

Selkein ero referenssitoteutuksen ja optimoidun toteutuksen suhteellisissa ajoissa on prosenttipisteen hakemiseen kuluvassa ajassa. Referenssitoteutus käyttää C:n `qsort`-funktiota taulukon lajitteluun, kun taas optimoidussa toteutuksessa prosenttipiste haetaan histogrammista. Histogrammitaulukon läpikäynti on hyvin nopeaa, sillä sen koko on pieni suhteessa kuvien pikselimäärään.

Osatulosten laskenta-aikoja tarkasteltaessa on kuitenkin otettava huomioon, että optimoidussa toteutuksessa konvoluution yhteydessä suoritetaan myös keskihajonnan summien päivitys, ja keskihajonnan laskennan yhteydessä kerätään samalla histogrammitaulukko.

7.3.2 Alustat ja kääntäjät

Nopeuden riippuvuutta käytetystä kääntäjästä, käyttöjärjestelmästä ja laitteistosta testattiin samalla testikuvasarjalla kuin yllä toteutusten yhteydessä. Tulokset on esitetty kuvaajassa 7.1 sekä taulukossa 7.3.



Kuva 7.1: *Algoritmista toteutetun optimoidun version suoritusaika kuvaa kohti eri alustoilla ajettuna. Kuvassa on esitetty eri osa-alueiden ajankäyttö. Risteillä on merkitty kokonaissuoritusaika 20 mittauksesta, mikä havainnollistaa tulosten vaihteluväliä.*

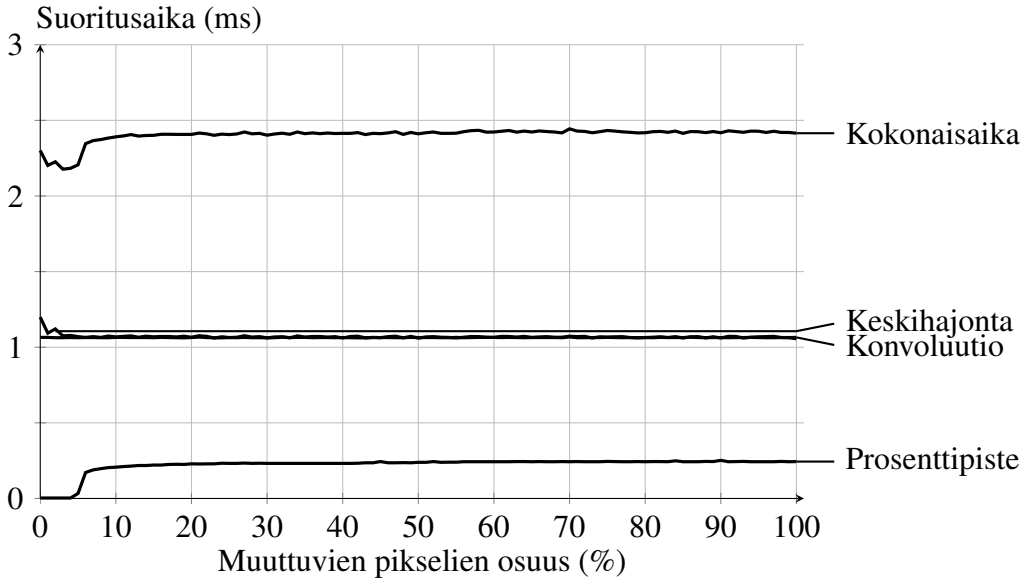
Taulukko 7.3: *Optimoidun toteutuksen nopeus eri kääntäjillä ja alustoilla käytettynä.*

Testilaite	Käyttöjärjestelmä	Kääntäjä	Laskenta-aika (ms)	Nopeus (MPix/s)
1	Windows XP	Microsoft Visual C++	2,3	134
1	Windows XP	MingW32 (GCC)	2,2	139
1	Ubuntu Linux	GCC	2,0	156
2	Debian GNU/Linux	GCC	2,3	135

Taulukon tulosten perusteella erot eri alustojen välillä ovat pieniä. Testilaitteen 1 tulokset Ubuntussa olivat hieman muita paremmat, mutta ero selittynee taustakuorman määrällä. Ubuntu oli käynnistetty CD-levyltä eikä siten sisältänyt mitään ylimääräisiä ohjelmia, kun taas Windows oli ollut jo aiemmin käytössä ja taustalla oli muitakin ohjelmia.

