Tommi Teistelä

Yleinen laskenta grafiikkasuorittimilla

Tietotekniikan
kandidaatintutkielma
25. tammikuuta 2012

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

Tekijä: Tommi Teistelä
Yhteystiedot: tommi.t.teistela@jyu.fi
Työn nimi: Yleinen laskenta grafiikkasuorittimilla
Title in English: General-Purpose Computing on Graphics Processors
Työ: Tietotekniikan kandidaatintutkielma
Sivumäärä: 33
Tiivistelmä: Esitellään nykyaikaisten grafiikkasuorittimien rakennetta, toimintaperiaatteita ja tutkitaan OpenCL:ää keinona käyttää niiden laskentakykyä yleisempään laskentaan. Toteutetaan osa JPEG-kuvanpakkausalgoritmia grafiikkasuorittimella OpenCL:n avulla.
English abstract: The structure and operating principles of modern graphics processing units are presented. OpenCL is examined as means to use their computing capability for general computing tasks. A subset of the JPEG image compression algorithm is implemented on OpenCL.
Avainsanat: grafiikkasuoritin, GPGPU, OpenCL, JPEG
Keywords: graphics processor, GPGPU, OpenCL, JPEG

Sisältö

1 Johdanto 1

2 Grafiikkasuorittimista 2
- 2.1 Taustaa 2
- 2.2 Toimintaperiaatteet 3
- 2.3 Laskentakerneli 3
- 2.4 Muistiarkkitehtuuri 4
- 2.5 Yhteenveto laitteiston ominaisuuksista 4

3 OpenCL 5
- 3.1 Suorituksen rakenne 5
- 3.2 Tiedon esittäminen 6
- 3.3 OpenCL C -kieli 7
- 3.4 OpenCL:n käyttö C-ohjelmassa 8
- 3.5 OpenCL 1.1 8

4 Muita kehitysvälineitä 9
- 4.1 CUDA 9
- 4.2 Grafiikkarajapinnat 10

5 Grafiikkasuoritin kuvanpakkauksessa 12
- 5.1 JPEG-algoritmi 12
- 5.2 Laskentatehtävän sovitus 13
- 5.3 Tuloksia 15

6 Yhteenveto 17

Lähteet 18

Liitteet

1 Lähdekoodia 20
- 1.1 OpenCL C -kerneli 20
- 1.2 C-pääohjelma 22

1 Johdanto

Graafisten työpöytien ja tietokoneviihteen yleistyminen on tuonut tavallisiin työasemiin huomattavaa rinnakkaislaskentakykyä erilaisten grafiikkasuorittimien muodossa. Viime aikoina laitteiden suorituskyky on herättänyt kiinnostusta niiden käyttöön grafiikan lisäksi muussakin laskennassa, ja uudemmat grafiikkasuorittimet sisältävätkin erityisesti yleisempään laskentaan tarkoitettuja ominaisuuksia.

Tässä kandidaatintutkielmassa on tutkittu nykyaikaisten grafiikkasuorittimien (engl. GPU, *Graphics Processing Unit*) ominaisuuksia ja keinoja soveltaa niiden laskentatehoa grafiikan piirron lisäksi muuhun laskentaan. Tavoitteena on antaa hyvä kuva tyypillisistä grafiikkasuorittimista, niiden käytössä tarvittavista ohjelmointirajapinnoista sekä molempien vahvuuksista ja heikkouksista erilaisissa laskentatehtävissä. Tarkemmin tutustutaan yleistymässä olevaan OpenCL-rajapintaan ja toteutetaan sen avulla osittain grafiikkasuorittimella toimiva JPEG-algoritmi.

Eri laitteilla ja rajapinnoilla on käytössä samoista käsitteistä useita eri termejä. Tässä on johdonmukaisuuden vuoksi suosittu OpenCL:n termistöä.

2 Grafiikkasuorittimista

2.1 Taustaa

Grafiikkaoperatioiden kiihdytykseen erikoistuneet laitteet eivät ole mitenkään uusi käsite tietokonemaailmassa. Useimmissa vanhemmissa laitteissa grafiikan tuottaminen oli pitkälti toteutettu laitteistolla yleiskäyttöisempien suorittimien hitauden takia. Alunperin täysin merkkipohjaisten IBM PC-työasemien alkeelliset kuva- ja ääniominaisuudet lienevät enemmänkin poikkeus sääntöön.

Nopeammat laajennusväylät, kuten VESA Local Bus ja myöhempi PCI, sallivat kuitenkin PC-järjestelmien grafiikkatoimintojen laajentamisen: kaksiulotteista piirtoa nopeuttavat näytönohjaimet olivat jo yleisiä 1990-luvun alkupuolella. Aiemmin pelihallilaitteissa sekä pelikonsoleissa käytetyt 3D-grafiikkapiirit saivat jalansijaa myös PC-käytössä vuonna 1996 julkaistun 3dfx Voodoo-laajennuskortin myötä. Useat muut näytönohjainvalmistajat julkaisivat lähivuosina omia grafiikkakiihdyttimiään.

Näytönohjainvalmistajien kilpailu tuotti yhä nopeampia ja monimutkaisempia laitteita, joiden myötä kasvoi myös niiden ohjelmointiin tarvittava työmäärä. Rajapintoja laajennettiin uusilla kutsuilla ja tiloilla, kunnes niiden määrä pakotti toteuttamaan osan niistä ohjelmoitavilla suorittimilla erikoistuneiden piirien sijaan. Nvidian GeForce 3-näytönohjain oli ensimmäinen, jonka sisäinen ohjelmoitavuus oli käytettävissä laitteen ulkopuolisen rajapinnan kautta. GeForce 3:n ajurille oli mahdollista syöttää lyhyitä, sen omalla assembly-kielellä kirjoitettuja ohjelmia, jotka korvasivat laitteen tavallisen toiminnan koordinaattimuunnoksissa ja pikselien väriarvojen laskemisessa. Ohjelmoitavuuden myötä myös yleistyi termi GPU (*Graphics Processing Unit*) kuvaamaan tällaista grafiikkasuoritinta [7].

Kun muutkin laitevalmistajat ryhtyivät tarjoamaan ohjelmoitavuutta, standardeja tarvittiin, ja pohja niille löydettiin pitkään ei-tosiaikaisessa 3D-grafiikassa käytetyistä *varjostinohjelmista* (esim. RenderMan-arkkitehtuuri ja sen Shader-ohjelmat).

Microsoftin Direct3D-rajapinnan HLSL (High Level Shader Language) ja Khronos Groupin OpenGL:n GLSL, sekä laitevalmistaja Nvidian Cg-kieli ovat niiden pohjalta kehitettyjä C-kieltä muistuttavia "varjostinkieliä" (engl. *shading language*). Uudemmat laitteistosukupolvet ovat myöhemmin poistaneet varjostinohjelmien rajoituksia koodin pituudessa, tuetuissa tietotyypeissä ja kerralla käsiteltävän tietomäärän koossa.

2.2 Toimintaperiaatteet

Reaaliaikaisessa grafiikassa käsitellään tyypillisesti suuria määriä kolmen tai neljän (liuku)luvun vektoreita. Samaa lyhyttä ohjelmaa käytetään hyvin monen tuloksen laskemiseen, eivätkä tulokset ole riippuvaisia toisistaan.

Grafiikkasuorittimien arkkitehtuuri erikoistuu tällaiseen laskentatehtävään sisältämällä suuren määrän hyvin yksinkertaisia suorittimia (engl. *processing element*, *ALU*), jotka on ryhmitetty *laskentayksiköihin* (engl. *compute unit*). Saman yksikön sisällä olevat suorittimet suorittavat jokaisen saman ohjelman käskyn yhtäaikaisesti: jos ohjelma sisältää ehtorakenteita ja yksikin suoritin joutuu suorittamaan ehdollisen osan koodia, koko yksikkö joutuu odottamaan sen suorittamista. Käskyjen suoritus estetään maskilla ytimissä, joissa suorituksen ehto ei täyty [5], ja ohjelman eri haarat suoritetaan peräkkäin. Yksittäistä laskentayksikköä voisi tämän perusteella mallintaa leveänä SIMD-aritmetiikkaytimenä [7][5], joka kykenee suorittamaan monimutkaisempia ohjelmia, uhraten tehokkuutta tarvittaessa.

