(OpenCL) ohjelmointiopas

vaativaan ja edistyneeseen ohjelmointiin

Alkusanat

Tämä opas kertoo, kuinka päästä nopeasti kiinni rinnakkaislaskentaan OpenCL-rajapintaa hyväksikäyttäen. Lukijan on syytä hallita ohjelmoinnin perustietämys. Ilman ohjelmointiosaamistakin tämä opas voi olla, jos ei mitään muuta, niin ainakin mielenkiintoinen ikkuna ohjelmoinnin maailmaan: esimerkkikoodit on pyritty tekemään mahdollisimman hyvin (ellei, tästä on selkeä maininta ja syy), joten niistä löytyvät kommervenkit ja kikat ovat suurimmilta osin tarpeellisia, eli niillä on aina jokin perustelu.

Tämän oppaan rinnalla on hyvä pitää auki varsinainen OpenCL referenssistandardi. Etenkin kappaleesta ”3. The OpenCL Architecture”, jossa kuvataan OpenCL-arkkitehtuuri kuvituksen kera on hyötyä. OpenCL:n liittyviä asioita (ei ohjelmointimielessä) ei ole tässä pyritty turhaan toistamaan, sillä niiden osalta niin virallinen, kuin epävirallinenkin eli niin valmistajien, kuin tavallisten käyttäjienkin tekemä materiaali on suurimmilta osin hyvälaatuista: ei välttämättä selkeää, mutta asiavirheitä ei niinkään ole. Sen sijaan ohjelmointimielessä tilanne on päinvastainen. Esimerkit ovat heikkolaatuisia ja sisältävät paljon turhaa toistoa, toisin sanoen, ovat teknisessä arvostelumielessä ala-arvosia: ei ole oikein ymmärretty, mitä ollaan tekemässä.

Tämän oppaan esimerkkikoodit ovat lisenssoitu LGPL-lisenssillä. Niitä voi siis vapaasti hyödyntää niin ilmaisessa kuin kaupallisessakin käytössä, joskin koodit niihin tehtyine muutoksineen on aina luovutettava niitä pyytävälle. Käytetty lisenssi ei kuitenkaan ”myrkytä” sitä käyttävää ohjelmakoodia, eli kaupallisessa tapauksessa varsinaista rahaa tahkoava osuus pysyy juuri sen piirin tietona, kuin ohjelmakoodin omistama taho haluaa. Käytännössä esimerkkikoodien toiminallisuus liittyy kaikkeen sellaiseen ns. ylimääräiseen aputoimintaan, millä kukaan ei sinällään luo tiliä; se ainoastaan helpottaa alkuunpääsyä, sekä auttaa havaitsemaan kaikkia niitä tilanteita, johin C++-ohjelmointikielellä on jo valmis ratkaisu integroituna suoraan kieleen itseensä. Aikaa ei siis tule tule tuhlattua pyörän keksimiseen uudelleen ja voidaan keskittyä laskennallisen ongelman parhaaseen mahdolliseen ratkaisuun.

Suomenkieliset termistöt eivät tahdo oikein pysyä kehityksen mukana, joten termeistä käytetty englanninkielinen sana on myös esitetty aiheen yhteydessä. Tämä helpottaa lähdemateriaalin etsimistä, sillä suomenkielisen termin perusteella voi olla etenkin asiaan vihkiytymättömän vaikea löytää tietoa.

Oulussa, 20.8.2015

Pekka Seppänen

OpenCL-ohjelmointi

OpenCL-rajapinta on vapaa ja avoin C-kielinen rajapinta, jonka avulla voidaan ohjata laitteita, jotka soveltuvat rinnakkaislasketaan. Tyypillisin esimerkki on tietokoneen näytönohjain, *Graphical Prosesing Unit*, eli GPU. Laskentaa voidaan myös suorittaa normaalilla suorittimella (CPU), mutta nopeusetua ei tällöin käytännössä saavuteta. Sen sijaan eri tilanne on, jos käytettävissä on esimerkiksi jokin FPGA-piiri (tai miksei myöskin ASIC-piiri kehityksen myöhemmässä vaiheessa), jonka valmistaja tarjoaa tälle OpenCL-tuen.

OpenCL-rajapinta jakautuu kahteen pääosaan: isäntään (*host*), joka käyttää OpenCL-rajapintaa ja ytimeen (*kernel*), joka suorittaa varsinaisen laskemisen. Teknisesti isäntä voi olla toteuttu millä tahansa ohjelmointikielellä, josta voi kutsua C-funktioita. Sulautetussa järjestelmässä tämä voi olla esimerkiksi ARM-suoritin, joka toimii sulautetussa Linux-ympäristössä. Isännän tehtävä on ohjata kaikkea niitä toimenpiteitä, joita tarvitaan laskemisen avuksi lähtien syötteiden lukemisesta ja muuttamisesta haluttuun muotoon. Ydin ohjelmoidaan aina OpenCL-kielellä. Se on C-pohjainen kieli, johon on lisätty joitakin juuri rinnakkaislasketaan soveltuvia ominaisuuksia. Käytännössä näitä ovat muuttuja- ja funktiotarkenteet sekä vektorityypit. Ytimen ohjelmointi eroaa siis hyvin vähän tyypillisestä sulautetun järjestelmän C-ohjelmoinnista. Suurin konkreettinen ero on se, että OpenCL-kielelle ei ole olemassa erillistä kääntäjää, vaan ytimen lähdekoodi syötetään OpenCL-rajapinnan lävitse ja rajapinnan toteutus huolehtii ohjelmakoodin kääntämisestä.

Laskenta voidaan suorittaa 1-, 2- tai 3-ulotteiselle syötteelle. Rajapinta kutsuu ydintä yhden kerran kohti jokaista elementtiä. 1-ulotteisella syötteellä suoritetaan rinnakkain aina N (eli $N \times 1$) kappaletta ytimiä ja 2- ja 3-ulotteisilla syötteillä $N \times M$ kappaletta. Se, kuinka suuriksi N ja M voidaan valita riippuu laskentaan käytettävästä laitteesta. Tyypillisesti N *tai* $M <= 32$. Alueesta käytetään nimitystä *workgroup* eli työryhmä. On huomioitava, että ydin voi jakaa laskentaan tarvittavaa väliaikaista tietoa ainoastaan yhden työryhmän sisällä.

Rinnakkaisuuden ohella isäntä voi myös hajauttaa laskennan useammalle eri laitteelle. Tällöin isäntä lohkoo tai pilkkoo syötteen parhaaksi katsomallaan tavalla ja syöttää pilkotut lohkot eri laitteille. Laskennan jälkeen tieto yhdistetään lopulliseksi tulokseksi.

Käännösympäristö

Esimerkkiohjelmaksi on valittu kuvan pyörittäminen kierto- tai leikkausmatriisin (*shear matrix*) avulla. Isäntäohjelma on ohjelmoitu *C++14*-kielellä. Syy tähän on yksinkertainen. Käyttäjä tai ohjelmoija joutuu tekemään hyvin paljon sellaista työtä käsin, jonka voi välttää ottamalla kaiken irti ohjelmointikielestä. Tällöin ohjelmointivirheiden määrä voidaan jo lähtökohtaisesti minimoida, sillä ohjelmakoodia ei tarvitse ”leikata ja liimata” ja pystytään keskittymään itse ytimen suunnitteluun ja toteutukseen.

Ohjelmointiympäristöksi on tarkoituksella valittu askeettinen terminaaliympäristö, kääntäjäksi *GCC 4.9* ja kääntöä ohjaamaan *GNU Make*. On ehdottoman tärkeää, että GCC-kääntäjää käytettäessä versionumero on vähintään 4.9, muutoin kääntäjä ei sisällä kaikkia tarvittavia ominaisuuksia ja isäntäohjelmaa ei saada käännettyä. Sinänsä mikä tahansa standardin täyttävä kääntäjä soveltuu, muita toimivia kääntäjiä on esimerkiksi *Clang*. Sen sijaan esimeriksi edes uusi *Microsoft Visual Studio 2015* ei sovellu, sillä sen mukava tuleva *MSVC* kääntäjä ei ole vielä ”aivan tätä päivää”.

Ohjelmakoodi toimii sekä *Windows*- että *Linux*-käyttöjärjestelmissä. Jälkimmäisen kanssa käännöstyökalut ovat helposti asennettavissa, mutta Windows-puolella tämä ei ole niin itsestäänselvyys: terminaaliympäristöksi on syytä valita *Cygwin*, mutta kääntäjänä *ei* tule käyttää Cygwinin omaa GCC-kääntäjää, vaan *MinGW64*:n 64-bittistä GCC-kääntäjää[1]. Tällöin vältytään siltä, että käännetyn ohjelman matkaan tulisi liittää Cygwinin dynaaminen apukirjasto. GNU Maken tehtävä on ohjata käännöstä siten, että *Makefile*-tiedoston perusteella määritellään käännettävät tiedostot ja niiden riippuvuudet. Kun käytössä on GCC-kääntäjä, sopivalla käännösvivulla kääntäjä laatii automaattisesti kunkin *.cpp*-lähdekooditiedoston perusteella listan sen vaatimista *.hpp*-määrittelytiedostoista. Windows-käyttöjärjestelmää käytettäessä tässä piilee yksi ongelmakohta, sillä kääntäjän on käytettävä samaa tiedostopolkujärjestelmää[2] terminaaliympäristön kanssa.

Laskenta on suunniteltu tehtäväksi näytönohjaimella, ja tätä varten tulee asentaa näytönohjaimen valmistajan toimittama OpenCL-kehityspaketti. Nvidialla valmistamilla näytönohjaimilla tulee siis asentaa ”CUDA Toolkit” ja vastaavasti AMD:n tapauksessa ”AMD OpenCL APP SDK”. Intel ei tarjoa näytönohjaimilleen (mikäli tällainen on esimerkiksi integroitu ko. valmistajan suorittimeen) ilmaista OpenCL-rajapintaa. Valitettavasti GNU Maken avulla ei voida tunnistaa ja täten valita automaattisesti (millään järkevällä tapaa) asennettua OpenCL-kehityspakettia, joten se tulee muuttaa käsin Makefile-tiedostoon. Tästä syystä mitään GNU Make-johdannaista (kuten GNU Automake) ei kannata käyttää missään uusissa ohjelmistoprojekteissa, jotka eivät ole tämän tyyppisiä triviaaleja ohjelmia – niiden kanssa törmätään jatkuvasti sellaisiin ongelmiin, että aikaa palaa huomattavasti kun apuohjelmiston puutteita joudutaan kiertämään. Parempi vaihtoehto olisi esimerkiksi *CMake*, mutta sen käyttämistä varten tarvittaisiin jo oma oppaansa, mikä tässä yhteydessä ei ole suotavaa.

Mikäli käännösympäristö on kunnossa, ohjelman kääntäminen tapahtuu yksinkertaisesta komentamalla ”make”. Tämän jälkeen muodostuu ajettava binääriohjelma ”dt3_opencl”.