7.3.3 Liikkeen määrä

Algoritmin nopeuden riippumista kuvien sisällöstä arvioitiin eri määrän muutoksia sisältävillä testikuvasarjoilla. Käytetyssä kuvasarjassa oli kymmenen kuvaa, joissa oli pohjana sama valokuva. Joka toisessa kuvassa tietty prosenttimäärä kuvan pikseleistä oli muutettu mustaksi. Kuvassa 7.2 on esitetty suoritusajan riippuvuus muutosten suuruudesta.



Kuva 7.2: *Algoritmin osa-alueiden suoritusaika kuvaa kohti, kun kuvassa tapahtuvien muutosten suuruus vaihtelee. Testissä haettu prosenttipiste oli 95 %.*

Tulosten perusteella konvoluution suoritusaika ei riipu muutosten suuruudesta. Tämä oli odotettavissa, sillä konvoluutiossa tehdyt operaatiot eivät ole riippuvaisia kuvan sisällöstä.

Keskihajonnan ajankäytössä puolestaan on erikoinen 0.1 ms nousu pienimmillä muutosmäärillä. Tälle ei löytynyt mitään selkeää selitystä, mutta syy liittynee jotenkin prosessorin sisäiseen toimintaan juuri tässä tapauksessa. Ilmiö ei toistu, jos OpenMP-rinnakkaistus tai kääntäjän optimoinnit on kytketty pois.

Prosenttipisteen hakuun kuluva aika nousee selkeästi muutosten ylittäessä 5 %. Tätä pienemmillä arvoilla haku päättyy heti histogrammin alkuun, eli pieniä varianssin arvoja vastaavaan kohtaan. Kun yli 5 %:lla pikseleistä on suuri varianssin arvo, on myös 95 % prosenttipisteen arvo suuri ja haku päättyy vasta histogrammitaulukon loppuosassa.

Algoritmin kokonaisajassa vaihtelut ovat kuitenkin pieniä. Algoritmin suoritusaika ei siten riipu merkittävästi testikuvasarjan sisällöstä.

7.3.4 Kuvan koko

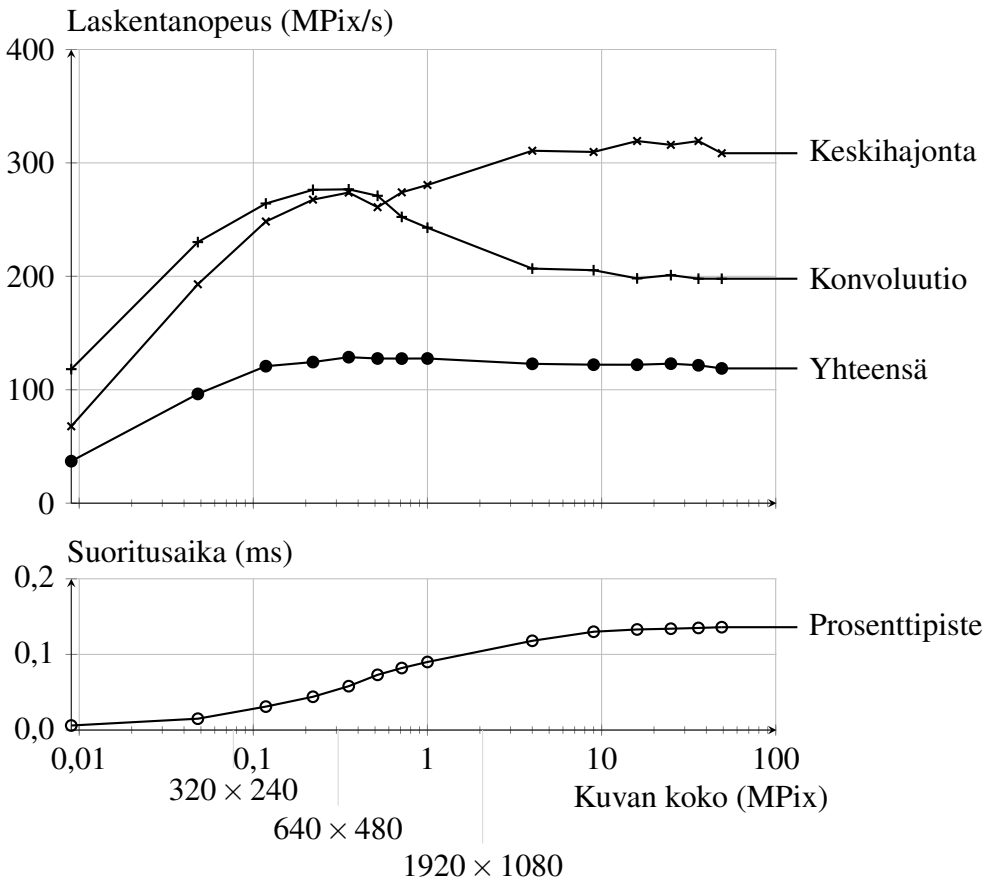
Algoritmin suoritusaika on oletettavasti lineaarisesti riippuvainen kuvan sisältämien pikselien määrästä. Sen vuoksi hyödyllisempää on tarkastella MPix/s-yksiköissä esitetyn nopeuden vaihtelua suhteessa kuvan kokoon. Tulokset on esitetty kuvassa 7.3.