Tällaisesta lyhyiden ohjelmien laajasta rinnakkaissuorituksesta eri syötteillä on myös käytetty lyhennettä SPMD (Single Program, Multiple Data) [6]. SPMD kuitenkin vihjaa, että ohjelmien suorituspolku voi olla erilainen eri ytimillä, mutta grafiikkasuorittimessa voi olla erilaisia suorituspolkuja työn alla vain yksi jokaista laskentayksikköä kohden.

Grafiikkasuorittimella on laskentayksiköiden omien rekistereiden lisäksi käytössään omaa muistia ja sisäinen muistiväylä sen käyttämiseen. Väylä eroaa tavanomaisten suorittimien muistiväylistä leveydellään (tehokäyttöön tarkoitettujen näytönohjainten muistiväylät ovat yleensä 256 tai 384 bittiä leveitä, x86-arkkitehtuurin kanssa käytetyt 64 bittiä) ja korkealla viiveellään. Väylän jatkuva siirtonopeus on kuitenkin moninkertainen x86-arkkitehtuurissa käytettyihin muistiväyliin verrattuna. Saman valmistajan ja laitesukupolven grafiikkasuorittimen eri mallit eroavatkin lähinnä muistiväylän leveydessä ja suorittimien lukumäärässä.

2.3 Laskentakerneli

Grafiikkasuorittimella rinnakkain suoritettavia lyhyitä ohjelmia kutsutaan yleisesti *kernel*-ohjelmiksi. Kernel-ohjelmaa voidaan pitää laskentaa suorittavan silmukan sisimpänä osana. Grafiikkarajapinnoissa kernel-ohjelmina toimivat eri piirtovaiheiden varjostinohjelmat.

2.4 Muistiarkkitehtuuri

Koko laitteelle yhteisen muistin lisäksi jokaisessa laskentayksikössä on omaa muistia, joka on jaettu sen ytimien kesken. Paikallinen muisti toimii grafiikkakäytössä ylimääräisenä välimuistina ja parametrien välitystyökaluna, mutta sopivaa laskentarajapintaa käyttäessä siihen voidaan viitata suoraan.

Yleensä muistiarkkitehtuuriin kuuluu myös kaksitasoinen välimuisti. Sen ensimmäisen tason L1-välimuisti on jaettu laskentayksikön suorittimien kesken, ja toisen tason L2-välimuisti on yhteinen koko grafiikkasuorittimelle.

Käskyjen samanaikaisuudesta laskentayksikön sisällä seuraa myös muistiin lukemisen ja kirjoittamisen samanaikaisuus: kun suoritettava ohjelma saavuttaa laitteen päämuistiin viittaavan luku- tai kirjoituskäskyn, muistiohjain saa laskentayksiköltä useita muistiosoitteita. Leveästä muistiväylästä on etua tässä, sillä jos laskentakernelin "lähekkäiset" suorituskerrat käsittelevät lähekkäin sijaitsevia muistialkioita, muistiohjain pystyy yhdistämään useita kirjoitus- ja lukuoperaatioita.

Muistiarkkitehtuuri siis olettaa vahvasti, että käsiteltävän tiedon paikallisuusaste on korkea, eikä ole läheskään riittävän nopea ohjelmaan, jossa jokainen suoritin tuottaa erillisen muistitoimenpiteen. Volkov ja Demmel [5] havaitsivat pitkän muistikopioinnin suoritusajan kasvavan lähes 80-kertaiseksi, kun kopioitavan tiedon (32-bittisiä lukuja) alkiot levitettiin 512 alkiokoon etäisyydelle toisistaan. P. Buin ja J. Brockmanin "Performance Analysis of Accelerated Image Registration using GPGPU":ssa saatiin myös huomattavaa parannusta nopeuteen muokkaamalla laskentaohjelman muistinkäyttöä [6].

2.5 Yhteenveto laitteiston ominaisuuksista

Tässä luvussa esiteltyjen ominaisuuksien pohjalta voidaan hahmotella seuraavat edellytykset tehokkaalle laskennalle grafiikkasuorittimella:

- Suorituspolkujen oltava mahdollisimman yhdenmukaisia (laskentayksikön sisällä) [7].
- Yhteisen muistin kirjoitus tai lukeminen ei saa hajaantua liikaa "vierekkäisten" suoritustehtävien välillä [5].
- Rinnakkaisia laskentatehtäviä voidaan tuottaa riittävästi laitteen kaikkien ytimien kuormittamiseen.

3 OpenCL

OpenCL on tunnetun OpenGL-rajapinnan määritelmiä ylläpitävän Khronos Groupin rajapintastandardi yleiseen laskentaan [2], joka on suunniteltu myös grafiikkasuorittimien vahvuuksia ja rajoituksia huomioiden.

OpenCL vastaa kutsu- ja nimeämiskäytännöiltään hyvin OpenGL:ää, ja sisältää OpenGL:n kaltaisen laajennusmekanismin. Laajennusmekanismia käyttäen ohjelma voi tiedustella rajapinnan toteutukselta lisäominaisuuksia, joita perusstandardi ei määrittele. Huomattava, lähes joka toteutuksessa saatavilla oleva laajennus, on kyky jakaa resursseja OpenCL- ja OpenGL-kontekstien välillä. Tiedon jako voi olla hyödyllistä tulosten visualisoinnissa tai OpenCL-laskennan käytössä osana OpenGL:ää käyttävää sovellusta.

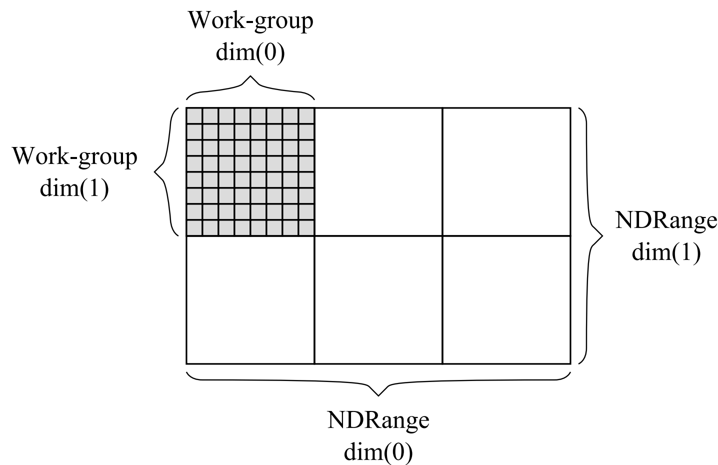
OpenCL:lle on olemassa myös virallinen C++-toteutus, joka käärii C-kielelle suunnatun rajapinnan eri objektityypit ja niihin liittyvät kutsut C++:n luokiksi ja metodeiksi. Käärintä helpottaa hieman virheiden havaitsemista ja C++:n oletusarvoiset parametrit lyhentävät omaa ohjelmakoodia.

3.1 Suorituksen rakenne

Rinnakkaissuorituksen perusta OpenCL:ssä on yksi-, kaksi- tai kolmiulotteinen alue (*NDRange*) indeksejä, jotka toimivat tunnisteina yksittäisille suoritustehtäville (*work-item*). Toisin kuin grafiikkarajapinnoissa, tietyn suoritustehtävän kirjoittama muistialue ei ole etukäteen määrätty. Säännöllinen muistin käyttö on silti toivottavaa laitteiston aiemmin esitellyn muistiarkkitehtuurin kannalta.

Indeksit jaetaan vielä pienempiin väleihin, paikallisiin työryhmiin (*work-group*). Jokaisella työryhmällä on käytössään omaa muistia, joka vastaa laitteen suorittimien paikallista muistia ja rekistereitä. Paikallinen muisti ei tämän takia kärsi juuri muistiväylän viiveestä ja soveltuu hyvin välituloksien jakamiseen työryhmän kesken. Koska laskentaohjelman sisällä muistia voidaan varata vain staattisesti, OpenCL-toteutus voi tarvittaessa varata liian suuren paikallisen muistialueen laitteen yleisestä muistista.

Jos tehtävän tarkalla osituksella ja paikallisen muistin käytöllä ei ole merkitystä, paikallisten työryhmien jakaminen voidaan jättää OpenCL-toteutuksen tehtäväksi.



Kuva 1: OpenCL:n suoritusrakenne

3.2 Tiedon esittäminen

OpenCL:ssä on kaksi perustapaa esittää laskentaohjelmassa luettavaa ja kirjoitettavaa tietoa: tavalliset puskuriobjektit (*buffer*) ja kuvaobjektit (*image*). Molempia voidaan lukea ja kirjoittaa OpenCL:ää käyttävästä ohjelmasta joko kuvaamalla (*map*) objektin sisältö laskentaa ohjaavan ohjelman muistiavaruuteen tai lisäämällä CL:n suoritusjonoon objektin luku- tai kirjoitustehtävä.