---

1 Kääntäjäpaketin nimi on ”mingw64-x86_64-gcc-g++”

2 Esimerkiksi Cygwin-ympäristö muuttaa osiotunnukset (*drive letter*) Windows-tyylisestä ”X:\” -merkinnästä enemmän Unix-henkiseen ”/cygdrive/x/” -tyyliin.

Esimerkkiohjelman pääpiirteet

Esimerkkiohjelma muodostuu muutamasta pääkomponentista. Vaikka ne kaikki tarvitaankiin kokonaisuuteen, niistä vain yhden sisäistäminen on suunnittelun kannalta olennaista, mikäli kohdeympäristöön on saatavilla moderni C++-kääntäjä.

Sillä OpenCL-rajapinta on C-kielinen, joutuu ohjelmoija käyttämään paljon aikaa niin muistin varaamiseen, kuin ennen kaikkea sen vapauttamiseen. Tämä vie paljon ajatustyötä ja on todistetusti riskialtista: tapahtuu puskurin yli- ja alivuotoja, tai muistia jää kokonaan vapauttamatta mikä on etenkin rajoitetuissa ympäristöissä tuhoisaa, sillä ongelman havaitseminen voi olla vaikeaa esimerkiksi pelkästään sen takia, että käytössä oleva muisti ei täyty heti vaan viikkojen tai kuukausien kuluessa. C++-tarjoaa kuitenkin tämän välttämiseksi oivat mahdollisuudet, sillä sen lisäksi, että muuttujalle voidaan luoda näkyvyysalue (*scope*), näkyvyysalueen molemmat reunat saadaan kiinni: luomalla raa'an tietomuuttujan (*POD, plain old data*) ympärille kehysluokka (*wrapper*) voidaan hallita niin muuttujaan luontiin, kuin tuhoamiseen liittyvät tilanteet rakentajien (*constructor*) ja hajoittimien (*destructor*) avulla.

Tämän lisäksi virhetilanteiden havannoiminen voidaan automatisoida hyvin pitkälti, tarvittaessa jopa täysin kokonaan. OpenCL-rajapinta käyttää jonkin verran ristiin C-ohjelmoinnin tapoja palauttaa funktiosta sekä varsinainen paluuarvo, kuin tieto funktiokutsun onnistumisesta. Joidenkin rajapintakutsuen yhteydessä (esimerkiksi tietokyselyt) paluuarvo on suoraan tieto siitä, että onnistuiko operaatio ja funktiolle syötettävä osoittimen osoittimen (*pointer to pointer*) sekä puskurin koon perusteella täytetään jokin haluttu puskuri. Paluuarvon perusteella voidaan päätellä, onnistuiko funktiokutsu eli sisältääkö puskuri halutun paluutiedon. Toisien rajapintakutsujen (esimerkiksi objektien luomien) tapauksessa paluuarvo on puolestaa halutun objektin instanssi (tai siis osoitin instanssiin) ja funktiolle on mahdollista syöttää osoitin muuttujaan, johon tieto funktion onnistumisesta tallennetaan. Paluuarvon perusteella voidaan kyllä suoraan päätellä, onnistuiko instanssin luominen, siis funktiokutsu. Mikäli osoitin on arvoltaan *0*, eli ns. nollaosoitin (*null pointer*), kutsu epäonnistui. Tämä ei kuitenkaan kerro sitä, miksi kutsu epäonnistui ja täten jättää niin ohjelmoijan kuin käyttäjänkin epätietoon. Jotta tieto mahdollisesta epäonnistumisesta saataisiin, joutuu C-ohjelmoija ensin määrittelmään muuttujan, johon virhekoodi tallennetaan ja tämän jälkeen syöttämään funktiolle em. muuttujaan osoittimen ja paluuarvon ollessa nolla tulostamaan virhekoodin perusteella virheen. Mantran jatkuva toistaminen johtaa kuitenkin helposti inhimillisiin virheisiin tai tilanteeseen, jossa on kiireellisesti testattava uutta tilannetta. Tällöin ohjelmoijan työajan ollessa rajallinen puuttuva aika saadaan jättämällä virheen tulostamiseen tai käsittelyyn kuluva työavaihe kokonaan pois – tämä on ohjelmiston kannalta huono asia, mutta valittava tosia reaalimaailman paineessa.

OpenCL-funktiokutsun kehysluokka

Luomalla funktiokutsun ympärille kehysluokka[3] (*wrapper*, eli pikemminkin kääre), voidaan molemmat edellä kuvatut tavat välittää virhekoodi käsitellä automaattisesti. Tämä on ”px_wrapcl” komponentin päätehtävä. Samalla voidaan välttää etenkin Windows-ympäristössä käännöksen linkitysvaiheessa tarvittavien kääntäjäspesifisten kirjastotiedostojen tarve. OpenCL-rajapinnan toteutus on käyttöjärjestelmästä riippumatta sijoitettu dynaamisesti ladattavaan kirjastotiedostoon[4]. Tyypillisesti käännösvaiheessa kääntäjälle syötetään em. tiedostosta muodostettu erillinen tiedosto, jonka perusteella kääntäjä automaattisesti muodostaa tarvittavan ohjelmakoodin varsinaisen kirjastotiedoston lataamiseen sekä muuntaa funktiokutsut viittaamaan kirjastoon. Tätä tiedostoa ei kuitenkaan kukaan edellä mainituista valmistajista tarjoa Windows-käyttöjestelmällä kuin MSVC-kääntäjälle, joka ei siis sovellu esimerkkiohjelman kääntämiseen, joten suora linkittäminen dynaamisesesti ladattavaan kirjastoon ei täten onnistu. Tämä kyseinen tiedosto voitaisiin luoda erillisellä apuohjelmalla, mutta apuohjelman asentaminen ja Makefile-tiedoston muokkaaminen tälle tuottaa omat haasteensa. Lisäksi, polku kirjastoihin jouduttaisiin myöskin asettamaan käsin, mikä entisestään lisää monimutkaisuutta. Dynaaminen kirjasto voidaan kuitenkin ladata ohjelmasta käsin käyttäen siihen soveltuvaa käyttöjärjestelmän tarjoamaa rajapintaa. Tässä tapauksessa dynaaminen kirjasto asennetaan automaattisesti sellaiseen järjestelmähakemistoon, josta se löytyy pelkän nimen perusteella, mikä entisestään helpottaa toimintaa. Kun kirjasto on ladattu, sen sisältä voidaan hakea funktiota niiden nimillä. Mikäli funktio löytyy kirjastosta, sille palautuu funktio-osoite joka voidaan suoraan muuntaa funktiosoittimeksi kun tiedetään funktion paluuarvo, parametrityypit sekä kutsutyyppi (*calling convention*).

Edellä mainittu tapahtuu kaksiportaisesti: *px_wrapcl.hpp*-tiedostossa `__PX_APIWRAP__DECL_` makro esittelee (*declare*) kehysluokan vastaavalle OpenCL-rajapinnan funktiokutsulle. `extern`-määrite kertoo, että muuttuja (tässä tapauksessa `__wrapcl_type__CL_API_CALL_` kehysluokka) on julkinen ja linkkeri muodostaa sille käännösyksikön ulkopuolelle näkyvän tunnisteen. Sillä funktio-osoittimen tyyppi riippuu sekä funktion paluuarvosta että sille syötettävistä argumenteistä, on kehysluokka kaksi malliparametriä (*template*) `_Rc` (viittaa paluuarvoon) sekä `_Ar…` (viittaa funktion parameterihin). Näistä jälkimmäinen on C++11:n mukana tullut uusi ominaisuus, muuttuvamääräinen malliparametri (*variadic template*), joka määritellään ”...” jälkiliitteellä. Se tarkoittaa sitä, että malliparametri voi esiintyä *0–N* kertaa. Näin yhden kehysluokan alle saadaan kaikki funktio-osoittimet mille tahansa paluuarvolle ja mille tahansa parametreille. OpenCL-rajapinta määrittelee kutsutyypin aina makrossa `CL_API_CALL`, joten oikean kutsutyypin valitseminen on helppoa.

Makrossa esiintyvä `decltype`-pseudofunktio palauttaa funktion sisällä olevan lausekkeen paluuarvon. Ensimmäisessä makrokutsussa se on esimerkiksi `decltype(wrapcl_(&::clGetPlatformID))`. `wrapcl_` on puolestaan vastaavilla `_Rc` ja `_Ar…` malliparametreillä oleva apufunktio, jolle syötetään mahdollinen funktion osoite ja nimi. C++-kääntäjä pyrkii aina täydentämään malliparametrit funktiolle syötettyjen argumenttien perusteella, jolloin tässä tapauksessa sekä `_Rc` että `_Ar…` täydentyvät automaattisesti kääntäjän toimesta. Mikäli syötetty funktion osoite on nolla, yrittää funktio hakea (Windows-versioissa) dynaamisesti

3 C++-ohjelmointikielessä sekä *class* että *struct* tarkoittavat molemmat samaa tietorakennetta, eli luokkaa. Ainoa ero on, että oletuksena *class* määriteellä näkyvyysmäärite (*access modifier*) on *private*, kun struct määritteellä se on *public*.

4 *Oletuksena .so* (Linux) tai .dll (Windows) tiedostopääte.

ladattavasta OpenCL-kirjastosta nimen perusteella ko. funktiota. Nyt decltype-pseudofunktio aiheuttaa sen, että wrapcl_-funktiokutsua ei oikeasti tehdä ja pseudofunktio palauttaa funktiokutsun paluuarvon. Tässä tapauksessa se on __wrapcl_type__CL_API_CALL_ kehysluokka, jolle on täydenetty malliparametrit esimerkkissä ilmentyvän OpenCL-funktion clGetPlatformId mukaan. _Rc täydentyy paluuarvon tyypiksi joka on cl_int, ja _Ar… puolestaan tyypeiksi cl_uint, cl_platform_id* ja cl_uint*. Kehysluokka on nyt täydellisesti määritelty[5] ja voidaan siis esitellä linkattavassa muodossa. Esitelty muuttuja nimetään siten, että rajapinnassa esiintyvään *cl*-etuliitteeseen lisätään eteen *wrap*, eli kehysluokka clGetPlatformID funktiolle löytyy wrapclGetPlatformID muuttujasta. Muuttujaa voidaan kutsua kuin mitä tahansa funktiota, sillä kehysluokka määrittelee funktiokutsuoperaattorin (*function call operator*).