Kuvaajan perusteella algoritmin kokonaissuoritusnopeus on yli 0,1 megapikselin kuvilla likimain vakio. Pienikokoisilla kuvilla suoritusnopeus laskee, mikä johtuu luultavasti lähinnä reunatarkistuksiin kuluvasta ajasta ja kiinteäaikaisista toimenpiteistä, kuten säikeiden käynnistyksestä. Osa-alueiden suhteelliset nopeudet kuitenkin vaihtelevat myös suurikokoisilla kuvilla.

Konvoluutio hidastuu kuvakoon ylittäessä 1 MPix. Tämä johtuu luultavasti ensimmäisen tason välimuistin täyttymisestä, sillä konvoluutio tarvitsee laskennassa nykyisen rivin lisäksi myös edellisiä ja seuraavia rivejä. Rivin pituuden kasvaessa kaikki tarvittavat rivit eivät mahdu yhtäaikaisesti välimuistiin.

Keskihajonta puolestaan nopeutuu suurikokoisilla kuvilla. Välimuistin täyttyminen ei vaikuta yhtä merkittävästi keskihajonnan laskentaan, koska summataulukoista tarvitaan aina kerrallaan vain yhtä riviä. Laskennassa on kuitenkin joitakin kiinteäaikaisia osuuksia: histogrammitaulukon alustus nollaksi vie noin 0,2 ms, ja lisäksi aikaa kuluu säikeiden käynnistykseen. Kuvakoon kasvaessa kiinteäaikaisten toimintojen osuudet per pikseli pienenevät, mistä aiheutuu havaittu nopeutuminen.

Prosenttipisteen hakuun kuluva aika nousee kuvakoon kasvaessa. Histogrammitaulukon koko on kuitenkin vakio, joten sen läpikäyntiin kuluvan ajan ei pitäisi muuttua. Todennäköisin selitys havaitulle muutokselle on välimuistin täyttyminen. Pienikokoisilla kuvilla histogrammitaulukko pysyy välimuistissa kokonaan, koska muuta tietoa on vähän. Suurilla kuvilla keskihajontaa laskettaessa harvinaisempia varianssin arvoja vastaavat



