Leevi Leino

GPU-laitteistokiihdytetyn ohjelman optimointimenetelmät ja periaatteet

Tietotekniikan kandidaatintutkielma

20. toukokuuta 2024

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Leevi Leino

Yhteystiedot: leinolm@student.jyu.fi

Ohjaaja: Tuomo Rossi

Työn nimi: GPU-laitteistokiihdytetyn ohjelman optimointimenetelmät ja periaatteet

Title in English: Optimization techniques and principles of GPGPU programs

Työ: Kandidaatintutkielma

Sivumäärä: 23+0

Tiivistelmä: GPU-laskennassa esiintyy rinnakkaisen arkkitehtuurin takia erityisiä haasteita optimoinnin kanssa. Keskussuorittimelle optimaalisin toteutus ei usein ole optimaalisin rinnakkaisen laskennan yksikölle ja esimerkiksi käskyjen määrä ei ole hyvä metriikka ohjelman optimaalisuudelle. Tämä tutkielma toimii johdattelevana kartoituksena GPU-optimoinnin keskeisimpiin teemoihin ja esittelee yleisimpiä optimointimenetelmiä.

Avainsanat: GPU, optimointi, GPGPU, OpenCL

Abstract: With regard to optimization, GPGPU programming comes with its own set of challenges. Code optimized with single thread performance in mind might not be optimal for a parallel processor and instruction count might not correlate with performance. This thesis aims to work as an introduction to the realm of GPU optimization and presents the most prevalent themes and techniques.

Keywords: GPU, optimization, GPGPU, OpenCL

Kuviot

Kuvio 1. Käskyjen suoritus ilman liukuhihnaa 9
Kuvio 2. Käskyjen suoritus liukuhihnalla 10
Kuvio 3. Liukuhihnan tukkeutuminen hyppykäskyn takia 10

Sisällys

1 JOHDANTO 1

2 GRAFIIKKAPROSESSORIT 2
2.1 OpenCL:n Rakenne 2
2.2 Optimoinnista 3

3 MUISTIN KÄYTTÖ 5
3.1 Muistihierarkia 5
3.2 Muistiviittaukset 6

4 KOODIPOLUT 8
4.1 Käskytason rinnakkaisuus 8
4.2 Haarojen eliminointi 10

5 MUITA MENETELMIÄ 13

6 YHTEENVETO 15

LÄHTEET 16

1 Johdanto

GPU:t eli grafiikkaprosessorit ovat jokaisen kotitalouden supertietokoneita. Ne tarjoavat usean kertaluokan verran enemmän laskentatehoa käyttäjilleen.

Grafiikkaprosessorien huima laskentateho on toteutettu soveltamalla suuren määrän rinnakkaisuutta. Nykyaikaisissa grafiikkaprosessoreissa on laskentayksiköitä tuhansia, mutta yhden yksikön laskentateho on paljon heikompi. Täten huiman laskentatehon hyödyntäminen vaatii erilaista lähestymistapaa ohjelmaa suunniteltaessa.

Rinnakkaisuus tuo mukanaan omia haasteita ja ongelmia, ja tässä tutkielmassa esitellään yleisempiä optimoinnin teemoja ja niihin perustuvia menetelmiä.

Tutkielman luku 2 sisältää tiiviin katsauksen grafiikkaprosessorin toimintaperiaatteeseen ja antaa yleiskuvan keskeisimpiin teemoihin. Myöhemmissä luvuissa käsitellään oleellisimpia teemoja ja lopuksi käydään kokoava yhteenveto.

2 Grafiikkaprosessorit

Grafiikan sovellukset ovat luonteeltaan suorittimelle raskaita. Esimerkiksi suorakulmion muotoisen alueen täyttäminen yhdellä värillä veisi suorittimelta $O(n \times m)$ verran aikaa, missä n ja m ovat suorakulmion leveys ja korkeus. Muita tyypillisiä suorittimelle raskaita grafiikan sovelluksia ovat kontrastin muutokset, konvoluutioon perustuvat kuvan manipulaatiot (Gauss - sumennus, terävöitys) ja 3D-grafiikan liukuhihna.