Edellä siis ainoastaan esiteltiin muuttujalle kehysluokan tyyppi. Tämän lisäksi *px_wrapcl.cpp*-lähdekooditiedossa täytyy muuttuja myös varsinaisesti määritellä. Lähdekooditiedostosta löytyvä __PX_APIWRAP__DEFS_ makro eroaa edellisestä siten, että muuttujan arvoksi tulee wrapcl_ funktiokutsusta palautuva arvo. Käyttöjärjestelmästä riippuen, funktiolle syötetään ensin joko funktion todellinen (linkkerin täydentämä) osoitin (__WIN32__ makro määrittelemättä, eli ei Windows) tai nolla (Windows), joka aiheuttaa sen, että wrapcl_ funktio yrittää hakea funktiota sen nimellä dynaamisesti ladattavasta kirjastosta. Hakemista varten nimen täytyy olla merkkijonomuodossa. Makrossa tämä tapahtuu lausekkeessa ”cl” # _name. Lausekkeen suoritus on seuraavaa: C-esikäsittelijä muuttaa ristikkomerkin oikealla puolella esiintyvän termin merkkijonoksi. Edelleen, ensimmäisessä tapauksessa se on esimerkiksi GetPlatformID. C-kielessä kaksi kiinteää merkkijonoa (*constant character string*) voidaan liittää toisiinsa ilman eri operaattoria. Näin siis kääntäjä muodostaa lausekkeesta ”cl” ”GetPlatformID” yhden merkkijonon, joka on ”clGetPlatformID”. Tässä tapauksessa lähdekooditiedossa ei ole juuri muita määrittelyjä, sillä malliluokkien ja -funktioiden tapauksessa sekä esittely että määritely voidaan tehdä yhdellä kertaa. Lähdekooditiedossa tapahtuva määrittely käyttää siis vastaavasta *.hpp*-tiedostosta löytyvää wrapcl_ funktiota, jonka toiminta on kuvatta yllä. Lopusta löytyvä { mf, fn } lause on myös uusi C++11-ominaisuus, joka rakentaa (*construct*) eli toisin sanoen muodostaa paluuarvossa määritelyn tietotyypin. Tässä tapauksessa se on siis kehysluokka. Muodustus tapahtuu siis nyt siten, että luokan instanssilla syötetään argumenteiksi mf ja fn, joista ensimmäinen on funktion osoite (joka voi myös nolla) ja toinen funktion nimi. Tässä tapauksessa nimi on enää viitteellinen ja sitä käytetään ainoastaan lokimerkintöihin ym. toimiin.

Ilman ohjelmointikokemusta __wrapcl_type__CL_API_CALL_ kehysluokan funktiokutsuoperaattori eli operator() voi näyttää erittäin epäselkeältä. Kutsuoperaattorin parametrit on määritelly malliparametrin _CompatAr... avulla. Nytkin kyseesssä on muuttavamääreinen malliparametri. Kutsuoperaattorissa voitaisiin käyttää suoraan luokan malliparametria _Ar..., mutta tämä aiheuttaisi lisätoimenpiteitä ohjelmoijalta: kutsuoperaattorille pitäisi syötää argumentit täysin samoilla tyypeillä, kuin mitä ne ovat vastaavasta kutsuttavassa OpenCL-funktiossa. Tämä johtaa kuitenkin siihen, että C++-kielen ominaisuuksia ei voitaisi täysimääräisesti hyödyntää. Myöhemmin esimerkkiohjelmassa dynaamisesti varattavaa tietoa tallennetaan mm. shared_ptr ja unique_ptr luokkien avulla, jotka ovat ns. älykkäitä osoittimia. Näiden hallitsemia osoittimia ei voi kuitenkaan automaattisesti muuttaa takaisin C-tyylisiksi osoittimiksi (tämä olisi teknisesti mahdollista, mutta ominaisuus on tarkoituksella

5 Luokka on siis clGetPlatformID funktiolle __wrapcl_type__CL_API_CALL_< cl_int, cl_uint, cl_platform_id*, cl_uint* >, tai oikeastaan mille tahansa funktiolle, jolla on em. paluuarvo ja parametrityypit.

ohjelmointivirheiden ehkäisemiseksi estetty). Muutos voidaan toteuttaa kuitenkin erillisellä apufunktiolla, mutta tämän automatisoimiseksi kutsuoperaattorin on sallittava *yhteensopiva* parametrityyppi, joka ei tietenkään ole täysin sama kuin OpenCL-funktiolle syötettävän argumentin lopullinen parametrityyppi. Tästä seuraa myös jonkin verran muita vaatimuksia, jotka on otettava huomioon apufunktiota (tai apufunktioita) suunniteltaessa. Funktiokutsuoperaattorin tapauksessa yhteensopivia argumentteja ei kannata syöttää kopioina, sillä kopiointi on näennäisesti hidasta. Toisaalta, parametrityyppi ei voi olla viittaus (*reference*), sillä tässä tämä estää väliaikaisten instanssien syöttämisen (esimerkiksi apufunktio joka palauttaa kääntäjän toimesta pinosta varatun luokan instanssin). Tätä ominaisuutta tarvitaan kuitenkin, mikäli automaattinen virheiden havaitseminen halutaan toteuttaa. C++11 tarjoaa kuitenkin tämän, todellisuudessa hyvin tyypillisen vanhan sukupolven C++-ongelman, ratkaisemiseksi uuden viittaustyypin. Oikean arvon viittauksella[6] (*rvalue reference*) mahdollinen instanssi luodaan vasta kutsuttavan funktion sisällä, eikä ennen kuin funktiota kutsutaan (kuten esimerkiksi kopioitaessa menetellään). Tämäkään ei ole täysin ongelmatonta, sillä tästä seuraa nyt uusi ongelma: esimerkiksi taulukot syötetään tyypillisesti, kuten on tehty myös OpenCL-rajapinnassa, C-kielessä osoittimena taulukon ensimmäiseen elementtiin, jolloin vältetään taulukon kopioiminen. Tyyppimuunnos taulukkotyypistä osoittimeen on automaattinen, joten tapa on kaikilta osin luonteva. Tässä tapauksessa (oikean arvon viittaus) tyyppimunnosta ei voi kuitenkaan enää automaattisesti tehdä, vaan se on tehtävä eksplisiittisesti! Tästä syystä apufunktiohin on lisättävä myös erikoistettu versio, joka tyyppimuuntaa taulukon[7] osoittimeksi.

Kutsuoperaattorissa kutsutaan edellä kuvatun tyylistä apufunktiota `unwrap`. Todellisuudessa em. apufunktio kutsuu totetusta `__unwrap` nimiavaruuden `detail` alta – kyseessä on käytännössä täysin ns. artistin näkemys, eli `unwrap` voitaisiin haluttaessa suoraan erikoistaa (*spealize)* kaikille operaattorille syötetyille argumenteille. Kutsuun liitetään kiinteästi mukaan osoitin kehysluokan instanssiin sekä lopullinen haluttu, varsinaiselle OpenCL-funktiolle syötettävä parameterityyppi `_Ar`. Mikäli kutsuoperaattorille syötetään väärä määrä argumenttejä, ohjelma ei käänny sillä `_Ar` ja `_CompatAr` ei vastaa toisiaan sisältämiensä parametrien määrän puolesta. Tämä on haluttu tilanne.

Edellä todettujen älykkäiden osoittimien lisäksi `unwrap` apufunktiolla hoidetaan OpenCL-funktion argumentiksi syötettävän virhekoodin osoittimen luominen sekä nollaosoittimet. Nollaosoittimien erillinen käsittely on myöskin C++11-ominaisuus, josta on suuresti hyötyä etenkin C-rajapintoja hyödynnettäessä. C-kielessä nollaosoitinta kuvaava makro `NULL` on yleisellä tasolla korkeintaan toteuttu yksinkertaisesti tyyppimuunnoksella `void` tietotyyppiin, eli `(void*) 0`. Jo C++98-kielestä lähtien edes tyyppimuunnosta ei ole sallittu, vaan `NULL` *on lauseke, joka saa arvon nolla*. Riippuen kääntäjästä tai käännösasetuksista, osoitinmuunnokset tavumäärältään vastaaviin kokonaislukutyyppeihin ovat sallittuja ja tapahtuvat automaattisesti. Tällöin vaarana on, että argumenttejä syötettäessä lukuarvo *0* ja osoitin osoitteeseen *0* menevät ohjelmoijan virheestä johtuen sekaisin: esimerkiksi lukuarvon (vrt. puskurin koko) ja osoittimen (vrt. puskurin osoite) argumentit syötetään väärässä järjestyksessä. C++11:ssa on kuitenkin käytössä erillinen nollaosoitin `nullptr`, ja tämän tietotyyppi `nullptr_t`. Tämä tietotyyppi *ei* ole tyyppimuunnettavissa osoittimeksi ja sen avulla funktio voidaan erikoistaa nollaosoitinta varten. Kehysluokkaa käytettäessä kaikki

6 Vakiintunutta suomenkielistä termiä ei ole, jotkin lähteet puhuvat ”tuplaviittauksesta”, joka juontuu suoraan viittauksen muodosta `&&` (vrt. normaali viittaus, joka on `&`).

7 Valittavasti edes C++17 ei mahdollista kääntäjän varaaminen muuttuvakokoisten taulukkojen (*VLA*, *variable length array*) syöttämistä minkään tyypin viittauksina, joten näiden kohdalla ohjelmoijan on tehtävä käsin eksplisiittinen tyyppimuunnos.

nollaosoittimet, joiden arvo ei tule jostakin osoitinmuuttujasta, on syötettävä muodossa nullptr, muutoin ohjelma ei käänny. nullptr_t tietotyypin perusteella on erikoistettu apufunktio, joka palauttaa osoittimen arvolla *0*.

Osoittimen perusteella välitettävää virhekoodia varten tarvitaan apuluokka __wrapcl_check_err. Luokka on joiltakin osin ylisuunniteltu (*over engineering*). Tiedossa on, että paluuarvo on kokonaisluku ja vieläpä pieni sellainen (esitettävissä yhdellä tavulla). Käytännössä se on tyypiltään aina cl_int. Apuluokka on kuitenkin suunniteltu niin, että virhekoodin tyyppi voi olla mikä tahansa ja sille varataan muisti erillisen puskuriluokan shared_buf avulla. Näin apuluokalla ei ole malliparametrejä, joten se voidaan luoda ilman tietoa palautuvan virhekoodin tietotyypistä. Apuluokkaa ei koskaan luoda suoraan, vaan vasta erikoistetun __unwrap apufunktion sisällä. __unwrap apufunktio erikoistetaan aputyypille __wrapcl_check_err__here, joka on tyhjä luokka. Kaikki C++-kääntäjät osaavat optimoida tyhjän luokan avulla tehtävät operaatiot, joten tällaisen tietotyypin kuljettaminen funktiokutsusta toiseen on käytännössä ilmaista. Aputyypillä on kuitenkin tarkoituksella pitkä ja hankala nimi, jotta se ei sekoitu muihin ohjelman tarvitsemiin tyyppeihin. Aputyypin luomista varten on triviaali funktio wrapcl_err, jota tulee käyttää kehysluokkaa käytettäessä. Sillä väliaikaisen paluuarvon palauttaminen (*RVA*, *return value optimization*) on myös optimoitu kaikilla C++-kääntäjillä, on jälleen kerran tämän tyyppinen luokan muodostaminen normaali C++-käytäntö. Seuraa kuitenkin ongelma: muodostettavan __wrapcl_check_err instanssi on talletettava johonkin niin, että se on olemassa OpenCL-funktiokutsun ajan ja mielellään myös sen jälkeen. Sillä apuluokka on malliparametriton, sille voidaan varata tila kehysluokan malliparametrittomassa yläluokassa __wrapcl. Muutoin seuraa tyypillinen muna-kana -ongelma, jonka poistaminen on teoriassa mahdollista, mutta käytännössä ei ole vaivan arvoista: __unwrap erikoistamiset voitaisiin esitellä ennen funktiokutsuoperaattoria ja määritellä vasta sen jälkeen. Tällöin periaatteessa kehysluokan instanssi voitaan syöttää myös malliparametrin avulla __unwrap apufunktiolle. Tällöin tarvitaan kuitenkin malliparametereja käytännössä jokaisen apufunktion kohdalla, joka on työlästä ja toisaalta vaatii paljon kääntäjältä. Se ei siis ole vaivan arvoista.