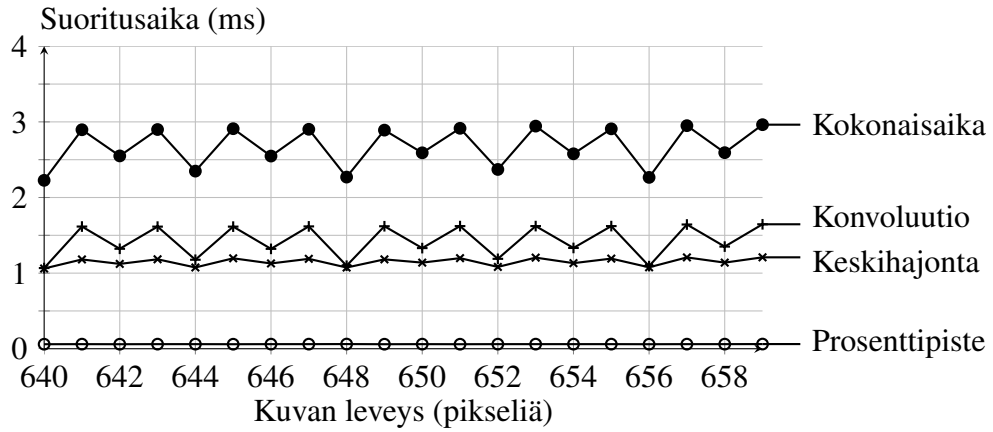
Kuva 7.3: *Algoritmin suoritusaika ja nopeus eri kokoisilla neliön muotoisilla harmaasävykuvilla. Sivun pituus on testipisteissä jaollinen 16:lla. Konvoluution ja keskihajonnan suoritusaika on likimain suoraan verrannollinen kuvan kokoon, minkä vuoksi niiden nopeus on esitetty MPix/s-yksikössä. Prosenttipisteen suoritusaika taas on likimain vakio.*

histogrammitaulukon arvot poistuvat välimuistista, jolloin prosenttipisteen haku kestää kauemmin.

7.3.5 Kuvan leveys

Algoritmi käsittelee kuvaa 16 tavun lohkoissa, joten jos kuvan leveys ei ole jaollinen 16:lla, joudutaan jokaisen rivin lopussa suorittamaan erikoistapaus tätä varten. Lisäksi prosessorin muistihaut ovat tehokkaimpia, kun muistiosoite on jaollinen 4:llä tavulla, eli 32 bitillä. Tämä aiheutuu dataväylän leveydestä. Kuvassa 7.4 on esitetty laskenta-aika suhteessa kuvan leveyteen.

Kuvaajan perusteella suurin vaikutus on leveyden parillisuudella. Sen sijaan ero 8:lla ja 16:lla jaollisten leveyksien välillä on pieni. Tämän perusteella suurin vaikutus on muistihakujen osoitteilla, johon verrattuna SSE2-lohkojen tasan meneminen ei merkitse



Kuva 7.4: *Algoritmin suoritusajan riippuvuus kuvan leveydestä. Testi on suoritettu harmaasävykuvilla, eli kuvan leveys on samalla yhden rivin koko tavuissa.*

paljoa. Tämä on ymmärrettävää, sillä muistihaut vaikuttavat jokaisen pikselin käsittelyyn, kun taas lohkojen tasan meneminen vaikuttaa vain rivin lopussa.

Konvoluution yhteydessä suoritetaan aina muistihakuja myös parittomista osoitteista. Sen vuoksi jatko-optimoinnissa voitaisiin arvioida, nopeutuisiko laskenta lataamalla lohkot aina tasan jaollisista osoitteista. Tällöin lataamisen jälkeen lohkoa täytyisi muokata vielä bittisiirroilla.

7.3.6 Kuvatyypit

Taulukossa 7.4 on esitetty algoritmin nopeus eri kuvatyypeillä. Keskihajonnan ja prosenttipisteen suoritusaika riippuu siitä, kuinka monelle värikanaville sovellusohjelma haluaa keskihajonnan laskea. Konvoluutio taas suoritetaan aina kaikille värikanaville.

Taulukko 7.4: *Algoritmin nopeus eri kuvatyypeillä. Testikuvan koko oli* 640×480.