3D-grafiikan liukuhihna koostuu useasta vaiheesta. Oleellisin vaihe on 3D-mallien kärkipisteiden affiinit kuvaukset ja projektio tasolle. Seuraavaksi oleellisin vaihe on kärkipisteiden muodostavien monikulmioiden rasterointi.

Edellä mainitut grafiikan sovellukset ovat luonteeltaan erittäin rinnakkaisia, josta syystä grafiikkaprosessorit ovat rakennettu soveltaen varsin rinnakkaista arkkitehtuuria. Grafiikan sovellusten monimutkaistuttua grafiikaprosessoriden on täytynyt sopeutua olemalla yleiskäyttöisempiä (Myllykoski 2015). Nykyään grafiikkaprosessoreja voi käyttää yleislaskentaan käyttäen rajapintoja kuten OpenCL ja CUDA.

2.1 OpenCL:n Rakenne

OpenCL on avoimen lähdekoodin ohjelmointikehys rinnakkaiseen laskentaan (*OpenCL specification* 2023). OpenCL mahdollistaa ohjelmien ajettavan alustoilla, jossa on käytössä sekä normaaleja suorittimia että erilaisia grafiikkaprosessoreja. Näin on helppo vaihtaa ohjelmaa suorittava laitteistokokoonpano yhdestä toiseen.

OpenCL jakaa laitteiston päälaitteeseen ja useisiin laitteisiin. Laitteet voivat olla esimerkiksi grafiikkaprosessoreja tai tietokoneita. Laite sisältää laskentayksiköitä (engl. *compute unit*), jotka sisältävät suorituselementtejä (engl. *processing element*).

Yksi laskentayksikkö suorittaa yhtä työryhmää. Työryhmä (engl. *work-group*) on kokoelma toisiinsa liittyviä työalkioita (engl. *work-item*), jotka jakavat lokaalin muistin. Työalkiot ovat laskennan yksiköitä ja niiden voidaan ajatella olevan säikeitä (Myllykoski 2015).

Työryhmän työalkioita voidaan ajaa joko yhdessä SIMD - tyylisesti tai erillisesti SPMD - tyylisesti. SIMD (Single Instuction Multiple Data) on laskennan rakennemalli, jossa säikeet suoritetaan rinnan käyttäen yhteistä käskyosoitinta. Tällöin kaikkien säikeiden täytyy suorittaa täysin samat käskyt. SPMD (Single Program Multiple Data) - arkkitehtuuri puolestaan mahdollistaa säikeiden välille eriaikaisen suorituksen, sillä jokaisella säikeellä on oma käskyosoitin. Täten säikeet voivat olla toisiinsa verrattuna aivan eri kohdissa koodin suorittamista.

OpenCL - ohjelma koostuu päälaitteen suorittamasta koodista (engl. *Host code*) ja laskentayksiköiden suorittamista koodeista, joista kutsutaan ydinfunktioiksi (engl. *Kernel*). Päälaite antaa laitteille ydinfunktioita komentoina. Komentojen antaminen tapahtuu päälaitteen suorittamassa koodissa, jonka ohjelmoija kirjoittaa itse. Ohjelmoijalla on näin suuri vapaus delegoida ydinfunktioita haluamallaan tavalla.

2.2 Optimoinnista

Hijma ym. (2023) kuvailee optimointimenetelmien jakautuvan neljään laajaan teemaan: Muistiviittaukset, epäsäännöllisyys, tasapainotus ja päälaitteen kanssa kommunikointi. Teemojen välillä on jonkin verran päällekkäisyyksiä ja moni menetelmä kuuluu useaan kategoriaan.