Puskuri muistuttaa käyttötavoiltaan ja rajoituksiltaan muistiosoitinta C-ohjelmassa ja voi sisältää vapaamuotoista tietoa: CL-ohjelma voi lukea ja tulkita puskuriobjektin sisältöä vapaasti.

Kuvaobjekti sisältää kaksi- tai kolmiulotteisen taulukon vektoreita, joiden sallitut tietotyypit riippuvat laskentalaitteesta. Jokainen standardin mukainen OpenCL:n toteutus tukee kuitenkin vähintään neljän komponentin vektoreita, joiden yksittäisten komponenttien tietotyyppinä on jokin OpenCL:n perustietotyypeistä [2]. Kuvaobjektista lukeminen ja siihen kirjoittaminen on laskentaohjelmassa mahdollista vain erityisesti kuville tarkoitetuilla OpenCL C:n funktioilla. Kernelifunktio ei myöskään saa lukea ja kirjoittaa samaa kuvaobjektia yhtäaikaisesti, ja kolmiulotteiseen objektiin kirjoittaminen edellyttää toteutukselta *cl_khr_3d_image_writes*-laajennusta.

Rajoituksista huolimatta kuvaobjekteista voi olla hyötyä. Kuvaobjekteja käsitellessä laite voi hyödyntää niihin erikoistuvaa välimuistiaan ja on usein nopeampi, erityisesti jos lähekkäiset kernelifunktion suoritukset käsittelevät lähekkäisiä kaksi- tai kolmiulotteisen tiedon alkioita.

3.3 OpenCL C -kieli

OpenCL-standardi määrittelee kutsurajapinnan lisäksi sen kanssa käytettävän ohjelmointikielen, OpenCL C:n, joka perustuu C99-standardin mukaiseen C-kieleen [2] ja laajentaa sitä mm. vektorityypeillä sekä osoiteavaruusmääreillä (engl. *address space qualifier*), joilla ilmaistaan objektin sisältävä muistin osa.

Kielessä on kohdelaitteesta johtuen joitakin rajoituksia tavalliseen C-kieleen verrattuna. Rekursiiviset funktiokutsut ja osoittimet funktioihin eivät ole OpenCL C:ssä mahdollisia, sillä kääntäjä pyrkii purkamaan funktiokutsut pois ohjelmasta. C:n vakiokirjasto ei ole käytettävissä, mutta standardi määrittelee pitkälti vastaavan joukon matemaattisia funktioita, sekä joitakin grafiikkasuorittimille tyypillisiä kaksi- ja kolmiulotteisiin kuvaobjekteihin erikoistuvia funktioita.

Tehokkuussyistä kieli ei automaattisesti takaa muistioperaatioiden suoritusjärjestystä. Suoritustehtäviä voidaan tarvittaessa synkronoida erityisillä *fence*- ja *barrier*-käskyillä.

OpenCL C:llä kirjoitettuja ohjelmia voidaan syöttää CL-toteutukselle käännettäväksi ja kohdelaitteelle suoritettavaksi rajapinnan kutsuilla. OpenCL:n toteuttava ajuri voi pitää omaa välimuistiaan ohjelmista, mutta tarvittaessa ohjelman (laitekohtainen) käännetty versio voidaan noutaa *clGetProgramInfo*-kutsulla ja ottaa talteen myöhempää käyttöä varten.

Esimerkiksi Euler-integraatiota vastaava ohjelma voidaan esittää OpenCL C-kielellä näin:

```
__kernel void summa(__global const float *x,
                    __global const float *v1,
                    __global const float dt,
                    __global float *x1){
    int i = get_global_id(0);
    x1[i] = x[i] + v1[i] * dt;
}
```

Listaus 1: OpenCL C -esimerkki

Tässä *__kernel* tekee summa-funktiosta "ytimen", jota voidaan käyttää CL-laitteen suoritusjonossa. Funktion parametreissä käytetty *__global*-osoiteavaruusmääre asettaa parametrin koko suorituksen kesken jaetuksi. Näitä parametrejä voidaan asettaa rajapinnan kutsuilla — tässä tapauksessa ne osoittavat laitteelta ennalta varattuihin puskureihin.

Funktiokutsu *get_global_id(0)* palauttaa tietylle kernel-funktion suoritukselle ominaisen indeksin (dimensiossa 0), joka ohjelmassa nimeää päivitettävän taulukon *x1* alkion.

Jos ohjelman sisällä määriteltävästä muuttujasta puuttuu osoiteavaruusmääre, kuten tässä muuttujasta i, se katsotaan osaksi vain tietyn suorituksen sisällä olemassaolevaa, paikallista *__private*-osoiteavaruutta.

3.4 OpenCL:n käyttö C-ohjelmassa

OpenCL-käyttökontekstin alustaminen, laitteen muistin varaaminen sekä ohjelman kääntäminen ja suorittaminen vaativat huomattavasti enemmän ohjelmakoodia. Tässä käydään läpi vain pääkohdat; kokonainen ohjelma esitellään liitteessä.

Aloittaakseen laskennan OpenCL-yhteensopivalla laitteella ohjelman tulee:

- Selvittää saatavilla olevat OpenCL-alustat (*clGetPlatformIDs*)
- Tiedustella joltakin alustalta sen käytössä olevia laitteita (*clGetDeviceIDs*)
- Valita laite ja luoda laitekonteksti, jolla viitataan siihen ohjelmassa (*clCreateContext*)
- Luoda komentojono laitteelle (*clCreateCommandQueue*, jonoja voi olla useita)
- Varata laitteelta muistia käsiteltävälle tiedolle (*clCreateBuffer*)
- Luoda laskentaohjelmaobjekti, syöttää sen lähdekoodi käännettäväksi ja luoda kernel-objekti (*clCreateProgramWithSource*, *clBuildProgram*, *clCreateKernel*)
- Kiinnittää laskentakernelin parametrit ja lisätä laskentatehtävä suoritusjonoon (*clSetKernelArg*, *clEnqueueNDRangeKernel*)

Monet CL-rajapinnan kutsut voidaan asettaa toimimaan asynkronisesti, jolloin kutsuvan ohjelman suoritus ei pysähdy odottamaan OpenCL-suorituksen valmistumista.

3.5 OpenCL 1.1

OpenCL-standardin uusi versio 1.1 lisää rajapintaan joitakin grafiikkakäytöstä tuttuja operaatioita ja tietotyyppejä, kuten kolmen elementin vektoreita, ja sallii OpenCL-komentojen lisäämisen suoritusjonoon useammasta säikeestä turvallisesti (lukuunottamatta *clSetKernelArg*-kutsua). Säikeistyksen tueksi rajapintaan on myös lisätty tukea mm. tapahtumien käsittelylle.

4 Muita kehitysvälineitä

Alunperin ainoat kehitysvälineet grafiikkasuorittimille olivat grafiikkarajapintoja, jotka hankaloittivat teknisesti alkeellisten grafiikkasuorittimien ohjelmointia entisestään. Ohjelmoinnin avuksi kehitettiin joitakin abstraktioita, kuten BrookGPU [7], jotka pyrkivät kätkemään eroja toimintatavoissa tai yksinkertaistamaan ohjelmointia. Sittemmin erikoistuneita laskentarajapintoja on ilmestynyt useita.

4.1 CUDA

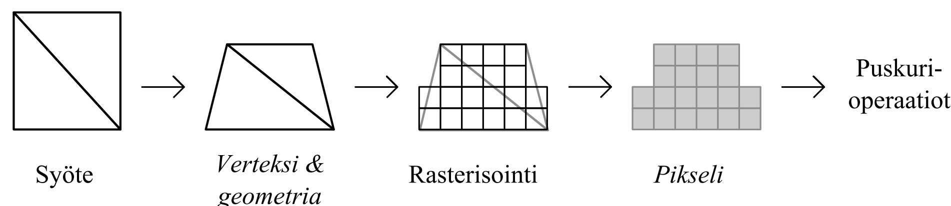
CUDA (Compute Unified Device Architecture) [1] on Nvidian pitkään suosiossa ollut laskentarajapinta. Uudempi OpenCL muistuttaa useilta ominaisuuksiltaan CUDA:a, eroten lähinnä termien valinnoissa, mutta CUDA on vahvemmin sidottu (Nvidian) grafiikkasuorittimien muisti- ja suoritusmalleihin. Matalamman abstraktiotason takia CUDA sallii joitakin OpenCL:ssä kiellettyjä operaatioita, kuten epäsuoria muistiviittauksia.