Kuvatyyppi	Konvoluutio (ms)	Keskihajonta (ms)	Prosenttipiste (ms)	Yhteensä (ms)
Harmaasävy	1,1	1,2	0,05	2,4
RGB24	4,9	$3 \cdot 2{,}2$	$3 \cdot 0{,}05$	11,7
(A)RGB32	5,3	$4 \cdot 2{,}2$	$4 \cdot 0{,}03$	14,2

Taulukon perusteella suoritusaika kuusinkertaistuu RGB32-kuvilla harmaasävykuviin verrattuna. Pelkästään kuvan koosta odotettavissa oleva muutos olisi nelinkertaistuminen, joten algoritmi hidastuu yllättävän paljon. Keskihajonnan osalta hidastuminen johtuu luultavasti siitä, että keskihajonta lasketaan erikseen jokaiselle värikanavalle. Tällöin summataulukot täytyy käydä läpi pätkittäin, koska eri värikanavien pikseliarvot ovat limittäin.

7.3.7 Kääntäjäparametrit

Eri kääntäjäoptimointien vaikutus algoritmin nopeuteen on esitetty taulukossa 7.5. Testit on suoritettu GCC:llä, ja muissa testeissä käytetyt käännösparametrit ovat taulukon viimeinen eli nopein vaihtoehto.

Taulukko 7.5: *Algoritmin nopeus GCC:n eri käännösparametreilla. Kaikissa kohdissa on lisäksi mukana* `-march=pentium4` *SSE2-käskyjen käyttöönsaamiseksi.*

Kääntäjän parametrit	Konvoluutio (ms)	Keskihajonta (ms)	Prosenttipiste (ms)	Yhteensä (ms)
`-O0`	22,3	13,5	0,19	36,4
`-O1`	2,3	3,2	0,11	6,1
`-O2`	2,2	2,6	0,10	5,0
`-O3`	2,2	2,2	0,06	4,5
`-O3 -fopenmp`	1,1	1,2	0,05	2,4

Kääntäjän optimoinneilla on selvä vaikutus algoritmin nopeuteen. Eniten nopeutuu konvoluutio, mikä onkin ohjelmakoodin kannalta monimutkaisin osuus. Merkittävä osa nopeutuksesta `-O0` ja `-O1` -tasojen välillä johtuu luultavasti `inline`-funktioista, joita koodissa on käytetty sen selkeyttämiseksi. Kun optimoinnit on kytketty pois `-O0`:lla, kuluu huomattavasti aikaa näiden apufunktioiden kutsumiseen.

Rinnakkaistamisen kytkeminen päälle `-fopenmp`:llä lähes kaksinkertaistaa nopeuden. Algoritmi siis rinnakkaistuu tehokkaasti, ja hyödyntää molemmat prosessoriytimet.

7.3.8 Laskentajärjestys

Taulukossa 7.6 on esitetty mittaustulokset, joiden perusteella arvioitiin algoritmin tehokkuutta välimuistin ja hyppyjen ennustuksen kannalta. Tuloksissa ei havaittu merkittäviä ongelmakohtia, joten muutoksia mittaustulosten perusteella ei tehty.

Viimeisen tason välimuistista puuttuvat muistihaut aiheuttavat sen, että prosessori joutuu odottamaan tiedon hakemista ulkoisesta muistista. Tämä on hidasta verrattuna prosessorin normaaliin suoritusnopeuteen. Testatulla prosessorilla, jossa on 800 MHz muistiväylä ja 2 GHz kellotaajuus, ulkoisen muistin käsittely vie noin 70 prosessorin kellojaksoa [1, s. 2-59]. Lisäksi viimeisen tason välimuistin käsittely vie 14 kellojaksoa.

Taulukon muistihakuihin kuluva aika on siis $(935\ 000 \cdot 14 + 9\ 180 \cdot 70)/(2\ \text{GHz}) \approx 6{,}9$ ms. Tämä on melko suuri osuus 24 ms kokonaissuoritusajasta, mutta muistihakujen viivettä on mahdoton poistaa täysin. Siksi muistihakujen jatko-optimoinnilla ei pystyttäisi saavuttamaan kovin merkittävää nopeutusta suhteessa työmäärään.