Kun erikoistettu __unwrap apufunktio luo __wrapcl_check_err apuluokan, on apufunktiolla tiedossa myös todellinen virhekoodin tietotyyppi _Ar malliparametrissä. Tämän perusteella virhekoodille voidaan sekä varata tarvittava tila, että muodostaa sopiva osoitin tähän. Tietotyypin piilottamiseen käytetään tyypillistä C++-metodia: luodaan funktiosoitin, joka ei sisällä malliparametreja ja tämän arvoksi annetaan malliparametrisoitu (*templated*) funktiokutsu, jonka kaikki malliparametrit on määritelty. Sillä virhekoodin tarkastamiseksi riittää totuusarvo, onnistuu tämä helposti. Varsinaisen virhekoodin tulostamisessa on esimerkkiohjelmassa ajan säästämiseksi hieman oikaistu. Virhekoodin tyyppi oletaan aina *cl_int* tyyppiseksi. Mikäli näin ole, ei virhekoodia tulosteta. Ohjelma ei kuitenkaan kaadu, sillä todellisen puskuriin talletetun tiedon tietotyypistä pitää puskuriluokka huolen.

Esimerkkiohjelmassa, tai kehysluokassa, virheen tapahtuminen ei aiheuta välittömiä toimenpiteitä, vaan jokaisen funktiokutsun kohdalla ohjelmoija voi päättää mitä tehdään. Mikäli virheen tapahtuessa ohjelman suoritus haluttaisiin aina lopettaa, voidaan virheen palauttamisen sijasta käyttää poikkeuksia (*exceptions*). Lisäksi kehysluokkaan voidaan toteuttaa toiminto, jossa kullekin funktiokutsulle voidaan määritellä lista virheistä, jotka ovat odotettavissa. Tällä tavoin virhetilanteita voidaan suodattaa. Sillä kehysluokan toteutus on jo tällä tasolla monimutkainen, näitä edellä mainittuja toimintoja ei ole kuitenkaan toteutettu. Tästä syystä esimerkiksi clGetDeviceID

funktion yhteydessä näkyy pääohjelmassa virheilmoitus CL_DEVICE_NOT_FOUND, mikäli yhtään haluttua OpenCL-laitetta ei ole kyseiseltä alustalta saatavilla. Tämä ei kuitenkaan ole virhe virheen todellisessa merkityksessä: OpenCL-rajapinta on vain syytä tai toisesta suunniteltu niin, että mikäli OpenCL-laitteita ei ole käytettävissä, palautuu lukumäärän *0* sijasta kyseinen virheilmoitus. Kyseessä on hieman ironinen tilanne, sillä käytettävissä olevien laitteiden selvittämiseen ei ole käytettävissä muita funktioita.

Tekstivuo ja tekstimuunnokset

Esimerkkiohjelma tulostaa tietoa ohjelman kulusta tekstimuodossa terminaaliin. Tekstivuota ei kuitenkaan käsitellä C-kielisessä ohjelmissa monesti käytössä olevalla printf-funktiolla tai sen johdannaisilla. Tämä johtuu siitä, että vaikka aluksi printf-tyylinen tulostusfunktio voi vaikuttaa helpolta sen käyttö muuttuu kuitenkin erittäin nopeasti hankalaksi ja kömpelöksi. C++-ohjelmoinnissa erittäin tärkeässä osassa on funktioiden ylikuormittaminen[8] (*overload*), erikoistaminen sekä ohjelman koodipolun ohjaaminen jo käännösvaiheessa erilaisten tietotyyppinen ja malliparametrien avulla. Sillä printf-komennot suorittavat muotoilut (teksti, desimaaliarvo, heksa-arvo ym.) vasta ajon aikana annetun merkkijonoformaatin perusteella, ei käännösvaiheessa tapahtuvaa muotoilua tietotyypin perusteella voi tehdä. Tämä rajoittaa merkittävästi sitä, mitä ohjelmoija voi tehdä ja eritoten muuttaa ohjelman elinkaaren aikana.

”px_strm”, ”px_strm_adapter” sekä ”px_enum” komponentit keskittyvät hyvin voimaakkaasti tekstin muotoiluun. Näistä jälkimmäisen avulla luetellut tyypit (*enum* eli *enumeration type*) korvataan vastaavilla tekstilausekkeilla. Kun luetellulle tyypille määritellään kutakin arvoa vastaavat tekstit, ei terminaaliin tulostettaessa tarvitse käyttää erillistä apufunktiota numeroarvon muuttamiseen tekstimuotoon. Lisäksi, voidaan määritellä edustaako lueteltu tyyppi bittimaskia vaiko uniikkia arvoa. Sillä tekstiä tulostettaessa molempia tyyppejä täytyy käsitellä eri tavoin, on erittäin hyödyllistä, että molemmat käsittelytavat tarvitsee toteuttaa vain kerran. Lueteltujen tyyppien kanssa on tämän lisäksi hyödynnetty C++11:n myötä käytettävissä olevaa, erillistä lueteltua luokkaa eli enum class määritettä. Sen lisäksi, että luetellulle luokalle voidaan määritellä todellinen (kokonaisluku)tietotyyppi, sen jäseniin tulee aina viitata luetellun luokan nimellä. Näin kahdessa tai useammassa eri luetellussa luokassa voi olla samannimisiä jäseniä. Perinteisen luetellun tyypin kohdalla tämä ei ole mahdollista ilman eri toimenpiteitä. Valitettavasti OpenCL-rajapinnassa ei todennäköisestä juurikin tästä syystä käytetä lueteltuja tyyppejä, vaan kiinteät arvot määritellään perinteisillä makroilla. Tämä on huono tapa, sillä makroilla ei ole nimiavaruutta edes C++-kääntäjää käytettäessä ja arvojen perusteella ei voida ylikuormittaa funktioita. Arvojen muuttamiseksi vastaaviksi luetelluiksi tyypeiksi on näytetty kaksi eri tapaa. Joko ne voidaan yksitellen määritellä, kuten on tehty pääohjelmassa eli ”px_main” komponentissa. Tämän lisäksi ”px_wrapcl_errc” komponentissa on luotu (tässä tapauksessa virhekoodille) eräänlainen pseudotyyppi, jossa ei varsinaisesti luetella arvoja vaan sen perusteella saadaan ylikuormitus tehtyä. Vastaavasti makron avulla yhdistetään kunkin makron arvo vastaavaan makron nimeen, jolloin valmiin ohjelmakoodin kopioimiseen ja muokkaamiseen ei kulu merkittävästi aikaa. Tässä tapauksessa virhekoodin makrojen nimet ovat itseselitteisiä, joten sinällään niitä ei ole edes tarvetta avata enempää.

Erittäin yksinkertainen ”px_strm_adapter” komponentti mahdollistaa nopean tavan luoda (väliaikaisia) merkkijonoja. Toiminallisuutta tarvitaan esimerkiksi silloin, kuin halutaan liittää poikkeukseen (*exception*) tieto siitä, mitä ja mahdollisesti missä on tapahtunut. Mikäli käytettäisiin perinteistä printf-tulostustapaa, tämän tyyppisiä tilanteita varten jouduttaisiin varamaan määrätyn mittainen puskuri yhteiseltä näkyvyysalueella tai staattisesti. Tekstitulostuksessa tämän puskurin

8 Kun samalla nimellä luodaan useampi funktio, joilla ei ole malliparametreja, mutta joilla kullakin on eri määrä, eri tietotyypin parametrejä (sekä mahdollisesti eri paluuarvo, *ei* kuitenkaan *pelkästään* eri paluuarvo) puhutaan ylikuormittamisesta. Sen sijaan, jos tietylle tai tietyille malliparametreille (ei välttämättä kaikille, tällöin tosin kyseessä on *partial specialization* eli osittain erikoistaminen) määrätään poikkeava toteutus, puhutaan erikoistamisesta. Termit menevät erittäin helposti sekaisin, mutta ovat kuitenkin siis eri asia omine rajoituksineen.

koko joudutaan kuitenkin usein arvamaan, sekä pitämään huoli siitä, ettei puskuria ylitetä. Sillä ohjelmoija joutuu tekemään tämän kaiken joka kerta erikseen, on jälleen kerran vaarana, että aikataulusta nipistetään juuri nämä tarkastukset poistamalla. Sen sijaan käytettäessä C++-standardikirjaston string ja stringstream luokkia tapahtuu muistin varaus automaattisesti, eikä ylivuotoja pääse syntymään. Lisäksi kaikki ylikuormitetut tekstimuunnokset ovat automaattisesti käytettävissä.

”px_strm” komponentti sisältää esimerkkejä tyypillisistä tekstimuunnoksista. Desimaali- ja heksalukumuunnokset löytyvät kyllä suoraan standardikirjaston dec ja hex manipulaattorien kautta, mutta näitä (sekä muita vastaavia) käytettäessä törmätään helposti tyyliongelmiin: nämä manipulaattorin muuttavat tekstivuon lukutyylin pysyvästi, eli esimerkiksi lukujen kohdalla jokaisen desimaali- ja heksaluvun kohdalla tulee manipulaattorilla varmistaa, että lukujärjestelmän muunnos on haluttu. Tieto siitä, mitkä manipulaattorit ovat voimassa saadaan kuitenkin ohjelmallisesti selville. Siirtämällä tekstimuotoiluun liittyvä toiminnallisuus __strm__forward_ apuluokan alle, voidaan standardimuotoilut tallentaa ennen varsinaisen tyylitoteutuksen kutsumista ja palauttaa kutsun jälkeen entiselleen. Toinen esimerkki tekstimuunnoksesta löytyy ”px_main” pääkomponentista. device_sizes apuluokan tarkoitus on toimia kehyksenä OpenCL-rajapinnan arvolle, joka ilmoittaa laskettavan alueen 3-ulotteisen koon. Koko palautuu rajapinnalta kolmen elementin taulukkona ja apuluokan ainoa jäsenumuuttuja on toteutettu tämän mukaan. Apuluokan jäseneksi on kuitenkin määritelty ylikuormitettu tekstivuo-operaattori[9], jota kutsutaan automaattisesti, kun jotain apuluokan tyyppistä muuttujaa ollaan tulostamassa. Näin tulostaessa, menipä syöte joko terminaaliin tai tiedostoon, ei tarvitse erikseen kutsua mitään erillistä apufunktiota. Lisäksi, sillä viittaus varsinaiseen kohteena olevaan tekstivuohon kulkee operaattorin argumenttina, ei (yleensä) tarvita väliaikaisia apupuskureita. On huomattava, että tällä tavoin suoraan luokan sisällä määriteltävälle tekstivuo-operaattorille on annettava friend tarkenne. Tämä aiheuttaa sen, että tekstivuo-operaattori[10] ei ole enää luokan varsinainen jäsen, vaan itsenäinen funktio, jolla on pääsy luokan kaikkiin jäseniin. Käytetty tyyli on kuitenkin ohjelmointimielessä hyödyllinen, sillä yhdellä kertaa voidaan niin esitellä kuin määritellä luokan ulkopuolinen funktio, joka voi käyttää luokan kaikkia jäseniä toteutuksessaan.