Huomattavin ero OpenCL:n ja CUDA:n ohjelmointikäytännöissä on CUDA:n tapa käyttää omaa C-kääntäjäänsä, joka sallii laskentalaitteelle syötettävän ohjelman kirjoittamisen samaan tiedostoon tavallisen C-koodin kanssa. Laitteella suoritettava koodi ja varastoitava tieto erotetaan pääohjelmasta CUDA:n C-kieleen lisäämillä määreillä, kuten __*device*__ ja __*host*__. Näin merkitty koodi käännetään eri tavalla. OpenCL:ssä laitteelle tähdätty OpenCL C-koodi syötetään pääohjelmasta ajurille, joka kääntää sen suorituksen aikana.

Nvidian OpenCL-toteutus on tätä kirjoittaessa rakennettu valmiin CUDA-arkkitehtuurin päälle — joka näkyy lähinnä virheellistä OpenCL-koodia kääntäessä syntyvissä virheilmoituksissa.

4.2 Grafiikkarajapinnat

Ensimmäiset yritykset käyttää grafiikkaprosessoreita muuhun laskentaan rakennettiin olemassaolevien grafiikkarajapintojen päälle [6]. Grafiikkarajapinnan käyttäminen muussa laskennassa voi olla hyödyllistä, jos ohjelmisto on ennestään siitä riippuvainen, tehtävä soveltuu kuvattavaksi grafiikkarajapinnan ehdoilla tai alustalle ei ole toimivaa toteutusta muista työkaluista.



Kuva 2: Grafiikkarajapinnan toiminta. Ohjelmoitavat vaiheet *korostettu*.

Perinteisellä grafiikkarajapinnalla laskemisen pääongelmina ovat hankala muistinkäsittely ja kuvaan 2 tiivistetty suoritusrakenne, jossa kirjoitettava muistin osa määräytyy etukäteen edellisten vaiheiden perusteella. Grafiikan kernel-ohjelmissa käytetyt varjostinkielet ovat monipuolisia, mutta eivät tämän rajoituksen takia voi jakaa tuloksiaan tehokkaasti.

Rajapintojen tapa kuvata tietoa ja laskea vastaa hyvin laitteen rakennetta, mutta monet algoritmit eivät sovellu esitettäväksi tässä muodossa ja tiedon pakkaaminen laitteen ymmärtämiin kuvaformaatteihin voi tehdä ohjelmista hankalaa luettavaa.

Nopeus on yleensä grafiikassa tarkkuutta merkittävämpää, joten grafiikkarajapintojen numeerinen tarkkuus on heikosti määritelty ja rajapinnan omat keinot virheenkäsittelyyn lähes olemattomia. Esim. OpenGL:n tapauksessa tietotyyppien numeerista tarkkuutta voi tiedustella *glGetShaderPrecisionFormat*-kutsulla [3], mutta useiden matemaattisten funktioiden tarkkuuksia varjostinkielessä ei määritellä [4]. Käytös virhetilanteissa jätetään toteuttajan päätettäväksi, kuitenkin OpenCL:ää vastaavalla ehdolla: virhe ei saa pysäyttää varjostinohjelman suoritusta.

Rajapinnat eivät siis tarjoa suoraan tapoja etsiä (tai edes havaita) suorituksessa tapahtuvia virheitä. Jos laskennan tulokset voidaan järkevästi esittää kuvana, yleinen tapa virheiden etsintään onkin ollut tietoa väreiksi koodaava “tulostuslause” varjostinohjelmassa [7]. Grafiikkarajapintoja käyttävien ohjelmien kehittämiseen ja vikojen etsintään on kuitenkin olemassa ulkoisia työkaluja, kuten ApiTrace [11].

Grafiikkarajapinnoissakin on tapahtunut kehitystä yleisemmän laskennan suuntaan. Microsoftin lähinnä peleissä käytetyn Direct3D-grafiikkarajapinnan 11. version mukana julkaistu DirectCompute laajentaa D3D:tä OpenCL:n kaltaisilla muistiavaruusmääreillä ja puskureilla, joihin voidaan kirjoittaa vapaasti "compute shader"-ohjelmista. Tätä kirjoittaessa DirectCompute on saanut huomiota lähinnä pelien tehostetyökaluna. Joitakin vastaavia toimintoja saatiin myös OpenGL:ään syksyllä 2011 julkaistussa OpenGL 4.2:ssa [3].

5 Grafiikkasuoritin kuvanpakkauksessa

Tutkitaan vielä JPEG-algoritmin sovittamista OpenCL:lle ja grafiikkasuorittimelle laitteen jo esiteltyjen ominaisuuksien pohjalta. Tavoitteena on tuottaa toimiva toteutus JPEG:n perustoiminnoista ja etsiä tapoja käyttää laitteen rakennetta hyväksi algoritmin optimoinnissa.

5.1 JPEG-algoritmi

JPEG on vuosina 1992-1993 standardoitu bittikarttakuvien pakkausmenetelmä, joka perustuu kaksiulotteiseen diskreettiin kosinimuunnokseen (*Discrete Cosine Transform, DCT*) ja Huffman-koodiin [8][9]. Nimi on peräisin standardin alunperin luoneelta Joint Photography Experts Groupilta.

Menetelmässä jaetaan ensin lähdekuva pakkausyksiköihin (yksinkertaisimmillaan 8×8 pikselin neliöihin) laajentaen kuvaa tarvittaessa, ja yksiköt käsitellään riveittäin alkaen kuvan vasemmasta yläkulmasta. Jokaisen yksikön sisältö s kosinimuunnetaan vastaavan kokoiseksi matriisiksi $\mathbf{S}$:

$$S_{vu} = \frac{1}{4}C(u)C(v)\sum_{x=0}^{7}\sum_{y=0}^{7} s_{yx} \cdot \cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16} \tag{1}$$

jossa $u, v \in [0, 7]$ ja

$$C(x) = 1/\sqrt{2} \quad \text{kun} \quad \text{x} = 0$$

$$C(x) = 1 \quad \text{muuten}$$

Tämän jälkeen matriisin sisältämät kosinitekijät supistetaan jakamalla ne alkioittain nk. kvantisointimatriisin (engl. *quantization matrix*) $\mathbf{Q}$ kanssa ja pyöristämällä tulos:

$$Sq_{vu} = round\left(\frac{S_{vu}}{Q_{vu}}\right)$$

Tämä on pakkauksen häviöllinen vaihe: pakkaussuhdetta voidaan hallita muuttamalla matriisin sisältöä. Vaiheen seurauksena suuri osa kuvan tekijöistä pyöristyy yleensä nollaksi. Huomattavaa on, että pakkaus on rajallisen lukutarkkuuden takia häviöllinen, vaikka $Q_{vu} = 1 \,\forall\, u, v \in [0, 7]$.

Viimeisessä *entropiakoodausvaiheessa* matriisin $\mathbf{Sq}$ alkiot järjestetään yksiulotteiseksi listaksi siten, että matriisia käydään läpi vinosuunnassa ja alkio Sq_{00} on listan ensimmäisenä. Koska tyypillinen kvantisointimatriisi sisältää suurempia jakajia korkeampia taajuuksia vastaaville tekijöille, järjestämisessä syntyvän listan nollasta eroavat alkiot keskittyvät suureksi osaksi sen alkuun, matalampia taajuuksia

vastaaviin kosinitekijöihin. Ensimmäinen arvo, eli kosiniyhtälöiden joukon vakiotekijä, koodataan erotuksena edellisestä vastaavasta arvosta. Loput nollasta eroavat arvot tallennetaan lyhyimpänä mahdollisena bittiesityksenään siten, että jokaista arvoa edeltää Huffman-symboli parille (edeltävien nollien lukumäärä, bittiesityksen pituus). Yli 15 nollan jaksoille ja pakkausyksikön lopulle käytetään omia Huffman-symboleitaan.