Muistiviittauksiin perustuvat optimointimenetelmät perustuvat siihen, että informaation hakeminen muistista saattaa viedä huomattavan määrän aikaa. Järkevä muistin käyttö tarkoittaa, että pyritään hakemaan ulkoisesta muistista mahdollisimman harvoin informaatiota ja pyritään saada muistihausta mahdollisimman paljon hyötyä.

Epäsäännnöllisyyteen perustuvat optimointimenetelmät pyrkivät vähentämään tilanteita, joissa koodi haarautuu useaksi valinnaiseksi poluksi. Tälläisiä ovat esimerkiksi silmukat ja konditionaalilauseet.

Tasapainotuksella pyritään pitämään huoli, että mahdollisimman suuren osan ajasta käytetään rinnakkaiseen laskentaan ja että suurin osa säikeistä tekee työtä. Tasapainotuksen menetelmiä ovat muun muassa synkronointiin kuluvan ajan minimointi ja säikeiden välinen työn määrän tasapainotus.

Päälaitteen kanssa kommunikointiin perustuvat optimoinnit keskittyvät grafiikkaprosessoria ympäröivän laitteen hyödyntämiseen. Tälläisiä optimointimenetelmiä ovat esimerkiksi ongelman sarjallisesti raskaan osan suorittaminen päälaitteen suorittimella ja laitteiden välinen optimoitu kommunikointi.

Luvussa 3.2 käsitellään muistiviittauksiin perustuvia menetelmiä ja luvussa 4.2 käsitellään epäsäännnöllisyyteen perustuvia optimointitekniikoita ja arvioidaan niiden tehokkuutta. Luvussa 5 esitellään muita optimointitekniikoita.

3 Muistin käyttö

Muistin käyttö on hidasta. Esimerkiksi Ryoo ym. (2008, Taulu 1) mukaan GeForce 8 - mallin näytönohjaimilla syntyy yli sadan syklin viive laitteen ulkopuolisesta muistista haettaessa. Myöskin Myllykoski (2015) toteaa latenssin tyypillisesti olevan satoja syklejä. Täten ollen toistuva muistista hakeminen hidastaa ohjelmaa huomattavasti.

Ulkoisen muistin hitauteen on monia tekijöitä, joista suurin osa liittyy tiedonsiirtoon. Hajasaantimuisti (engl. *Random Access Memory*) on tietokoneessa fyysisesti erillinen komponentti. Erillisten komponenttien välinen tiedonsiirto tapahtuu emolevyn muistiväylien kautta ja tiedonsiirto vaatii erillisiä toimenpiteitä, joista aiheutuu enemmän viivettä.

Täten on tärkeää minimoida ulkoisten muistiviittausten määrä. Osiossa 3.1 käsitellään grafiikkaprosessorin sisäisen muistin rakennetta ja luvussa 3.2 esitellään erilaisia tapoja optimoida muistiin viittaaminen.

3.1 Muistihierarkia

OpenCL määrittelee neljä muistialuetta (*OpenCL specification* 2023, 3.3.1). Suurin kapasiteetiltaan on globaali muisti, joka tyypillisesti toteutetaan hajasaantimuistilla. Globaali muisti siksi onkin muistialueista hitain.

Lokaali muisti muistialueena on yhden työryhmän nähtävillä kerrallaan. Lokaali muisti on tyypillisimmin grafiikkaprosessorin sisäistä muistia ja siksi onkin huomattavasti nopeampaa latenssin ja kaistanleveyden kannalta (Myllykoski 2015, 2.1.3). Säikeiden kontekstit tallennetaan tyypillisesti tähän muistialueeseen.

Kilpajuoksutilanteiden välttämiseksi OpenCL:ssä on käytössä synkronointimekanismeja kuten muurit (engl. *barrier*). Muuri estää säiettä etenemästä tiettyä pistettä pidemmälle, ennen kuin muut työryhmän alkiot ovat päässeet samaan pisteeseen. Näistä synkronointimekanismeista koituu myös viivettä, jos säikeet eivät ole synkronoitu (*OpenCL specification* 2023).