Edellä kuvatussa device_sizes esimerkissä on hyvä sisäistää varsinaisen tekstivuo-operaattorin sekä tältä näyttävän, ehkä osana samassa ketjussa olevan bittisiirto-operaattorin ero. Kun standardikirjaston tekstivuoluokkien avulla halutaan tulostaa syöte, on tekstivuo-operaattorin oltava *itsenäinen funktio*. Tämän tunnistaa siitä, että operaattorin toisena argumenttinä on käsiteltävä tieto; tyypillisesti tämä välitetään joko viittauksen tai osoittimen välityksellä turhan kopioimisen välttämiseksi. Sen sijaan esimerkiksi ”px_strm_adapter” komponentissa on käytetty __strm_adapter apuluokan *jäsenfunktioksi* määriteltyä bittisiirto-operaattori. Kyseistä operaattoria kutsutaan ainoastaan silloin, kun bittisiirto-operaattorin (ks. alaviite 9) oikealla puolella on kyseisen luokan instanssi, ei jokin standardikirjaston tekstivuoluokka, eli *basic_ios* pohjainen toteutus! Lopullinen tekstijono *string* muodossa __strm_adapter apuluokasta saadaan jäsenfunktioksi määritellyn tyyppimuunnosoperaattorin avulla. Tyyppimuunnosta osoittimeksi merkkijonoon (tietotyyppi *[const]* char*) ei määritellä sen vuoksi, että varsinaisen merkkijonon olemassaoloa ei

9 Todellisuudessa kyseessä on bittisiirto-operaattori, mutta on tässä yhteydessä kyseisen termin käyttäminen on harhaanjohtavaa varsinaisen toiminnan kannalta.

10 On huomattava, että friend tarkenne ei missään nimessä koske ainoastaan tekstivuota, vaan on on käytettävissä kaikille ulkopuolisille funktioille ja luokille, joille halutaan antaa pääsy luokan suojattuihin (*protected*) ja yksityisiin (*private*) jäseniin.

tällöin voida taata. string luokkaa käytettäessä sen sisältämän merkkijonon kopioiminen ja tarvittavan muistin varaaminen tapahtuu kuitenkin automaattisesti, toisin kuin osoittimilla. Tämän tyyppisissä ohjelmissa ei saavuteta minkäänlaista nopeusetua käyttämällä C-tyylistä merkkijonokäsittelyä.

Puskuriluokat

Edellä kuvattujen aputoimintojen lisäksi tarvitaan myös dynaamisesti varattavia puskureita varten omat komponenttinsa. ”px_shared_buf” komponentti toteuttaa 1-ulotteisen, dynaamisesti joko tietotyypin tai annetun puskurikoon mukaan varattavan tavupuskurin. Tavupuskurin ilmentämää shared_buf luokkaa voidaan kopioida ilman mittavia nopeudellisia vaikutuksia. Tulee kuitenkin huomata, että luokka on toteutettu siten, että kaikki luokan instanssin kopiot jakavat saman puskurin: mikäli puskuriin tehdään muutoksia, tulevat näkyvät muutokset kaikissa luokan instanssin kopioissa. Muistin hallintaan käytetään älykästä shared_ptr osoitinta. Tämä osointinluokka ei kuitenkaan sisällä tietoa siitä, kuinka laaja muistialue on sen hallinnoiman osoittimen käytössä. Tämä tieto voitaisiin tallettaa esimerkiksi shared_buf luokan jäsenmuuttujaksi, mutta eräs paljon käytetty tapa on hyödyntää mahdollisuutta määrittää älykkäälle osoittimelle oma vapautusluokka (*deleter*). Kun tämän luokan jäsenmuuttujaksi sijoitetaan tieto varatun puskurin koosta, saadaan puskurin koko selville pelkästään älykkään osoittimen avulla. Näin esimerkiksi shared_buf luokkaa muutettaessa (esimerkiksi jos luokkaan lisätään toiminto puskurin kahdentamiseksi, koon kasvattamiseksi ym.) ei tarvitse luokan sisällä huolehtia sekä varsinaisen älykkään osoittimen että puskurin koon kuljettamisesta eri instanssien välillä, vaan älykkään osoittimen oma toteutus huolehtii ensin mainitusta.

shared_buf luokkaan on lisäksi toteutettu yksinkertainen tietotyyppitarkastus. Se käyttää hyväkseen standardikirjaston type_info luokkaa, joka sisältää C++-ohjelmointikielen ajoaikaisen tyyppi-informaation (*RTTI,* r*un-time type information*). Sen lisäksi, ettei luokalla ole esimerkiksi malliparametria joka tulisi tietää, luokan avulla voidaan verrata, ovatko kaksi tyyppiä keskenään samoja (tai asettaa kaksi eri tyyppiä järjestykseen, tätä ominaisuutta voidaan myös hyödyntää joissakin sovellutuksissa). Näin puskuriluokkaan voidaan rakentaa toiminto, joka estää virheelliset tyyppimuunnokset. Näin ohjelmoidessa tehnyt väärät oletukset eivät muodosta joissain tilanteissa vaikeasti havaittaa virhetilannetta, jossa jokin tietotyyppi alustetaan täysin epäyhteensopivalla tiedolla.

Tyypillisesti sekä C- että C++-ohjelmointikielet mielletään vahvasti tyypitetyksi kieliksi, eli kaikilla muuttujilla on jokin konkreettinen tietotyyppi. Sama muuttuja ei voi esimerkiksi olla eri kontekstista riippuen esimerkiksi joko kokonaisluku tai liukuluku. Tietotyypille on voitu sallia, että sama muistialue voi kuulua useammalle eri tietotyypille *union* tietorakenteen avulla. C-kielessä sekä C++11-kieltä edeltäneissä versiossa tässä rakenteessa on ollut se rajoite, että tietorakenteessa määritellyt tietotyypit eivät voi olla muita kuin kielen omia triviaaleja tietotyyppejä (*POD, plain old data*). C++11:n myötä *union* tietorakenteessa myös kompleksiset tyypit ovat sallittuja, mutta tämäkään ei mahdollista täyttä dynaamisuutta tietorakenteen käyttämisessä, sillä sekä tietoa luettaessa että kirjoitettaessa täytyy viitata haluttuun tietotyyppiin rakenteen jäsenmuuttujan avulla. Komponentissa ”px_variant_buf” on toteutettu yksinkertainen muuttuvatyyppinen (ns. *variant*) puskuriluokka, jonka avulla jotkin ennalta määrätyt tietotyypit voidaan dynaamisesti hallita luokan avulla. Sen lisäksi, että tiedon tallentamisen ja lukemisen osalta apuluokka toimii kuin *union* tietorakenne, se sisältää vertailuoperaattorit, joiden avulla luokan sisältämää informaatiota voidaan verrata *ilman tarkkaa tyyppitietoa*. Tämä on äärimmäisen hyödyllistä ja tätä toiminallisuutta myös pääohjelma käyttää hyväkseen. Dynaaminen tietotyypitys täysin kattavalla tasolla on hankalaa, ja vaatii monien asioiden huomioimista. Tämän esimerkkiohjelman tasolla kaikissa tilanteissa toimivan toteutuksen laatiminen ei ole aikataulun puitteissa mahdollista, joten varsinaista totetutusta

ei tässä käsitellä sen tarkemmin – sanottaakoon kuitenkin, että valittu lähetystapa ei todennäköisesti johda hyviin lopputuloksiin, joten kyseessä on enemmänkin esimerkki siitä, mitä polkua ei kannata kulkea. Joka tapauksessa muuttuvatyyppisen puskuriluokan käyttö selkeyttää varsinaista OpenCL-rajapinnan käyttöä huomattavasti. Kuten ”px_main” pääohjelmasta voidaan havaita, OpenCL-laitteiden tiedot saadaan tulostettua sekä tulkittua vaatimusten osalta erittäin vähällä ohjelmakoodilla ilman, että syntyy tarvetta muodostaa käsitellyyn poikkeustilanteita joidenkin tietotyyppien osalta. Tämä vähentää niin inhimillisten ohjelmointivirheiden määrää kuin nopeuttaa ohjelmiston kehitystä, sillä muutosten tekeminen on yksinkertaista. Aikaa ei kulu tarpeettomasti koodin kopioimiseen, kuten on valittavan usein tehty muiden ”C++” OpenCL-esimerkkien kanssa. Vastaavasti, riittää, että kaikkia tietoja koskeva muutos tehdään vain yhteen paikkaan, eikä sitä tarvitse käsin monistaa. Tietokyselyitä tehdään kuitenkin useita kymmeniä kappaleita ja vieläkin enemmän tietoa rajapinnan kautta olisi saatavilla, joten ohjelmointi leikkaa ja liimaa menetelmällä on erittäin raskasta ylläpidollisessa mielessä.

Pääohjelma

Pääkomponetti ”px_main” ohjaa edellä kuvattujen apuluokkien ja -toimintojen avulla varsinaista OpenCL-rajapintaa. Rajapinnan käyttäminen tapahtuu pääpiirteissään siten, että rajapinnan funktioiden avulla haetaan halutun tyyppiset OpenCL-alustat (*platform*). Yhdellä alustalla voi olla yksi tai useampi OpenCL-laite, joka puolestaan voi olla niin suoritin, näytönohjain kuin FPGA-tai ASIC-piirikin. Käytettävistä ja ohjelman kriteerit täyttävistä (esimerkiksi muistin ja tuettuen toimintojen osalta) laitteista valitaan käyttöön yksi tai useampi laite, joille kullekin luodaan oma konteksti (*context*). Kontekstiin on liitettävä komentojono (*command queue*), joka käytännössä tarkoittaa muistiväylää OpenCL-isäntäohjelma ja -ytimen välillä. Kontekstin ohella käännetään varsinainen OpenCL-lähdekoodi ohjelmaksi (*program*), sekä valitaan käännetystä ohjelmasta suoritettava ydin. Suoritettava ydin asetetaan ajettavaksi luotuun komentojonoon, jonka jälkeen yleensä loogisinta on lisätä ajettavaksi pyyntö kopioida tuloset OpenCL-laitteelta takaisin isäntäohjelmaan. Kun sekä suoritus, että kopiointi on suoritettu, on OpenCL-ajo valmis. Sillä rajapinta huolehtii komentojonoon asetettujen tehtävien suorittamisesta, tulee vähintään viimeisen tehtävän kohdalla huolehtia eli käytännössä tulosten kopimisen osalta huolehtia, että isäntäohjelma on synkronoitu rajapintaan: molempien suoritus tapahtuu rinnakkain, eli rajapintakutsu esimerkiksi tehtävän lisäämisen komentojeen osalta voi palautua aikaisemmin, kuin mitä lisätty tehtävä on todellisuudessa suoritettu.