Pelkkä JPEG-standardi ei määrittele ollenkaan värikuvan tallennusmenettelyä. Tässä standardia on epävirallisesti täydennetty JPEG File Interchange Format (JFIF):n ohjeilla värikuvien käsittelystä JPEG-tiedostomuodossa [10]. Värikuvan tapauksessa väritieto muunnetaan yleensä JFIF:n mukaan YC_BC_R-muotoon, jossa Y-komponentti esittää kuvan kirkkautta ja C_B-, C_R-komponentit värisävyjä. Koska kirkkaustieto on ihmissilmän kannalta tärkeämpää eikä digitaalikameroiden ottamissa kuvissa usein ole mosaiikki- ja kohinasuodatuksien jälkeen yhtä tarkkaa väritietoa, väritieto voidaan varastoida matalammalla tarkkuudella (engl. *chroma subsampling*). Väriavaruusmuunnos aiheuttaa kuvaan ylimääräisen häviötekijän, sillä YC_BC_R on approksimaatio televisiossa käytetystä analogisesta väriavaruudesta eikä kuvaudu häviöttä RGB:ksi tai toisinpäin.

Algoritmi ei sovellu vektorigrafiikan bittikarttaesitysten tai nk. pikselitaiteen pakkaamiseen. Pakkausyksiköiden sisälle syntyvät häiriökuviot erottuvat helposti muuten tasaisista väreistä ja aaltoyhtälöjoukko on muutenkin heikko tapa esittää harvoja, teräviä reunoja.

JPEG määrittelee myös muita vaihtoehtoisia pakkausmenetelmiä, mutta vain pieni *baseline*-osajoukko standardista näyttää olevan aktiivisessa käytössä. Erityisesti viimeisen vaiheen Huffman-koodauksen vaihtoehtona esitelty aritmeettinen koodaus on huonosti tuettu. Standardi tosin luettelee kymmenen sitä mahdollisesti koskevaa patenttia vuosilta 1986-1992 [8].

JPEG-pakkausta muistuttava menetelmä on oleellinen osa monia yleisiä videonpakkausmenetelmiä, toimien mm. MPEG-videovirrassa esiintyvien kokonaisten kuvien (*keyframe*, MPEG: *I-frame*) pakkauksessa [12][13].

5.2 Laskentatehtävän sovitus

Pakkausvaiheet nollien koodaukseen asti ovat hyvin säännöllisiä ja vaikuttavatkin soveltuvan hyvin aiemmin esitettyihin grafiikkasuorittimen ominaisuuksiin.

Alkuun JPEG:n ominaiset pakkausyksiköt vaikuttaisivat luonnolliselta perustalta tehtävän ositukselle, mutta jos jokainen suoritustehtävä käsittelee eri pakkausyksiköitä, muistin luku- ja kirjoitusoperaatiot hajautuvat pakkausyksikön koon — 64

alkion — väleille. Paremmin laskentalaitteen rakennetta vastaava ohjelma saadaan osittamalla tehtävä vielä pienempiin osiin: luodaan jokaiselle pakkausyksikön osalle oma laskentatehtävänsä ja jaetaan paikalliset suoritusjoukot vastaamaan pakkausyksiköiden rajoja. Näin voidaan myös hyötyä suoritusjoukon kesken jaetusta paikallisesta muistista.

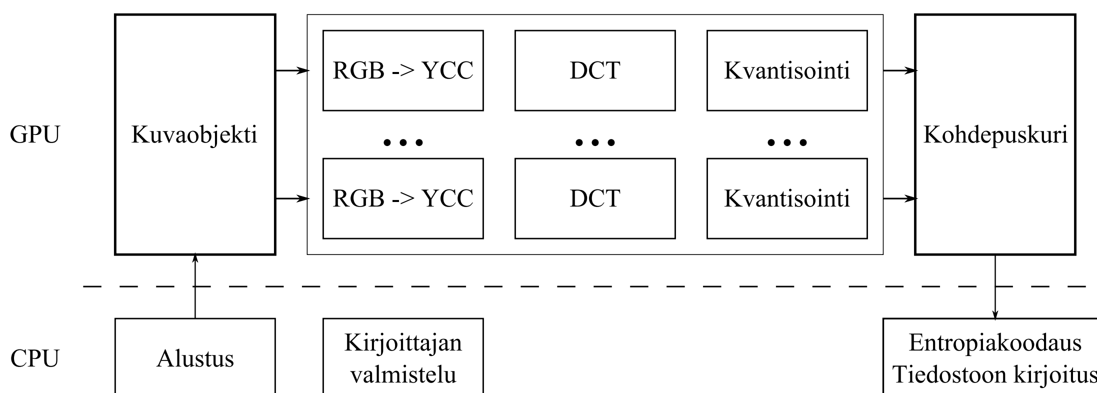
Koska OpenCL-kernelin toteuttamassa osassa algoritmiä pakkausyksiköiden välillä ei ole riippuvuuksia, ylimääräinen välituloksen kirjoitus kuvan väriavaruusmuunnoksessa voidaan välttää paikallisen muistin käytöllä. Riittääkin, kun jokainen työryhmä lukee oman kuva-alueensa sisällön ja muuntaa sen tarvittavaan väriesitysmuotoon paikallismuistiinsa.

Triviaali toteutus yhden kokonaisen pakkausyksikön kosinimuunnoksesta on vaativuudeltaan luokkaa $O(N^4)$, jossa N on neliömäisen pakkausyksikön leveys. Tietyn kosinitekijän laskeminen voidaan kuitenkin esittää kahtena yksiulotteisena muunnoksena. Käyttämällä OpenCL:n paikallista muistia välitulosten varastointiin, puretun muunnoksen riveittäin ja sarakkeittain suoritettavat osat voidaan hajauttaa koko työryhmän laskettavaksi.

OpenCL:n kuvaobjekteilla on joitakin tässä hyödyllisiä ominaisuuksia. Kuva-alan ulkopuolelta lukeminen on sallittua ja vierekkäisille kuvapisteille voidaan tarvittaessa laskea tehokkaasti keskiarvoja. Ensimmäinen ominaisuus on käytännöllinen kuvan laajentamisessa kahdeksalla jaollisiin mittoihin. Usein myös kuvaobjektien lukeminen on keskimäärin nopeampaa tehokkaan kuviin erikoistuvan välimuistin takia. Kuvaobjektien käyttö rajoittaa kerralla käsiteltävän kuvan koon laitteen tekstuuriyksikköjen rajoihin (ainakin 8192×8192 OpenCL 1.1:ssä). Useimmat JPEG-kuvat eivät kuitenkaan ole näin suuria. Standardi itse sallii korkeintaan 65535×65535-kokoisen kuvan.

Pakkauksen viimeinen vaihe jätetään ohjelman C-kielisen osan tehtäväksi JPEG/JFIF-tiedoston kirjoittamisen yhteydessä. Nollajonojen pituuksien etsintä ja niiden Huffman-koodaus yhdessä muiden symbolien kanssa hajaantuu vahvasti eikä vaikuta hyödylliseltä ajaa grafiikkasuorittimella, ellei pakkausta ohjaava keskussuoritin ole erityisen heikko suorituskyvyltään.

Käsiteltävän kuvatiedon kulkua ohjelmassa on hahmoteltu kuvassa 3.



Kuva 3: Jpegcl:n toiminta

5.3 Tuloksia

Suorituskyvyn arvioimiseksi ohjelman suoritusaikoja verrattiin yleisesti käytettyyn Independent JPEG Groupin [14] libjpeg:n cjpeg-komentorivityökaluun. Ohjelmat ajettiin samanmuotoisia ja mahdollisimman samankokoisia JPEG-tiedostoja tuottavilla parametreillä, ja ohjelmien suoritusaikoja mitattiin komentoriviltä. Mittauslaitteistona oli Intelin 3.6 GHz Core 2 Duo E8400 ja Nvidia GeForce GTX 560 Ti -grafiikkasuoritin.

Vertailukuvat olivat digitaalikameran ottama, aiemmin JPEG-pakattu kuva Kortepohjan luontomaisemasta, keskikokoinen raaka ruutukaappaus pelistä ja pieni pala web-grafiikkaa. Eri ohjelmien pakkaamissa kuvissa ei ollut näkyvää eroa.

OpenCL:n alustamiseen kuluva aika vaihteli rajusti. Keskimäärin aikaa kului n. 40 millisekuntia, mutta välillä jopa yli sekunti (tai vielä enemmän, jos laskentakerneliä oli muutettu). Tavoitteena ei ollut mitata ajurin valmisteluun kuluvaa aikaa, joten ohjelma muutettiin ilmoittamaan suoritusaika ilman OpenCL:n alustuskutsuja. Näin saatuja tuloksia on taulukoitu alla.