Vakiomuisti (engl. *Constant memory*) on nimensä mukaisesti muistialuetta, johon on lasken-

tayksiköillä vain lukuoikeus. Tästä syystä muistiin voi moni laskentayksikkö viitata kerralla ja keskimääräisesti latenssia tulee vähän, sillä muistiviittauksista syntyy vain yksi transakio (Hijma ym. 2023, 6.1.1). Transaktio on Mutex-lukon kaltainen synkronointimekanismi, jolla voi varmistaa atomisuuden säilyttämällä rinnakkaisuuden lukkoja paremmin (Herlihy ja Moss 1993).

OpenCL määrittelee myös yksityisen muistin, joka on jokaisella säikeellä oma. Yksityinen muisti on yleensä toteutettu varaamalla osan globaalista muistista kyseiseen käyttöön. Yksityinen muisti ei ole muiden säikeiden käytettävissä.

3.2 Muistiviittaukset

Yksi yleisimmin käytetty optimointimenetelmä on yhdistetty muistihaku (engl. *Coalesced access*). Kun kaikki työryhmän säikeet hakevat peräkkäisiä muistiosoitteita samanaikaisesti, on laitteelle mahdollista yhdistää muistihaut yhdeksi. Näin saavutetaan suurin mahdollinen kaistanleveys. Yang ym. (2010) artikkelissa esitelty kääntäjä hyödyntää yhdistettyä muistinhakua optimoinnissa. Myöskin Kivioja, Mönkölä ja Rossi (2022) hyödyntävät tekniikkaa supranesteiden pyörteiden mallintamisessa.

Lokaali muisti on tyypillisesti laitteiston tasolla jaettu pankkeihin. Muistipyynnöt useaan muistialueeseen samassa pankissa aiheuttavat ristiriidan (engl. *Bank Conflict*). Tällöin muistipyynnöt täytyy suorittaa sarjallisesti (Kivioja, Mönkölä ja Rossi 2022, luku 4.1; Myllykoski 2015, luku 2.2). Kivioja, Mönkölä ja Rossi (2022) välttävät ristiriidan täyttämällä pankin kokoisen muistialueen täytteellä (engl. *padding*).

Khan ym. (2014) vuorostaan esittelevät kaksi matriisin transpoosialgoritmia, jotka hyödyntävät yhdistettyä muistinhakua välttäen pankkien ristiriidan ilman täytettä. Algoritmissä haetut rivit kirjoitetaan muistiin vinottain, jolloin pankkien ristiriitaa ei tapahdu. Algoritmien tehokkuutta vertailiin Nvidia:n transpoosialgoritmiin soveltamalla niitä gauss-sumennokseen. Testituloksissa kävi ilmi, että Khan ym. (2014) algoritmia käyttävä sumennos oli noin 6% nopeampi verrattuna Nvidia:n transpoosia hyödyntävään.

Esihaku (engl. *Prefetching*) on myös yleisesti käytetty optimointitekniikka. Tarkoitus on mi-

nimoida muistihaun latenssi hakemalla tarvittava tieto huomattavasti ennen sen tarvitsemista. Lee, Kim ja Vuduc (2012) arvioi esihaun hyöty- ja haittapuolia. Suurin haitta on käskyjen määrän huomattava kasvaminen. Lee, Kim ja Vuduc (2012) mainitsee, että ennenaikainen tai turha esihaku aiheuttaa välimuistin saastumista (engl. *cache pollution*), mikä aiheuttaa välimuistipuutoksia (engl. *cache miss*). Tosin Lee, Kim ja Vuduc (2012) huomioi, että tutkimuksen aikana esiintyi ennenaikaisia ja turhia esihakuja vähän.