Esimerkkiohjelmassa edellä kuvatut kokonaisuudet on pyritty selkeyden vuoksi niputtamaan omiksi kokonaisuuksikseen. OpenCL-laitteen valinta tapahtuu `prog_global` luokan avulla, `init` jäsenfunktiossa. Luokka sisältää jäsenmuuttujina mm. valitun kontekstin, komentojonon, näytteistäjän (*sampler*), ohjelman sekä ytimen instanssit. OpenCL-rajapinnassa näihin instansseihin viitataan `cl_context`, `cl_command_queue` jne. tietotyypeillä. Tyypit on määritelty vastaavasti määrittelemättömän luokan `_cl_context`, `_cl_command_queue` jne. osoittimiksi. Kyseessä on puhtaasti suunnittelutekninen valinta, sillä täten voidaan vähentää tahatonta paluuarvojen tai argumenttien sekoittumista keskenään, sillä osoittimen kohteena ovat selkeästi eri tietotyypit. Se, ettei tietotyyppejä eli em. luokkia ole määritelty, ei haittaa, sillä osoittimen tavukoko on luokasta riippumatta aina sama. Näin ohjelma kääntyy, linkittyy ja toimii kuten haluttua, ilman että todellisuudessa näitä `_cl_context` ym. (pseudo)luokkia koskaan määritellään missään rajapinnan käyttöjän näkökulmasta.

Sen jälkeen kun OpenCL-instanssit on varattu ja niillä on suoritettu tarvittavat operaatiot, tulee ne myös ennen ohjelman lopetusta vapauttaa. C++-kielen näkökulmasta, mikäli resurssin vapauttaminen täytyy erikseen ohjelmoijan toimesta tehdä käsin, on resurssin käyttö suunniteltu huonosti. Sillä resursseja varataan pitkin ohjelman suoritusta, tulee virhetilanteissa resurssien vapauttamisesta vaikeaa. Tähän liittyy etenkin se, että C++-kielisissä ohjelmissa on mahdollista poikkeuksien käyttö. Käytönnössä poikkeuksia on pakko käyttää virhetilanteissa, jotka ilmenevät esimerkiksi operaattoreissa: näiden kohdalla ei ole mahdollista paluuarvona tai argumenttina välittää tietoa siitä, onnistuko haluttu operaatio vai ei. Tällöin on erittäin hyödyllistä ja ennen kaikkea tärkeää, että resurssille määritellään näkyvyysalue (*scope*), jonka reunalle astuttaessa resurssi varataan ja jolta poistuttaessa resurssi vapautetaan. Kaiken hyvän lisäksi tämä on täysin *RAII*-hengen, eli *resource acquisition is initialization* mukaista: tarvittava resurssin varaataan, kun sitä käyttävän luokan instanssi alustetaan sekä vapautetaan kun instanssikin vapautetaan. Tästä syystä OpenCL-instanssien hallitsemista varten on toteutettu `__prog_pointer` apuluokka, joka

vapauttaa OpenCL-resurssin instanssin määrättyä rajapintafunktiota hyväksikäyttäen, kun apuluokan instanssia ollaan tuhoamassa. Älykkäitä osoitinluokkia ei voida suoraan käyttää, sillä osa OpenCL-rajapintafunktioista tarvitsee osoittimen osoittimeen, eli osoittimelle on varattava muistista tila, johon rajapintafunktiolle syötetään osoitin. Kuten lähdekoodissa on todettu, kun prog_global luokan intanssit sisältävät jäsenmuuttujat määritellään oikeaan järjestykseen, ei luokalle tarvitse luoda erillistä hajoitinta – kääntäjä luodessa sen automaattisesti vapauttaminen tapahtuu kuitenkin oikeassa järjestyksessä. Monimutkaisemmissa versioissa voidaan hyödyntää esimerkiksi standardikirjaston stack säiliöluokkaa, johon instanssit voidaan tallettaa ja myöhemmin poistaa käänteisessä järjestyksessä.

OpenCL-ytimeen vietävä ja sieltä tuotava tieto on välitettävä puskurien avulla. Puskuri voi olla niin tyypillinen yksiulotteinen vektori kuin yksi- tai moniulotteinen kuva. Kuvapuskuria on käytetty esimerkkiohjelmassa, sillä se on tässä tapauksessa loogisin valinta kuvainformaation välittämiseksi. Kuvapuskuria käytettäessä on huomattava, että kroma-, luminanssi- ja värikanavien asetuksissa riippuen puskuria tulee käsitellä OpenCL-ytimessä näiden asetusten kanssa yhteensopivilla funktioilla. Vaikka siis isäntäohjelman puolella jokainen kanava olisikin 8-bittinen kokoisluku, riippuen asetuksista voi varsinainen käsittely ytimessä tapahtuakin liukuluvuilla.

Esimerkkiohjelman suoritus alkaa OpenCL-laitteen valitsemisella. Ohjelma hakee rajapinnan avulla aluksi käytettävissä olevat alustat clGetPlatformID funktiota käyttämällä. Funktiosta käy ilmi tyypillinen tapa välittää tietoa rajapinnalta isäntäohjelmaan. Funktion argumenteiksi syötetään joko isäntäohjelman puskurin koko ja osoitin tai osoitin muuttujaan, johon rajapinta tallettaa tiedon tarvittavasta puskurikoosta. Jälkimmäisen kanssa on oltava kuitenkin tarkkana, sillä riippuen funktiosta puskurikoko ilmoitetaan joko elementtien määränä (kuten tässä) tai tavumääränä. Nämä eivät siis ole yksi ja sama asia. clGetPlatformID funktiota kutsutaan kaksi kertaa. Ensimmäisellä kertaa halutaan tietää, kuinka monta OpenCL-alustaa on käytettävissä. Sillä käytössä on moderni versio C++-ohjelmointikielestä (oikeastaan vähemmänkin moderni riittäisi tähän), voidaan suoraan saadun tiedon perusteella varata pinomuistista tarpeeksi iso taulukko. Tämän lisäksi voidaan olettaa, että palautuvia alustatunnuksia (*platform id*) on käytännössä aina ”vain kourallinen”, jolloin varattava taulukko mahtuu hyvin rajalliseen[11] pinomuistiin (*stack memory*). Toisella kertaa funktiokutsun argumenteiksi syötetään varatun taulukon koko sekä osoitin tähän. Taulukkoa ei tarvitse erikseen vapauttaa, vaan kääntäjä tekee sen automaattisesti.

Kun käytettävissä olevat OpenCL-alustat on haettu, käydään taulukko läpi väliperustaisella (*range based*) *for*-toistorakenteella. Tämä on uusi C++11 ominaisuus, joka helpottaa ohjemointia piilottamalla näennäistä iteroitiin tarvittavaa ohjelmakoodia. Syntaksi on puhtaasti oikopolku iteraattoreilla tehtävään iterointiin, joten se voidaan ylikuormittaa omille tyypeille täysin vapaasti. Tässä tapauksessa ylikuormittanen ei kuitenkaan ole tarpeen, sillä kyseessä on triviaali taulukko. OpenCL-alustan tiedot haetaan vastaavalla tavalla, kuin itse alustojen tunnuksetkin. Ensimmäisellä clGetPlatformInfo funktion kutsukerralla selvitetään tarvittavan puskurin koko ja toisella kutsukerralla syötetään varattu tämän perusteella puskuri.

Jokaiselta OpenCL-alustalta puolestaan haetaan käytettävissä olevat OpenCL-laitteet. Haussa laitteet rajataan CL_DEVICE_TYPE_GPU ehdolla, eli näytönohjaimiin. Laitekriteerit on esimerkin vuoksi toteuttu lambda-funktioilla. Tässä tapauksessa niiden käyttäminen on ihanteellista, sillä näitä voidaan kutsua aivan tavallisen funktio-osoittimen välityksellä. Tällä tavoin ei tarvitse

11 Voidaan olettaa, että pinomuistin koko on joitakin satoja kilotavuja, korkeitaan muutamia megatavuja per säie. Joka tapauksessa, huomattavasti vähemmän kuin mitä kekomuistia (*heap memory*).

määritellä erillisiä apufunktiota, joten ohjelmakoodi selkeytyy. Mikäli kriteerit täyttävä laite löytyy, luodaan sille komentojono clCreateCommandQueue funktiolla. Jonoon lisättyjen komentojen suoritusajat saadaan myöhemmin selville, kun jono luodaan CL_QUEUE_PROFILING_ENABLE lipulla. Tässä vaiheessa OpenCL-laite on alustettu ja käyttövalmis haluttua tehtävää varten.

Esimerkkiohjelma lukee kuvainformaation tiedostosta. Toteutuksen pitämiseksi mahdollisimman yksinkertaisena, tiedosto tulkitaan raakadatana[12] (*raw data*). Esimerkkitiedosto on kooltaan *2048 x 2048* kuvapistettä, kukin kuvapiste koostuu kolmesta 8-bittisestä luvusta. Värikanavat on tallennettu järjestyksessä R, G ja B. Mikäli ohjelma käännetään PX_MAIN__READ_ALPHA ja/tai PX_MAIN__WRITE_ALPHA makrot ollessa määriteltyinä, luetaan ja tallennetaan vastaavasti värikanavien lisäksi myös läpinäkyvyystieto 8-bittisenä lukuna. Sekä tiedoston lukeminen, että kirjoittaminen on toteutettu tekstivuo-operaattoreilla. Yksittäistä kuvapistettä vastaa rgba_data luokka, jota käsitellään kuin tavupuskuria. Kun tällä tavoin tulkitaan luokan intanssi suoraan tavupuskurina, on kiinnitettävä huomiota siihen, miten kääntäjä sijoittaa luokan jäsenet muistissa. Muistioperaatioiden kustannuksen minimoiseksi voi olla mahdollista, että jäsenet sijoitetaan esimerkiksi 4. ja 8. (vrt. *word* tai *quad word*) jaollisiin osoitteisiin. Tällöin suoraan muistiosoitteiden perusteella tehtävät operaatiot eivät välttämättä tuota haluttuja tuloksia, sillä jäsenet eivät olekaan muistiavaruudessa ”vierekkäin” kuten ohjelmakoodin perusteella saattaisi näyttää.