Jos pakattava kuva olisi valmiiksi näytönohjaimen muistissa (kuten olisi esim. kaapatessa videokuvaa ruudulta), jpegcl:n ei tarvitsisi kopioida kuvaa ensin näytönohjaimen muistiin. Lähdekuvan käsittely OpenCL:n kuvaobjektina sallii myös OpenGL:n tai muun grafiikkarajapinnan avulla tuotetun kuvan syöttämisen suoraan pakkauskernelille.

Suuremmissa kuvissa jpegcl-toteutuksen tapa muuntaa koko kuva kerralla ja kopioida se tiedostoon kirjoitettavaksi käy selvästi raskaaksi, ja tiedon kopioinnin ositus pienempiin askeliin voisi parantaa suorituskykyä. Koska OpenCL on suureksi osaksi asynkroninen rajapinta, kuvatiedon entropiakoodaus voisi käynnistyä, kun riittävä määrä JPEG:n koodausyksiköitä on jo tuotettu laskentakernelillä ja siirretty

Kuvan mitat	Toteutus	Tiedoston koko	Suoritusaika
3456 x 2592	IJG libjpeg 8c cjpeg	3605 KiB	695 ms
	jpegcl 0.6	3597 KiB	368 ms
1680 x 1050	IJG libjpeg 8c cjpeg	413 KiB	152 ms
	jpegcl 0.6	412 KiB	75 ms
504 x 312	IJG libjpeg 8c cjpeg	31 KiB	31 ms
	jpegcl 0.6	31 KiB	13 ms

Taulukko 1: JPEG-toteutusten suoritusaikoja

päämuistiin.

Keskikokoisiin kuviin jpegcl näyttäisi soveltuvan hyvin. Keskikokoisten ja pienien kuvien tapauksessa tehtävän purkamisesta pienempiin vaiheisiin tuskin olisi enää hyötyä, sillä laskentakernelin suoritusaika on lähes yhtä lyhyt kuin viive suorituksen aloittamisessa.

Ottaen huomioon, että lähes puolet pakkausajasta kuluu kirjoittaessa Huffman-koodattuja symboleita tiedostoon, pakkauksen puolittunut suoritusaika on huomattava parannus.

Jpegcl:n tarkempaa profilointia varten suorituksen blokkaavien OpenCL-kutsujen ympärille aseteltiin ajanmittauskoodia. Keskikokoisen kuvan tapauksessa suoritusaika jakautui seuraavasti:

```
Source read and upload: 0.028 s
Colorspace/DCT        : 0.013 s
Entropy coding        : 0.034 s (45.3% of total)
```

Koska aritmetiikan määrä kosinimuunnoksessa ei näytä erityisesti vaikuttavan jpegcl:n suorituskykyyn, monimutkaisempaa muunnosta käyttävä kuvanpakkausalgoritmi voisi olla grafiikkasuoritinta käyttäessä hyödyllinen. Esim. JPEG 2000 ja JPEG XR (entinen Microsoft HD Photo) käyttävät JPEG:iä monimutkaisempia muunnoksia paremman kuvanlaadun saavuttamiseksi. Molemmat standardit ovat tosin huomattavasti JPEG:iä monimutkaisempia toteuttaa, ja uuden mediastandardin kyseessäollessa patenttiongelmat ovat mahdollisia.

6 Yhteenveto

Työssä esiteltiin lyhyesti grafiikkasuorittimien historiaa, toimintaperiaatteita ja niiden tärkeimpiä suorituskykyyn vaikuttavia ominaisuuksia. Tarkemmin perehdyttiin OpenCL:ään grafiikkasuoritinten ja muidenkin rinnakkaisten laskentalaitteiden ohjelmointivälineenä. OpenCL:n muisti- ja suoritusmallit käytiin läpi ja OpenCL:ää käyttävän ohjelman rakennetta kuvailtiin. Muita grafiikkasuorittimen ohjelmoinnissa käytettäviä rajapintoja kuvailtiin lyhyesti.

Esimerkkitapauksena käytettiin JPEG-kuvanpakkausmenetelmää, jonka tärkeimmät toimintaperiaatteet esiteltiin lyhyesti. Käyttäen tietoa grafiikkasuorittimien toimintaperiaatteista, OpenCL:n päälle rakennettiin suorituskyvyltään melko kilpailukykyinen ja tarvittaessa hyvin suurelle suoritinmäärälle hajautuva toteutus JPEG-menetelmästä.

Työssä syntynyt *jpegcl*-kirjasto voisi pienillä parannuksilla olla käyttökelpoinen väline ohjelmassa, jossa kuvia käsitellään jo pääosin grafiikkasuorittimella. Ohjelman lähdekoodin tärkeimmät osat ovat liitteenä.

Lähteet

[1] Nvidia Corporation, *CUDA Programming Guide 1.1*.
Saatavilla WWW-muodossa:
<URL: `http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf` >. Viitattu 10.1.2012.

[2] Khronos Group, *The OpenCL Specification, Version 1.1*.
Saatavilla WWW-muodossa:
<URL: `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf` >.
Viitattu 10.1.2012.

[3] Khronos Group, *The OpenGL Graphics System: A Specification*, versio 4.2.
Saatavilla WWW-muodossa:
<URL: `http://www.opengl.org/registry/doc/glspec42.core.20110808.pdf` >.
Viitattu 10.1.2012.

[4] Khronos Group, *The OpenGL Shading Language*, versio 4.20.
Saatavilla WWW-muodossa:
<URL: `http://www.opengl.org/registry/doc/GLSLangSpec.4.20.6.clean.pdf` >.
Viitattu 23.1.2012.

[5] Vasily Volkov, James Demmel, *LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs (EECS-2008-49),* , 2008. Saatavilla WWW-muodossa:
<URL: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-49.html` >.
Viitattu 10.1.2012.

[6] Peter Bui, Jay Brockman, *Performance Analysis of Accelerated Image Registration using GPGPU*, 2009. Julkaistu osana "Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units":a, ISBN: 978-1-60558-517-8.

[7] John D. Owens, ym., *A Survey of General Purpose Computation on Graphics Hardware*, 2007.

[8] ISO/IEC, *10918-1, Information technology — Digital compression and coding of continuous-tone still images — Requirements and guidelines*, 1992.

[9] Gregory K. Wallace, *The JPEG Still Picture Compression Standard*, 1991. Saatavilla WWW-muodossa: <URL: `http://dl.acm.org/citation.cfm?id=103089` >.
Viitattu 22.1.2012.

[10] Eric Hamilton, *JPEG File Interchange Format, Version 1.02*, 1992. Saatavilla WWW-muodossa: <URL: `http://www.jpeg.org/public/jfif.pdf` >. Viitattu 22.1.2012.

[11] Zack Rusin, *apitrace*, https://github.com/apitrace/. Viitattu 22.1.2012.

[12] ISO/IEC, *13818-2, Information technology — Generic coding of moving pictures and associated audio information: Video*, 1995.

[13] ISO/IEC, *14492-2, Information technology — Coding of audio-visual objects — Part 2: Visual*, 2001.

[14] Independent JPEG Group, http://www.ijg.org/. Viitattu 22.1.2012.

1 Lähdekoodia

1.1 OpenCL C -kerneli

Sopiva tekstuuriyksikön tila riittää toteuttamaan JPEG:n vaatima kuvan laajentaminen pakkausyksikköjen mittoihin:

```
const sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE
                        | CLK_ADDRESS_CLAMP_TO_EDGE
                        | CLK_FILTER_NEAREST;
```

Listaus 2: Sampler-yksikössä käytetty tila

Tässä lauseen ensimmäinen vakio määrää, ettei kuvassa käytettäviä koordinaatteja skaalata välille [0..1]. Kuvan rajojen ulkopuolelta lukeminen määritellään tilalla *CLK_ADDRESS_CLAMP_TO_EDGE*: kuva-alueen ylittävät koordinaatit rajoitetaan sen sisälle. Kuvaa ei tarvitse skaalata, joten kuvan suodatus kytketään vielä pois.