Ydinfunktioiden yhdistämisellä (engl. *Kernel fusion*) voidaan mahdollistaa parempi muistin lokaalisuus (Filipovič ym. 2015). Eri ydinfunktioita suorittavien säikeiden välinen tiedonsiirto täytyy toteuttaa globaalilla muistilla. Ydinfunktioiden yhdistämisen ansiosta voidaan hyödyntää lokaalia muistia säikeiden väliseen tiedonsiirtoon. Wang, Lin ja Yi (2010) tutkii ydinfunktioiden yhdistämisen vaikutusta laskentayksikön käyttöasteeseen ja energian kulutukseen. Parhaimmillaan suoritusaika väheni 20%.

Kuitenkin on tärkeä pitää mielessä, että ydinfunktioiden yhdistäminen ei välttämättä ole suotuisaa. Yhdistäminen kasvattaa säikeen tarvitseman muistin määrää ja täten saattaa vähentää laitteen käyttöastetta (Filipovič ym. 2015). Käyttöaste vähenee myös, jos yhdistettyjen ydinfunktioiden työmäärä eriää huomattavasti (Wang, Lin ja Yi 2010). Tällöin aiemmin valmiit säikeet jäävät odottamaan toisia säikeitä.

4 Koodipolut

Koodin puhutaan haarautuvan, kun ehtolauseet määrittelevät valinnaisia koodipolkuja. Haarautuvat koodipolut ovat haastavia SIMD - rakenteellisille laitteille, sillä säikeillä on käytössä yhteinen käskyosoitin. Tämänhetkinen ratkaisu ongelmaan on suorittaa molemmat haarat sarjassa ja kytkeä pois päältä ne työalkiot, jotka eivät kulje haaraa pitkin (Myllykoski 2015).

Kun työalkiot kulkevat samaa koodipolkua pitkin, on kontrollivuo konvergentti (engl. *convergent*). Vastaavasti kun alkiot kulkevat eri koodipolkuja pitkin, sanotaan kontrollivuon olevan divergentti (engl. *divergent*) (*OpenCL specification* 2023). SIMD - pohjaiset laitteet ovat optimoituja konvergentille kontrollivuolle (*OpenCL specification* 2023; Myllykoski 2015).

Kontrollivuolla (engl. *Control flow*) viitataan koodin kulun rakenteeseen. Keskeisin asia on suoritusjärjestys ja mahdolliset haarat. Kontrollivuo esitetään yleensä vuokaavion avulla.

Haarautuvan koodin hitaus polveutuu myös pitkälti muistiviittausten hitaudesta. Laitteen suorittamat käskyt haetaan muistista ennen suoritusta. Osiossa 4.1 kerrotaan asiasta enemmän.

4.1 Käskytason rinnakkaisuus

Käskytason rinnakkaisuus (engl. *Instruction Level Parallellism*) on kokoelma kääntäjän tai laitteen toteuttamia optimointitekniikoita, jotka perustuvat käskyjen rinnakkaisuuteen. Näitä on muun muassa käskujen liukuhihna ja järjestämätön suoritus (engl. *Out-of-order execution*).

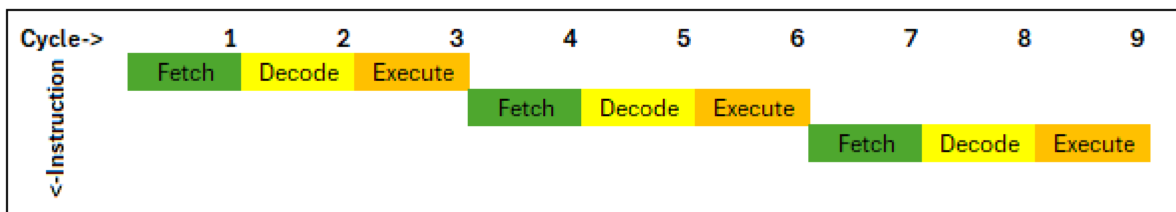
Nykyaikaisissa suorittimissa on käytössä käskyjen liukuhihna (engl. *Instruction pipelining*). Liukuhihna on rinnakkaistamiseen perustuva optimointimenetelmä, jossa tehtävä jaetaan osiin, joista jokaiselle on oma autonominen suorittaja (Ramamoorthy ja Li 1977). Nämä osat voidaan täten suorittaa rinnakkain, jolloin useampaa tehtävää voidaan suorittaa samanaikaisesti. Kuviossa 2 demonstroidaan käskyjen liukuhihnaa. Kuviossa 1 on esitettynä samat käskyt ilman käskyjen liukuhihnaa