Kuvapuskureita varten on prog_memory luokka, joka sisältää myös tiedon asteluvusta, jonka verran kuvaa halutaan kääntää. Esimerkkiohjelmassa käytetään clCreateImage2D funktiota kuvapuskurin luomiseen, vaikka funktio onkin poistonut jo OpenCL 1.2 versiossa[13]. Tämä johtuu siitä, että esimerkiksi Nvidia ei toimita kehityspakettinsa mukana rajapinnasta 1.2 versiota, vaan vanhemman 1.1 version – tämä siitä huolimatta, että ajurikerros mainostaa olevansa OpenCL 1.2 yhteensopiva. Lähdekuva tulee määritellä vain lukutilaan CL_MEM_READ_ONLY ja kohdekuva vain kirjoitustilaan CL_MEM_WRITE_ONLY lipuilla. Kuvapuskuri ei voi olla yhtä aikaa sekä luettavissa, että kirjoitettavissa. Lisäksi on huomattava, että kuvapuskurit, kuten muutkin OpenCL-puskurit ovat erillisiä varsinaisesta isäntäohjelmasta. OpenCL-puskuriin on siis erikseen siirrettävä isäntäohjelman puskurissa oleva tieto ja päinvastoin. Silllä kyseessä on kuitenkin erittäin rutiininomainen toiminto, määrittelemällä CL_MEM_COPY_HOST_PTR lippu OpenCL-rajapinta kopioi tiedon isäntäohjelman puskurista kyseiseen OpenCL-puskuriin. Myös muita tämänkaltaisia oikopolkuja on tuettuna OpenCL-rajapinnassa.

Kuvapuskureiden käsittelyyn liittyy myös näytteistäjä. Sen avulla voidaan valita miten toimitaan tilanteissa, joissa kuvapuskurin ulkopuolelta halutaan lukea informaatiota. Näytteistäjä voitaisiin myös määritellä varsinaisen ytimen sisällä. Ero on lähinnä siinä, halutaanko näytteistäjän asetuksia muuttaa ohjelmallisesti vai ei. Mikäli näytteistäjä määritellään ytimessä, ei näytteistäjän asetuksia voida muuttaa ilman, että ytimen lähdekoodiin tehdään muutoksia. Ymmärrettävistä syistä myös näytteistäjän määrittely tapahtuu hieman eri tavalla – varnainen toiminallisuus säilyy kuitenkin täysin samana.

Lähdepuskurien luomisen jälkeen voidaan varsinainen OpenCL-ohjelma kääntää sekä valita käännetystä ohjelmasta suoritettava ydin. Sinänsä OpenCL-ohjelman kääntäminen ei tarvitse kuin OpenCL-kontekstin, eli laitteen, jolloin mikäli tämän esimerkin ohjelma olisi monisäikeinen (*multi threading*) voitaisiin rinnakkain yhdessä säikeessä lukea kuvatietoa sekä varata näitä varten

12 Kaikissa moderneissa kuvankäsittelyohjelmissa on mahdollista tuoda tiedosto raakadatana (*import raw* ym.). Windows-käyttöjärjestelmälle tähän tarkoitukseen hyvin soveltuva ohjelma on esimerkiksi *IrfawView.*

13 Vertailun vuoksi kirjoitushetkellä tuorein OpenCL versio on 2.1.

kuvapuskurit, että toisessa säikeessä aloittaa OpenCL-ohjelman kääntäminen. Lähdekoodin saattaminen valmiiksi ohjelmaksi tapahtuu ensin kutsumalla clCreateProgramWithSource funktiota, jonka jälkeen ohjelma käännetään ja linkitetään clBuildProgram funktiolla. Isäntäohjelman tulee lukea lähdekoodi esimerkiksi tiedostosta parhaaksi katsomallaan tavalla tai sisällyttää se vaikka kiinteänä tekstijonona osaksi isäntäohjelmaa. Tässä tapauksessa lähdekoodi luetaan tiedostosta rivi kerrallaan. Mikäli se haluttaisiin sisällyttää osaksi ohjelmaa, voitaisiin käyttää esimerkiksi C++11:n raaka merkkijono-operaattoria *R”()”* (*raw string literal*); tällöin kääntäjä ei tulkitse ko. merkkijonon sisällä esiintyviä koodinvaihtomerkkejä, kuten rivivaihto *\n*, tai edes lainausmerkkejä. OpenCL-rajapinta mahdollistaa myös binääri kerrallaan tapahtuvan kääntämisen ja linkittämisen, sekä valmiiksi käännettyjen binääriohjelmien lataamisen. Lähdekoodia ei siis ole tarpeen joka kerta sekä kääntää, että linkittää vaan riittää, että se tehdän ainoastaan kerran kutakin OpenCL-laitetta kohden.

Mikäli ohjelman käännös epäonnistuu, saadaan tieto käännösvaiheesta (ei itse ohjelmasta) clGetProgramBuildInfo funktion avulla selville. On huomattava, että jos ja kun sekä käännös, että linkitys tehdään yhdellä kertaa täytyy ohjelmoijan itsensä ymmärtää palautuvan virheen laatu. Riippuen OpenCL-rajapinnan toteuttajasta, esimerkiksi linkitysvaiheen ongelmat eivät välttämättä ole selkeitä: varsinaisessa ohjelmakoodissa esiintyvät funktionimet voivat korvaantua kääntäjän omilla tunnisteilla käännösvaiheessa ja kryptisen näköinen virheilmoitus voi johtua yksinkertaisesta kirjoitusvirheestä funktion nimessä.

Kun OpenCL-laitteelle sopiva binääri on luotu, valitaan haluttu OpenCL-ydin clCreateKernel funktiolla. Tämän jälkeen syötetään ytimen argumentit clSetKernelArg funktiolla. On huomattava, että argumenttejä ei voi minkään argumenttityypin kohdalla välittää tai määrittää ”suoraan”, vaan funktiolle on annettava aina osoitin sekä tietotyypin koko. Esimerkkiohjelman tapauksessa siis astelukua ei voida suoraan välittää kiinteänä arvona, vaan sille on varattava oma muuttujansa – huolimatta siitä, ettei muuttujan arvoa muuteta alustamisen jälkeen. Argumenttien syöttämisen jälkeen OpenCL-ydin on täysin ajovalmis. Ytimen suoritus käynnistetään clEnqueueNDRangeKernel funktiolla, jonka argumenteiksi määritellään sekä laskettavan, että rinnaikkain suoritettavan laskenta-alueen koko.

Ytimen suorittamisen jälkeen jää tehtäväksi esimerkkiohjelman tapauksessa enää laskettujen tulosten siirtäminen takaisin isäntäohjelmaan ja sitä kautta tiedostoksi. clEnqueueReadImage funktilla tapahtuva kuvapuskurin kopioiminen voitaisiin määrittää tapahtuvaksi siten, että funktiokutsu ei palaudu ennen kuin komento on kokonaisuudessaan suoritettu (*blocking*), eli puskuri on kopioitu. Esimerkkiohjelmassa ei ole näin tehty, sillä sekä ydin että kopioiminen on synkronoitu isäntäohjelman suoritukseen. Vaikka todellista nopeushyötyä ei tässä juuri saavuteta, tapa toimii kuitenkin virikkeenä sille, miten rinnakkain tapahtuvilla suorituksilla voidaan nopeuttaa ohjelman toimintaa myös itse isäntäohjelmassa – ilman, että säikeitä edes on varsinaisesti käytössä.

Viimeistään tässä vaiheessa monissa internetistä löytyvistä OpenCL-esimerkeissä, olikin kyseessä mukamas ”C++-ohjelma”, isäntäohjelman puolella joudutaan vapauttamaan kaikki ohjelman varaamat resurssit. Sillä esimerkkiohjelmassa on kuitenkin hyödynnetty C++-ohjelmointikielen sisäänrakennettuja ominaisuuksia, tapahtuu resurssien vapauttaminen automaattisesti kääntäjän luomalla koodilla. Sama koskee myös kaikkia niitä tilanteita, jossa esimerkiksi virheestä johtuen ohjelman suoritus joudutaan lopettamaan ennenaikaisesti. Aivan täysin ongelmatonta tällainen näkyvyysalueisiin perustuva resurssointi ei kuitenkaan ole: Esimerkkiohjelmassa käytössä olevien prog_global ja prog_memory luokkien instanssit ovat kyllä tietyssä mielessä määritelty koko

ohjelman yhteisellä näkyvyystasolla (*global*), mutta todellisuudessa tällä tasolla olevat todelliset instanssit `__global` ja `__memory` vastaavasti, ovat ainoastaan heikkoja osoittimia[14] (*weak pointer*). Heikko osoitin ei omista (*own, ownership*) osoittimen resurssia, vaan mikäli resurssia halutaan käyttää, täytyy se erikseen lukita (*lock*) eli muuttaa varsinaiseksi älykkääksi osoittimeksi. Lukitseminen ei onnistu, mikäli resurssi on jo tuhottu (yksikään varsinainen älykäs osoitin ei enää osoita resurssiin) tai sitä ei ole vielä luotu. Mikäli edellä mainitut kahden luokan instanssit määriteltäisiin yhteisellä tasolla älykkäiksi osoittimiksi, niiden luominen ja alustaminen kyllä onnistuisi. Sen sijaan enää sitä, milloin niiden instanssit tuhottaisiin ei voitaisi riittävällä tarkuudella kontrolloida[15]. Ne tuhottaisiin ”joskus” sen jälkeen, kun suoritus siirtyy `main` funktion ulkopuolelle. Tällä on kuitenkin esimerkkiohjelman kannalta katastrofaalisia seurauksia: ohjelmassa käytetään paljon staattisia muuttujia (*static*), esimerkiksi lueteltujen tyyppien kanssa (vrt. ”px_enum” komponentti). Vaikka staattisten muuttujien alustamisjärjestys on määritelty[16], sen sijaan järjestystä missä ne tuhotaan tai vapautetaan ei ole. Tällöin esimerkkiohjemassa käy vääjäämättä niin, että kun testaustilassa (*debug*) tulostetaan OpenCL-rajapintafunktion paluuarvoa vastaava teksti, niin jonkin *clRelease* funktion kohdalla tekstin ja arvon yhdistävä, staattisesti varattu *map* säiliö onkin jo vapautettu! Tällöin ohjelma joko kaatuu, jää jumiin tai mitään ei yksinkertaisesti tapahdu – joka tapauksessa, kyseessä on määrittelemätön tapaus (*undefined behaviour*), eli toisin sanoen ohjelmointivirhe. Englanninkielisessä lähdemateriaalissa ongelmaan viitataan usein termillä *static initialization fiasco*, mutta tässä tapauksessa ongelma – tai fiasko – ei esiinny alustuksessa, vaan nurinkurisesti vapauttamisvaiheessa.

Älykkäillä osoittimilla staattisen muistinvarauksen ongelmat C++-ohjelmointikielessä voidaan kuitenkin välttää, kuten on tehtykin, määrittelemällä yhteisellä tasolla ainoastaan heikko osoitin. Resurssin alustamisesta ja vaputtamisesta huolehtiva älykäs osoitin voidaan sijoittaa sellaiselle näkyvyysalueelle, jolla resurssin tarvitseminen muut ominaisuudet ovat käytössä. Tässä tapauksessa `prog_global` ja `prog_memory` luokan instanssit ovat olemassa ainoastaan `main` funktion sisällä. Kun suoritus poistuu tältä funktiotasolta, joko normaalisti tai virheen seurauksena kääntäjän automaattisesti luoma koodi yhdessä älykkään osoittimen toteutuksen kanssa vapauttaa luokkien instanssit[17].