OpenCL-laskentakerneli *ftrans3* toteuttaa JPEG:n väriavaruus- ja kosinimuunnokset. Parametrit ovat lähdekuva, puskuri johon kosinimuunnetut ja kvantisoidut tekijät kirjoitetaan ja käytettävät kvantisointimatriisit. OpenCL ei muuten takaa paikallisen muistin eheyttä, joten ennen muistin lukua saman ryhmän laskentatehtävien synkronointi varmistetaan *barrier*-kutsuilla. Kutsu ei vaikuta olevan erityisen raskas laitteella, johtunee raudalla toteutetusta suoritustehtävien hallinnasta.

```
__constant float PI = 3.14159265358979323846f;

float RGB_to_Y(float3 rgb) {
    return dot(rgb, (float3){ 0.299f, 0.587f, 0.114f });
}
float RGB_to_Cb(float3 rgb) {
    return dot(rgb.xyz, (float3){ -0.1687f, -0.3313f, 0.5f }) + 0.5f;
}
float RGB_to_Cr(float3 rgb) {
    return dot(rgb.xyz, (float3){ 0.5f, -0.4187f, -0.0813f }) + 0.5f;
}
float3 RGB_to_YCbCr(float3 rgb) {
    return (float3) {
        dot(rgb, (float3){ 0.299f, 0.587f, 0.114f }),
        dot(rgb.xyz, (float3){ -0.1687f, -0.3313f, 0.5f }) + 0.5f,
        dot(rgb.xyz, (float3){ 0.5f, -0.4187f, -0.0813f }) + 0.5f
    };
}

// Performs 2D DCT for a given work-item. All 64 work-items together
// will first perform a row-by-row 1D DCT, then a second one by
```

```
// columns, storing intermediate results in local memory.
float fdct_2d(__local float *values, size_t u, size_t v) {
    __local float out[64];
    float sum = 0;

    float U16 = (float)u * PI / 16.0f;
    float V16 = (float)v * PI / 16.0f;

    // Arithmetic really does NOT seem like the bottleneck here,
    // but let's decompose the 2D DCT for style anyway.

    for(size_t x=0; x<8; x++) {
        sum += values[v*8+x] * native_cos(U16 * (float)(2*x+1));
    }
    out[v*8+u] = sum;

    barrier(CLK_LOCAL_MEM_FENCE);
    sum = 0;

    for(size_t y=0; y<8; y++) {
        sum += out[y*8+u] * native_cos(V16 * (float)(2*y+1));
    }

    sum *= 63.75f;
    if(u == 0) sum /= sqrt(2.0f);
    if(v == 0) sum /= sqrt(2.0f);

    return sum;
}

// Transforms RGB to Y, Cb and Cr and writes DCT coefficients to
// dst in JPEG file order, quantized by the matrices provided.
__kernel void ftrans3(__read_only image2d_t src
                     , __global short *dst
                     , __global uchar *quants
                     , __global uchar *quants2) {
    __local float local_color[64];

    size_t pos_x = get_global_id(0);
    size_t pos_y = get_global_id(1);
    size_t width = get_global_size(0);

    size_t block_x = get_group_id(0);
    size_t block_y = get_group_id(1);
```

```
    size_t blocks_h = get_num_groups(0);

    size_t block_offset = (block_y * blocks_h + block_x) * 64;

    size_t u = get_local_id(0);
    size_t v = get_local_id(1);

    int2 coord = (int2){ block_x * 8, block_y * 8 };

    // convert RGB to luma
    float3 rgb = read_imagef(src, sampler, coord+(int2){u, v}).xyz;
    local_color[v*8+u] = RGB_to_Y(rgb) - 0.5f;

    // wait for workgroup to finish sharing
    barrier(CLK_LOCAL_MEM_FENCE);

    float sum = fdct_2d(local_color, u, v);

    // quantize
    short final = (short)round(sum / (float)quants[v*8+u]);
    dst[3*block_offset + v*8+u] = final;

    // convert RGB to Cb
    local_color[v*8+u] = RGB_to_Cb(rgb) - 0.5f;
    barrier(CLK_LOCAL_MEM_FENCE);
    sum = fdct_2d(local_color, u, v);
    final = (short)round(sum / (float)quants2[v*8+u]);
    dst[3*block_offset + 64 + v*8+u] = final;

    // convert RGB to Cr
    local_color[v*8+u] = RGB_to_Cr(rgb) - 0.5f;
    barrier(CLK_LOCAL_MEM_FENCE);
    sum = fdct_2d(local_color, u, v);
    final = (short)round(sum / (float)quants2[v*8+u]);
    dst[3*block_offset + 128 + v*8+u] = final;
}
```

Listaus 3: JPEG:n kosinimuunnos ja väritransformaatiot OpenCL C:llä

1.2 C-pääohjelma

Allaolevassa C99-lähdekoodissa käydään läpi JPEG-pakkauksessa käytetyn OpenCL:ää kutsuvan C-koodin rakenne.

Pakkaajan tilaa kuvaa rakenne *JCL*:

```
typedef struct {
    cl_context context;
    cl_device_id device;
    cl_command_queue queue;

    cl_program   program;

    cl_kernel    k_ftrans;        // forward transform, grayscale
    cl_kernel    k_ftrans3;       // forward transform, YCbCr
    cl_kernel    k_rgb_to_ycbcr;  // color space conversion, RGB ->
        YCbCr

    cl_mem       m_quant_y;       // quantization matrix for luma
    cl_mem       m_quant_c;       // quantization matrix for chroma

    cl_mem       m_src;           // source image being worked on
    size_t       src_width;
    size_t       src_height;

    size_t       src_width_pad;
    size_t       src_height_pad;

    // output buffers
    cl_mem       m_out[JCL_NUM_BUFFERS];

    const unsigned char *q_y;
    const unsigned char *q_c;

} JCL;
```