Käskyjen liukuhihnat jakavat käskyt yleisimmin seuraaviin osiin: käskyn haku, käskyn purku, käskyn suoritus (engl. *Fetch, Decode, Execute*). Näistä osista käskyn hakuun kuuluu eniten aikaa. Useissa laitteissa käskyt ovat jaettu vielä useampiin osiin, jolloin on mahdollista suorittaa useampi käsky limittäin. Esimerkiksi Ramamoorthy ja Li (1977) dokumentissa jaetaan käskyt osiin IF(Instruction Fetch), ID(Instruction Decode), OF(Operand Fetch) ja EXEC(Execute).

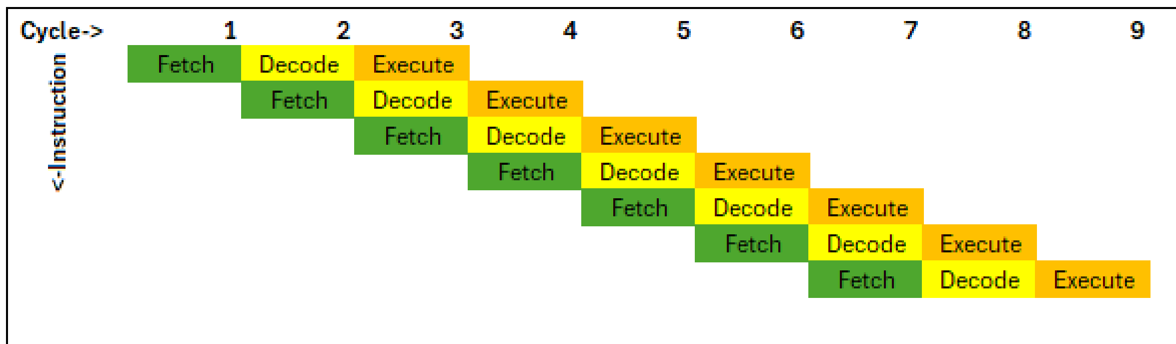
Toisistaan riippuvat käskyt hidastavat laskentatehoa. Liukuhihnassa täytyy jäädä odottamaan edellisen käskyn tulosta, ennen kuin käskyn suoritusta voidaan jatkaa. Ilmiöstä käytetään englanniksi nimitystä *Stalling*. Erityisen pahoja ovat esimerkiksi koodihaaroista syntyvät ehdolliset hyppykäskyt, sillä silloin liukuhihna jää ensimmäisessä vaiheessa odottelemaan edellisen käskyn viimeistä vaihetta. Kuviossa 3 on kuvattu ilmiötä.

Useissa suorittimissa on toteutettu spekuloiva suoritus (engl. *Speculative execution*). Spekuloiva suoritus pyrkii nopeuttamaan liukuhihnaa tekemällä ennustuksen kuljettavasta koodihaarasta tai riippuvan tiedon arvosta. Väärän ennusteen tehtäessä täytyy suorittimen korjata tilanne, mistä koituu hidastusta. Ilmiöstä käytetään nimitystä rangaistus (engl. *Penalty*).