---

14 Heikot osoittimet, `weak_ptr`, liittyvät suoraan älykkäisiin osoittimiin. Ne eivät ole suoraan C++-ohjelmointikielen sisäänrakennettuja tietotyyppejä, vaan löytyvät standardikirjastosta erillisinä luokkina, kuten varsinainen älykäs osoitin `shared_ptr`. Niille on muitakin hyödyllisiä käyttötarkoituksia kuin se, mitä tässä on sivuttu.

15 Vapauttaminen esimerkiksi `reset` jäsenfunktiolla taas sotii kaikkea sitä vastaan, mitä resurssien varaamisen tulisi C++-kielessä olla.

16 Muuttuja alustetaan, kun jokin ohjelman säie kohtaa staattisesti määritellyn muuttujan – sitä ei ole määritelty, onko alustamisen oltava säeturvallista (*thread safe*). Sen sijaan muuttujien keskinäistä alustamisjärjestystä ei ole määritelty.

17 Tässä tapauksessa ei lähdetä siitä olettamuksesta, että em. älykkäistä `shared_ptr` osoittimista olisi tässä vaiheessa useampi kuin yksi kopio olemassa. Ts. resurssia ei voitaisi vapauttaa, sillä `main` funktiosta poistuttaessa viittausmäärä (*reference count*) olisikin vielä suurempi kuin nolla.

OpenCL-ydin

OpenCL-ydin ohjelmoidaan OpenCL-ohjelmointikielellä, joka perustuu C99-ohjelmointikieleen (tai standardiin) lisäten siihen joitakin tietotyyppejä sekä valmiita funktiota. Joitakin laskennan kannalta tarpeettomia ja suhteessa teknisesti hankalia omaisuuksia, kuten funktio-osoittimet, on myös poistettu käytöstä. Lisäominaisuudet perustuvat puhtaasti laskentatarpeisiin, eivätkä niinkään ohjelmointiteknisiin seikkoihin (vrt. C ja C++). Heti aluksi on huomioitava, että se, mitä edellisissä kappaleissa käytiin lävitse liittyy isäntäohjelman toimintaa. Mikään rajapinnasta isäntäohjelmaan päin näkyvä ominaisuus ei ole käytettävissä, vaan OpenCL-kääntäjää koskevat omat lainalaisuutensa: ydin on siis rajapinnan ”sisäpuolella”, mikäli asian haluaa näin mieltää.

Kuvan pyörittämisen suorittava ydin on esitetty komponentissa ”px_rotate”. Näkyvin ero perinteiseen C-ohjelmaan verrattuna ovat vektorityypit (*intN*, *floatN* jne.) sekä funktio- ja parametritarkenteet (*__kernel*, *__read_only*, *__write_only* jne.). Näiden käyttäminen on pakollista. Varsinainen ydin on merkittävä *__kernel* funktiotarkenteella. Lisäksi, sillä kuvapuskurien on oltava aina joko luku tai kirjoitusmuotoisia, ne on merkittävä *__read_only* ja *__write_only* tarkenteilla vastaavasti. Sillä täysin sama ydin suoritetaan annetun syötteen joka pisteelle, tapahtuu sen hetkisen suorituskohdan tunnistaminen `get_global_id` funktiolla. Funktion argumentillä kerrotaan mikä dimensio (*0*, *1*, *2* eli *X*, *Y*, *Z* vastaavasti) halutaan saadaan selville. Esimerkkitapauksessa se on siis kuvakoordinaatti (*X*, *Y*). `get_global_size` funktio palauttaa puolestaan koko käsiteltävän alueen koon, eli esimerkissä kuvan mitat jotka ovat tässä tapauksessa *2048* x *2048*. Kaikki tyyppimuunnokset on käytännössä pakko suorittaa eksplisiittisesti. Tyyppimuunokset voidaan tehdä tyyppimuunnosoperaattorilla, kuten C-kielisissä ohjelmissa yleensä, tai käyttämällä sopivaa `convert_destType_sat[_roundingMode]` funktiota käyttämällä. Käyttämällä erillistä funktiota operaattorin sijasta, voidaan valita saturoitu ja/tai halutulla tavalla pyöristetty lopputulos. Erillisten, lähinnä `_sat` päätteisten funktioiden avulla hoidetaan myös etenkin DSP-suorittimille tyypilliset saturoidut matemaattiset operaatiot, kuten yhteen-, vähennys- ja kertolaskut. OpenCL-kielelle ominaiset tehtävät on siis toteutettu lähinnä luomalla kieleen (pseudo)funktioita, eikä niinkään määrittelemällä C-operaattoreita uusiksi tai tarjoamalla niille lisätoiminta, vrt. C++-ylikuormitus tai merkkijonomääritteet (*string literal*).

Esimerkkiohjelma kääntää kuvan käännösasetuksista riippuen joko yhdellä kiertomatriisilla tai kolmella leikkausmatriisilla. Laskostumis- tai pyöristysvirheiden (*aliasing error*) välttämiseksi kääntäminen tapahtuu kuitenkin ”väärin päin”. Ytimen käsittelemälle kuvapisteelle (*X*, *Y*) ei lasketa, mihin uuteen kuvapisteeseen (*X'*, *Y'*) se kääntyy, vaan kuvapiste *josta* kääntämällä kulman `deg` verran päästään nykyiseen kuvapisteeseen. Tällä tavoin saavutetaan kaksi etua: ensinnäkin, selkeästi näkyviä laskostumisvirheitä ei pääse syntymään ja toisekseen tämä edustaa ajattelutapaa, jolla juurikin OpenCL-ohjelmoinnin ominaispiirteitä voidaan hyödyntää. Kuten edellä on viitattu, näytteistäjä voidaan OpenCL:ssä määritellä ainoastaan lähdepuskurille. Hyödyntämällä näytteistäjää, kuvan laidoilla tapahtuva koordinaattipisteiden ”ulkopuolelta” piirtyvät kuvapisteet saadaan hoidettua täysin näytteistäjän avulla, jonka toimintaa voidaan vielä haluttaessa muuttaa eri näytteistysasetusten avulla.

2 x *4* matriisien kerto-operaatio on toteutettu `px_2x4_mtx` funktiossa. Funktio on määritelly `__private` tarkenteella, eli se ei ole ydin, mutta sitä voidaan kutsua ytimestä. Samalla tarkenteella on myös määritelty parametrit, eli ne ovat joitakin ytimen muuttujia; eivät kuitenkaan esimerkiksi isäntäohjelmalta tulevia puskureita. OpenCL-ohjelmointikielessä osoittimet toimivat kuten C-

kielessä yleensä. On huomattava, että vektorimuuttujien komponentteihin voidaan viitata joko *x*, *y*, *z* ja *w* notaatiolla tai komponentin järjestysnumerolla heksamuodossa. Tällöin järjestysnumeron eteen on kuitenkin lisättävä tunnus s tai S. Komponentteja voidaan myös yhdistää, eli esimerkiksi funktiossa käytetty mtx->xy (jossa mtx on nelikomponenttinen vektorityyppi) esitys on täysin sama kuin mitä olisi (f2)(mtx->x,mtx->y). On huomattava, että luonnollisesti mikäli kyseessä ei ole osoitin, tulee -> osoitinjäsenoperaattorin (*member of pointer operator*) sijasta käyttää pelkkää . jäsenoperaattoria (*member of object operator*) komponentteja käsitellessä, aivan kuten C-sukuisissa kielissä yleensäkin.

Kun haluttu lähdekoordinatti (*X'*, *Y'*) on laskettu matriisioperaatioiden avulla, luetaan kuvapiste lähdekuvapuskurista read_imagef funktiolla. Mikäli PX_ROTATE__VISUALIZE_LOCAL_GROUPS makro on määritelty, väritetään kuvasta kukin työryhmälohko. Tämän avulla voi viimeistään olla helpompi hahmottaa, kuinka työ(ryhmät) jakaantuvat. Kun kuvapiste on kirjoitettu kohdepuskurin (*X*, *Y*) koordinaattiin, on ydin suorittanut tehtävänsä sen pisteen osalta. Kun kaikki kuvapisteet on käsitelty, jatkuu esimerkkiohjelman pääasiallinen suoritus taas isäntäohjelmassa, kunhan laskettu kuvatieto on siirretty näytönohjaimen muistista OpenCL-rajapinnan välityksellä takaisin isäntäohjelman saataville, jossa se tässä tapauksessa talletaan uuteen tiedostoon.

Esimerkissä ei käsitellä työryhmän jäsenille yhteisesti näkyviä muuttujia. Nämä muuttujat määritellään yksinkertaisesti __*local* tarkenteella. Tällöin viimeistään on monesti tarpeen myös synkronoida ytimien keskinäinen suoritus työtyhmän sisällä barrier funktiolla.

Lopuksi

OpenCL-ohjelmointi on siis kaikessa yksinkertaisuudessaan C-kielistä ohjelmointia, joka ei olennaisilta osin eroa etenkään DSP-suorittimien kanssa tehtävästä ohjelmointityöstä: laskettaessa hyödynnetään ennen kaikkea fyysistä laitteistoarkkitehtuuria, jonka avulla moninkertainen nopeushyöty saavutetaan. Tietyllä tapaa monimutkaiset laskentaoperaatiot (vrt. saturoidut operaatiot: tulee tarkistaa, ettei operaation syöte vie lopullista tulosta minimi- ja maksimiarvojen ulkopuolelle; tämä on perinteisillä suorittimilla hidasta, sillä arvojen tarkastamiseen kuluu arvokkaita kellosyklejä) voidaan suorittaa hyvin nopeasti, muutamilla kellosykleillä. Tämän lisäksi rinnakkaislaskenta voidaan ulottaa yksinkertaisten toistorakenteiden, kuten *for,* ulkopuolelle. DSP-suorittimilla tämä on OpenCL-ohjelmointiin suurin ero, eli yhden DSP-ytimen sisällä (näennäien) rinnakkaisuus (*pipelining*) onnistuu vain jakamalla laskenta tapahtumaan yhtä aikaa eri laskuyksiköissä (*math unit* ym.).

Myös painotus OpenCL-isännän ja -ytimen välillä on selkeä: isäntäohjelma on laaja, ja ydin hyvin pieni suhteessa siihen. Sama koskee niin tätä dokumenttiä, kuin esimerkkiohjelmaakin. Tekstillisesti eniten painoarvoa on isäntäohjelmalla ja sen toteutuksessa, vastaavasti, ytimen käsittelyyn ei juuri ole tarvetta erikseen paneutua. Ohjelmakoodin puolesta ydin sisältää alle 5% siitä rivimäärästä, mitä tarvitaan isäntäohjelman käyttöön.