Listaus 4: Jpegcl-ohjelman tilarakenne

OpenCL:n alustus tapahtuu funktiossa `JCL_init`:

```
// try to avoid if(err != ... everywhere
#define JPEGCL_INIT_ERRORCHECK(desc) \
    if(err != CL_SUCCESS) { \
        fprintf(stderr, "jpegcl_init: failed: %s,\n CL: %s\n", desc,
            cl_err_str(err)); \
        return NULL; \
    }

JCL *jpegcl_init() {
    JCL *jcl = malloc(sizeof(JCL));

    cl_platform_id *platforms;
```

```
cl_uint num_platforms;
cl_int err;

int num_platform = 0;
int num_device = 0;
unsigned int i;

gettimeofday(&init_start, NULL);

err = clGetPlatformIDs(0, NULL, &num_platforms);
JPEGCL_INIT_ERRORCHECK("get␣number␣of␣platform␣IDs");

platforms = malloc(sizeof(cl_platform_id) * num_platforms);
err = clGetPlatformIDs(num_platforms, platforms, NULL);
JPEGCL_INIT_ERRORCHECK("get␣list␣of␣platforms");

printf("jpegcl_init:␣CL␣platforms:\n");
for(i=0; i<num_platforms; i++) {
    print_cl_platform_info(platforms[i]);
}

cl_context_properties props[] = {
    CL_CONTEXT_PLATFORM,
    (cl_context_properties)platforms[num_platform],
    (cl_context_properties)NULL
};

cl_context context;
context = clCreateContextFromType(props, CL_DEVICE_TYPE_GPU, NULL
                                          , NULL, &err);
jcl->context = context;

free(platforms);

JPEGCL_INIT_ERRORCHECK("create␣context␣for␣platform");

// get length of device list
size_t len_devices;
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL,
   &len_devices);

// alloc and fill device list
cl_device_id *devices = malloc(len_devices);
```

```
clGetContextInfo(context, CL_CONTEXT_DEVICES, len_devices,
    devices, NULL);

// pick one
printf("jpegcl_init: CL devices on platform %i:\n",
    num_platform);
for(i=0; i<len_devices/sizeof(cl_device_id); i++) {
    print_cl_device_info(devices[i]);
}
cl_device_id device = devices[num_device];
jcl->device = device;
free(devices);

printf("jpegcl_init: using platform %i, device %i.\n"
        , num_platform, num_device);
cl_command_queue queue = clCreateCommandQueue(context, device,
    0, &err);
JPEGCL_INIT_ERRORCHECK("create command queue on device");

jcl->queue = queue;

// set up program and kernels
char *str = read_file("jpeg.cl");
if(!str) {
    printf("jpegcl_init: unable to read jpeg.cl!\n");
    return 0;
}
const char *source = str;
cl_program program = clCreateProgramWithSource(context, 1,
    &source, NULL, &err);
free(str);
JPEGCL_INIT_ERRORCHECK("create program from source");

err = clBuildProgram(program, 1, &device, "", NULL, NULL);
size_t len_buildinfo;
clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, 0,
    NULL, &len_buildinfo);
if(len_buildinfo > 2) {
    char build_log[len_buildinfo];
    printf("jpegcl_init: build info length: %i\n",
        len_buildinfo);
    clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG
                        , len_buildinfo, build_log, NULL);
    printf("jpegcl_init: build log:\n%s\n", build_log);
```

```
}
if(err != CL_SUCCESS) {
    printf("jpegcl_init:␣program␣build␣failed:␣%s\n",
        cl_err_str(err));
    return NULL;
}
jcl->program = program;

cl_kernel k_rgb_to_ycbcr = clCreateKernel(program,
    "rgb_to_ycbcr", &err);
JPEGCL_INIT_ERRORCHECK("create␣RGB␣to␣YCbCr␣kernel");
cl_kernel k_ftrans = clCreateKernel(program, "ftrans3", &err);
JPEGCL_INIT_ERRORCHECK("create␣forward␣transform␣(Y)␣kernel");
cl_kernel k_ftrans3 = clCreateKernel(program, "ftrans3", &err);
JPEGCL_INIT_ERRORCHECK("create␣forward␣transform␣(YCbCr)␣
    kernel");

cl_mem m_quant_y = clCreateBuffer(context, CL_MEM_READ_ONLY, 64,
    NULL,&err);
JPEGCL_INIT_ERRORCHECK("create␣buffer␣for␣luma␣quantization␣
    matrix");
cl_mem m_quant_c = clCreateBuffer(context, CL_MEM_READ_ONLY, 64,
    NULL,&err);
JPEGCL_INIT_ERRORCHECK("create␣buffer␣for␣chroma␣quantization␣
    matrix");

jcl->k_rgb_to_ycbcr = k_rgb_to_ycbcr;
jcl->k_ftrans = k_ftrans;
jcl->k_ftrans3 = k_ftrans3;
jcl->m_quant_y = m_quant_y;
jcl->m_quant_c = m_quant_c;
jcl->src_width = 0;
jcl->src_height = 0;
jcl->src_width_pad = 0;
jcl->src_height_pad = 0;

jcl->m_src = NULL;
for(i=0; i<JCL_NUM_BUFFERS; i++) {
    jcl->m_out[i] = NULL;
}

// let's make a more interesting matrix
uint8_t *mat = malloc(64);
uint8_t *mat2 = malloc(64);
```

```
    int u,v;

    for(v=0; v<8; v++) {
        for(u=0; u<8; u++) {
            // make some quantizer vals
            mat[v*8+u] = std_luma_quantizers[v*8+u] / 2.5;
            mat2[v*8+u] = std_chroma_quantizers[v*8+u] / 2.5;
        }
    }
    jpegcl_i_setmatrices(jcl, mat, mat2);

    gettimeofday(&init_end, NULL);

    return jcl;
}
```

Listaus 5: Jpegcl-ohjelman alustus

Funktio *jpegcl_run* kutsuu *ftrans3*-kerneliä, lukee tulokset ja syöttää ne JPEG-tiedostoon kirjoittavalle *JFIFWriter*:ille.

```
int jpegcl_run(JCL *jcl, const char *outfile) {
    cl_int err;
    size_t width_pad = jcl->src_width_pad;
    size_t height_pad = jcl->src_height_pad;

    // 3 short-type coefficents per pixel
    size_t final_size = width_pad * height_pad * sizeof(short) * 3;

    jcl->m_out[0] = clCreateBuffer(jcl->context, CL_MEM_WRITE_ONLY,
        final_size, NULL, &err);
    if(err != CL_SUCCESS) {
        printf("jpegcl_run:␣unable␣to␣create␣target␣buffer(s)!␣
            (%s)\n"
                , cl_err_str(err));
        return 0;
    }

    cl_uint dct_global_size[] = { width_pad, height_pad };
    cl_uint dct_local_size[] = { 8, 8 };

    clSetKernelArg(jcl->k_ftrans, 0, sizeof(cl_mem), &jcl->m_src);
    clSetKernelArg(jcl->k_ftrans, 1, sizeof(cl_mem), &jcl->m_out[0]);
    clSetKernelArg(jcl->k_ftrans, 2, sizeof(cl_mem),
        &jcl->m_quant_y);
    clSetKernelArg(jcl->k_ftrans, 3, sizeof(cl_mem),
```

```
        &jcl->m_quant_c);
    clEnqueueNDRangeKernel(jcl->queue, jcl->k_ftrans, 2, NULL
                            , dct_global_size, dct_local_size, 0,
                              NULL, NULL);


    JFIFWriter *jw = JW_init(outfile, jcl->src_width,
        jcl->src_height, 3, jcl->q_y, jcl->q_c);

    // read results
    short *out = clEnqueueMapBuffer(jcl->queue, jcl->m_out[0],
        CL_TRUE, CL_MAP_READ, 0
                                    , final_size, 0
                                    , NULL, NULL, &err);
    JPEGCL_COMPRESS_ERRORCHECK("map DCT'd luma");

    gettimeofday(&begin_huffman, NULL);

    JW_write_scan(jw, out, width_pad * height_pad);

    clEnqueueUnmapMemObject(jcl->queue, jcl->m_out[0], out, 0, NULL,
        NULL);
    JW_close(jw);

    return 1;
}
```

Listaus 6: Jpegcl-ohjelman pakkausfunktio

OpenCL-kuvaobjektin luonti ja lähdekuvan lukeminen objektin sisällöksi on toteutettu *jpegcl_readimage*-funktiossa.

```
int jpegcl_readimage(JCL *jcl, const char *file) {
    Image *img = Image_readBMP(file);
    cl_int err;

    if(!img) {
        printf("jpegcl_readimage: unable to read %i\n");
        return 0;
    }

    /// @todo handle larger MCUs when subsampling
    jcl->src_width_pad = pad_to_multiple(img->width, 8);
    jcl->src_height_pad = pad_to_multiple(img->height, 8);

    // do we need a new source buffer?
```

```
if(jcl->src_width != img->width || jcl->src_height !=
   img->height) {
    cl_image_format format;

    format.image_channel_order = CL_BGRA;
    format.image_channel_data_type = CL_UNORM_INT8;
    jcl->src_width = img->width;
    jcl->src_height = img->height;

    if(jcl->m_src) {
        clReleaseMemObject(jcl->m_src);
    }

    jcl->m_src = clCreateImage2D(jcl->context, CL_MEM_READ_ONLY,
       &format
                                  , img->width, img->height, 0,
                                    NULL, &err);
    if(err != CL_SUCCESS) {
        printf("jpegcl_readimage:␣can't␣allocate␣image␣object!␣
           (%s)\n"
                , cl_err_str(err));
        Image_free(img);
        return 0;
    }

    printf("jpegcl_readimage:␣allocated␣new␣%i␣x␣%i␣image.\n",
       img->width, img->height);
}

size_t origin[] = { 0, 0, 0 };
size_t src_region[] = { img->width, img->height, 1 };
size_t row_pitch;
uint8_t *ptr;

ptr = clEnqueueMapImage(jcl->queue, jcl->m_src, CL_TRUE,
   CL_MAP_WRITE
                        , origin, src_region, &row_pitch
                        , NULL, 0, NULL, NULL, &err);
if(!ptr) {
    printf("jpegcl_readimage:␣clEnqueueMapImage␣failed!␣(%s)\n"
           , cl_err_str(err));
    Image_free(img);
    return 0;
}
```

```
    size_t x, y;
    uint8_t *data = img->data;
    uint32_t *pixels = (uint32_t*)ptr;
    uint32_t col = 0;
    for(y=0; y < img->height; y++) {
        size_t row_ofs = y * img->width;
        for(x=0; x < img->width; x++) {
            col = ((uint32_t)data[(row_ofs+x)*3 + 0] << 0)
                + ((uint32_t)data[(row_ofs+x)*3 + 1] << 8)
                + ((uint32_t)data[(row_ofs+x)*3 + 2] << 16);
            pixels[y*row_pitch/4+x] = col;
        }
    }
    clEnqueueUnmapMemObject(jcl->queue, jcl->m_src, ptr, 0, NULL,
       NULL);
    clFinish(jcl->queue);

    Image_free(img);
    return 1;
}
```

Listaus 7: Kuvan syöttö OpenCL:lle