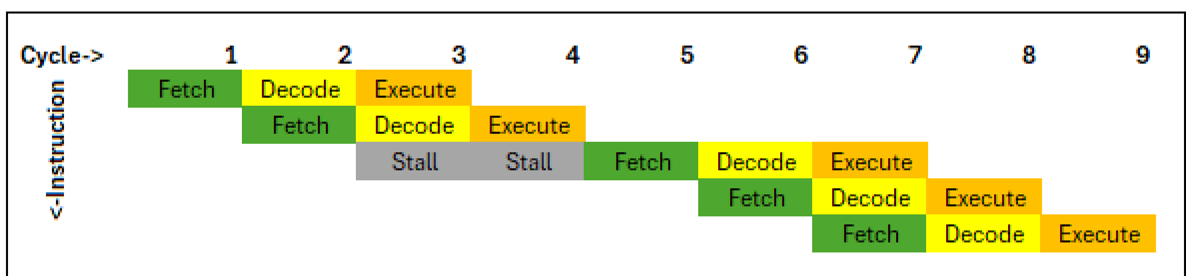
Ehdolliset hyppykäskyt ovat haitaksi myös muille käskytason rinnakkaisuuden tekniikoille, kuten järjestämättömälle suoritukselle ja sitä edistävälle *käskyjen vuoronnukselle*. Käskyjen vuoronnus (engl. *Instruction scheduling*) perustuu toisistaan riippumattomien käskyjen järjestämiseen sellaiseen muotoon, jossa liukuhihnan on mahdollista suorittaa peräkkäisiä käskujä ilman liukuhihnan jumiutumista. Myöskin järjestämätön suoritus hyötyy käskyjen vuorontamisesta.



Kuvio 1: Käskyjen suoritus ilman liukuhihnaa



Kuvio 2: Käskyjen suoritus liukuhihnalla



Kuvio 3: Liukuhihnan tukkeutuminen hyppykäskyn takia

4.2 Haarojen eliminointi

Haarautuva koodi on ollut monen tutkimuksen aiheena. Syynä on koodihaarojen negatiivinen vaikutus suorituskyvylle. Tutkimuksissa on pyritty eri tavoilla vähentämään koodihaarojen negatiivista vaikutusta suorituskyvylle.

Mueller ja Whalley (1995) tekstissään esittelemässä menetelmässä ensin etsitään koodista riippuvuuksia konditionaalilauseiden ja koodipolkujen välillä. Täten on mahdollista selvittää, onko joitain ehdollisia haaroja ylipäätänsä mahdollista välttää. Seuraavana muutetaan kontrollivuon rakenne muotoon, jossa ehdolliset hyppykäskyt voidaan korvata ehdottomilla hyppykäskyillä. Lopuksi järjestetään vuo muotoon, jossa ehdottomia hyppykäskyjä ei ollenkaan tarvita hyödyntäen koodin kahdentamista.

Tulosten mittauksessa ilmeni, että optimointimenetelmää hyödyntäen käännetyt ennalta valitut ohjelmat suorittivat keskimäärin 14% vähemmän käskyjä. Ehdollisia haaroja suoritettiin 20% vähemmän. On huomioitavaa, että ohjelmien koko kasvoi keskimäärin 16%, mikä voi lokaalin muistin kannalta tuottaa suorituskykyhaittoja.

Bodík, Gupta ja Soffa (1997) laajentavat tutkimusta tutkimalla haarojen vähentämistä myös useiden funktioiden muodostaman kokonaisuuden kannalta. Erityisesti keskitytään tarpeettomiin haaroihin, jotka syntyvät kutsun parametrien tarkastamisessa tai palautetun arvon tarkastamisessa. Ratkaisu on määritellä funktiolle useampi tulopiste ja paluuosoite, joita käytetään tiettyjä haaroja käytettäessä. Täten on mahdollista poistaa tarpeettomat haarat siirtymällä kutsun tai paluun ohessa oikealle haaralle. Bodík, Gupta ja Soffa (1997) menetelmillä saadaan suoritettujen ehdollisten hyppyjen määrää vähennettyä 3-18 prosentilla.