Korkean tason ohjelmointikielten ehtorakenteet, kuten `if`-lauseet, toteutetaan ehdollisilla hyppykäskyillä. Prosessorit pyrkivät ennustamaan nämä hypyt edellisten

Taulukko 7.6: *Prosessorin tapahtumalaskureiden avulla mitatut tiedot ohjelman suorituksesta. Testissä käytettiin kymmentä* 640×480 *harmaasävykuvaa, eli suoritusaika oli 24 ms.*

Tapahtuma	Lukumäärä
Kellojaksot	101 500 000
Muistihaut yhteensä	63 850 000
Muistihaut L2-välimuistista	935 000
Muistihaut, jotka puuttuvat L2-välimuistista	9 180
Ehdolliset hypyt	17 300 000
Väärin ennustetut hypyt	103 680

suorituskertojen perusteella, jotta prosessorin liukuhihna toimisi tehokkaasti. Testatulla prosessorilla liukuhihna on 18 käskyn pituinen [1, s. 2-7], joten väärin ennustettuun hyppyyn kuluva aika on enimmillään 18 kellojakson suuruusluokkaa [1, s. 2-39].

Taulukon hyppyihin kuluva aika on $(103\ 680 \cdot 18)/(2\ \text{GHz}) \approx 0{,}9$ ms. Tämä aika on lähes olematon suhteessa kokonaissuoritusaikaan, joten hyppyjen optimoiminen ei ole tässä tapauksessa hyödyllistä.

7.4 Tulosten vertailua

Algoritmin nopeutta voi arvioida vertaamalla sitä esimerkiksi Intel Integrated Performance Primitives -kirjaston nopeuteen. Intelin kirjastossa on toteutukset keskihajonnan ja konvoluution laskentaan, joita voitaisiin verrata vastaaviin osuuksiin tässä työssä. Vertailun suorittaminen vaatisi kuitenkin sopivan testiohjelman rakentamisen, joten sitä ei tämän työn puitteissa tehty.

NVidian CUDA on näytönohjainlaskentaan tarkoitettu kirjasto. NVidia on julkaissut esimerkin sen avulla toteutetusta konvoluutiosta [18]. Siinä RGB-kuville saavutettu nopeus 3×3-kernelillä oli noin 2 000 MPix/s. Tässä työssä tehdyn konvoluutioalgoritmin nopeus RGB-kuvilla on 63 MPix/s, mutta lukuun sisältyy myös varianssin summapuskurien päivitys. Joka tapauksessa näytönohjainlaskennalla pystyttäisiin selvästi nopeuttamaan algoritmia, ainoana haittapuolena huono yhteensopivuus eri laitteistojen kanssa.

8. YHTEENVETO

Toteutuksessa saavutettiin 134 MPix/s nopeus, mikä ylittää huomattavasti asetetun 40 MPix/s tavoitteen. Tärkeimmät tiedot projektista on esitetty taulukossa 8.1.

Taulukko 8.1: *Toteutetun projektin perustiedot.*

Toteutuskieli	C
Tarkoitus	Muutosten tunnistaminen videokuvasta.
Käytetyt optimointimenetelmät	SSE2 ja OpenMP
Työmäärä	
kirjaston toteutus	70 tuntia
työn kirjoittaminen	80 tuntia
Koodirivien määrä	
varsinainen toteutus	1 685, josta ohjelmakoodia 1 091 riviä
referenssitoteutus	538, josta ohjelmakoodia 414 riviä
testiohjelmat	996, josta ohjelmakoodia 790 riviä
Vahvuudet	Nopea toteutus vähäisillä riippuvuuksilla, toteuttaa tarkasti entisen rajapinnan.
Heikkoudet	Monimutkainen ohjelmakoodi, ei hyödynnä näytönohjainlaskentaa.

Toteutetun kirjaston rivimäärä on 1 685, mikä on suhteellisen vähän. Referenssitoteutuksen rivimäärä on 538, joten optimointi vain kolminkertaisti sen, ja lisäksi suuri osa optimoidun version riveistä on kommentteja. Varsinaisen koodin määrä hieman yli kaksinkertaistui. Pienestä rivimäärästä huolimatta erityisesti SSE2-käskykanta tekee toteutuksesta hieman vaikeaselkoisen.

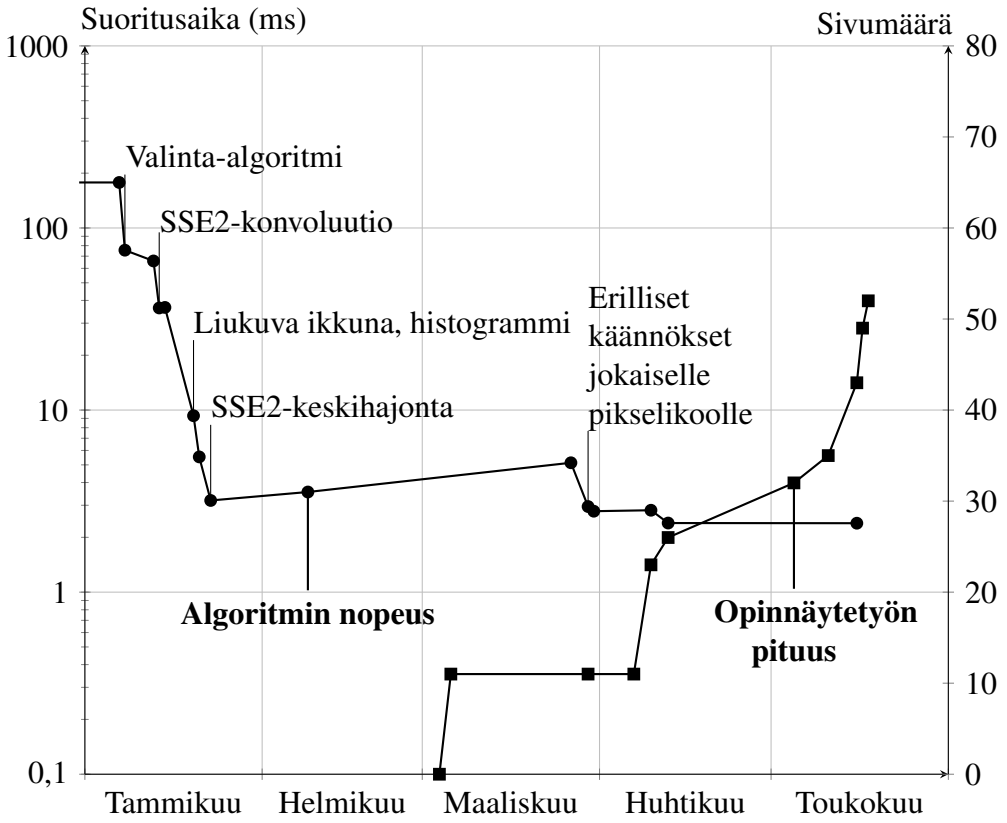
Suurin nopeusetu saavutettiin SSE2-käskykannan käytöstä ja OpenMP-rinnakkaislaskennasta. Näistä OpenMP:n käyttöönotto oli erityisen helppoa, kun taas SSE2-käskykanta vaikutti laajasti koko toteutuksen rakenteeseen esimerkiksi kokonaislukulaskennan osalta. Käytetyt nopeutustekniikat ja niiden arvioidut hyödyt ja työmäärät on listattu taulukossa 8.2.

Taulukossa on mainittu myös kirjallisessa esityksessä vähemmän huomiota saanut ohjelmakoodin rakenteellinen parantelu. Tällä tarkoitetaan esimerkiksi turhien ehtolauseiden ja laskutoimitusten karsintaa.

Kuvassa 8.1 on esitetty projektin eteneminen algoritmin nopeuden ja kandidaatintyön sivumäärän perusteella. Varsinainen optimointi ja menetelmien valinta valmistui jo tammikuussa, mutta myöhemmin aikaa kului vielä toteutuksen testaukseen ja hienosäätöön.

Taulukko 8.2: *Eri nopeustekniikoiden vertailu. Nopeusedut on arvioitu sen pohjalta, kuinka paljon suurempi lopullisen toteutuksen kokonaisaika olisi, jos yksittäinen tekniikka olisi jätetty käyttämättä.*

Tekniikka	Nopeusetu	Työmäärä	Oletukset
OpenMP	2×	2 h	2 prosessoriydintä
SSE2-käskykannan käyttö	5×	20 h	
Rakenteellinen parantelu	2×	5 h	
Erilliset käännökset	2×	2 h	
Konvoluution separointi	1.5×	10 h	5×5 kerneli
Keskihajonnan liukuva ikkuna	2×	10 h	10 kuvapuskuria
Prosenttipisteen histogrammi	40×	5 h	Vertailu C:n `qsort`:iin



Kuva 8.1: *Projektin eteneminen algoritmin nopeuden ja kandidaatintyön sivumäärän perusteella esitettynä. Kuvaan on merkitty joidenkin optimointitekniikoiden toteutusajankohdat.*

Projekti eteni vaihtelevasti, mutta valmistui kuitenkin melko lyhyessä ajassa. Vaikeimpia osuuksia olivat oikean toiminnan varmistaminen, separoituvan konvoluution toteutus sekä toimintaperiaatteiden selkeä esittäminen opinnäytetyössä.

Kokonaisuutena projekti oli onnistunut. Toteutettu kirjasto on toistaiseksi riittävän nopea, ja lisäksi tässä työssä on esitetty hyvä pohja mahdolliselle näytönohjainlaskennan toteutukselle tulevaisuudessa.

LÄHTEET

[1] Intel Corporation, *Intel®64 and IA-32 Architectures Optimization Reference Manual*, 2009.
http://www.intel.com/Assets/PDF/manual/248966.pdf.

[2] Intel Corporation, *Intel®64 and IA-32 Architectures Software Developer's Manual, Volume 1*, 2009.
http://www.intel.com/assets/PDF/manual/253665.pdf.

[3] AMD, *AMD Athlon™64 Processor Product Data Sheet*, 2006.
http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24659.pdf.

[4] AMD, *AMD Opteron™Processor Product Data Sheet*, 2007.
http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/23932.pdf.

[5] OpenMP Architecture Review Board, *OpenMP Application Program Interface*, 2008.
http://www.openmp.org/mp-documents/spec30.pdf.

[6] Free Software Foundation, GCC team, *GNU libgomp*.
http://gcc.gnu.org/onlinedocs/libgomp/, haettu 13.9.2010.

[7] Microsoft Corporation, *OpenMP in Visual C++*.
http://msdn.microsoft.com/en-us/library/tt15eb9t%28VS.80%29.aspx, haettu 13.9.2010.

[8] Khronos Group, *OpenCL – The open standard for parallel programming of heterogeneous systems*.
http://www.khronos.org/opencl/, haettu 13.9.2010.

[9] SiSoftware, *OpenCL CPU Performance (OpenCL vs native/Java/.Net)*.
http://www.sisoftware.co.uk/?d=qa&f=cpu_opencl, haettu 10.3.2010.

[10] S. Eddins, *Separable convolution: Part 2*. The MathWorks, Inc., 2006.
http://blogs.mathworks.com/steve/2006/11/28/separable-convolution-part-2/, haettu 7.4.2010.

[11] A. Perttula, K. Vattulainen ja T. Suurhasko, *Opintomoniste kurssille MAT-20500 Todennäköisyyslaskenta*, 9/2009.
http://www.math.tut.fi/courses/MAT-20500/tnlmoniste.pdf.

[12] F. J. Anscombe, “Topics in the investigation of linear relations fitted by the method of least squares,” *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 29, no. 1, p. 3, 1967.

[13] R. Bhatia and C. Davis, “A better bound on the variance,” *The American Mathematical Monthly*, vol. 107, no. 4, pp. 353–357, 2000.

[14] M. Blum, R. Floyd, and V. Pratt, “Time bounds for selection,” *Journal of Computer and System Sciences*, vol. 7, no. 4, pp. 448–461, 1973.

[15] I. Mornar, “Performance Counters for Linux, v8,” 2009. Linux-kernel mailing list. `http://lwn.net/Articles/336542/`.

[16] “Performance Counters for Linux Wiki.” `https://perf.wiki.kernel.org/`, haettu 14.9.2010.

[17] A. Lee, “Beware of QueryPerformanceCounter(),” 2006. Virtualdub blog. `http://www.virtualdub.org/blog/pivot/entry.php?id=106`.

[18] V. Podlozhnyuk, *Image Convolution with CUDA*. NVidia, 2007.