Murthy ym. (2010) tutkii *silmukoiden purkua* vaikutusta laitteistokiihdytetyn ohjelman suorituskykyyn. Silmukoiden purku (engl. *Loop unrolling*) on optimointimenetelmä, jossa iteratiivinen silmukka korvataan usealla kopiolla silmukassa suoritettavasta koodista. Täten silmukan rakenteeseen vaadittavia ehdollisia hyppylauseita ei tarvita. Näin saatuun koodiin voidaan hyödyntää käskujen vuoronnusta ja saavutetaan parempi käskytason rinnakkaisuus.

Tutkimuksen kohteena oli erityisesti kehitellä menetelmät ennustaa ja arvioida silmukan purkamisen tehokkuutta ja löytää optimaalisimmat purkumäärät annetulle ohjelmalle. Tutkimustuloksista kävi ilmi, että tekniikalla osattiin ennustaa jokaisessa testitapauksessa parhaan purkumäärän. Tuloksista huomaa myös silmukoiden purkamisen hyödyn, sillä suurimmassa osassa esitetyistä tuloksista silmukan purulla saavutettiin parempi suorituskyky.

Hyöty vaihteli paljon riippuen ongelman laadusta. Parhaimmassa tapauksessa saavutettiin noin 74.26% nopeutus, kun taas pahimmillaan suorituskyky väheni noin 42.61%. Kuitenkin kaikissa esitetyissä testituloksissa saavutettiin parempi suorituskyky jollain määrällä silmukoiden purkua.

W. Kreahling ym. (2003) ja W. C. Kreahling ym. (2005) tutkii haarojen vähentämistä konditionaalilauseiden yhdistämisellä. Idea on yhdistää usea konditionaalilause yhdeksi ja muuttaa kontrollivuo sellaiseen muotoon, että yleisimmin kuljettu polku on suoraviivaisin. Myös osittaista silmukan purkua hyödynnetään, jolloin usemamman iteraation konditionaalilauseet voidaan yhdistää. Esimerkiksi useamman muuttujan erisuuruus nollasta voidaan tarkastaa ketjuttamalla bittitason and-operaattoreita ja lopuksi vertaamalla yhtäsuuruutta nollaan.

Sekä yhden, että usean muuttujan sisältävien konditionaalilauseiden yhdistämisellä saatiin suoritettujen ehdollisten hyppykäskyjen määrää vähennettyä noin 15.81%. Myöskin suori-

tettujen käskyjen määrä väheni keskimäärin 5.74%. Tehtyjen hyppyjen määrä väheni 20.18 prosentilla, mikä on varsin suotuisaa käskyjen liukuhihnalle.

Vespa, Bauman ja Wells (2015) lähestyy ongelmaa varsin erilaiselta kannalta. Idea on poistaa if-lauseiden luomat haarat kokonaan hyödyntäen algebrallisia lauseita. Esimerkiksi lauseke

```
if (p) {
  x=a;
} else {
  x=b
}
```

saadaan muutettua muotoon

$$x = p \times a + (1 - p) \times b,$$

joka sievenee muotoon

$$x = p \times (a - b) + b.$$

Monimutkaisemmatkin rakenteet kuten sisäkkäiset if-lauseet on mahdollista korvata käyttäen dokumentissa kehitettyä metodia.

Testituloksissa huomattiin metodin nopeauttavan suoritettavaa ohjelmaa 50-200% prosenttia (Vespa, Bauman ja Wells 2015). Metodi on erityisen tehokas SIMD-pohjaiselle arkkitehtuurille. On huomioitavaa, että metodi ei sovellu algoritmeille, jotka hyödyntävät laajasti sivuvaikutuksia.

5 Muita menetelmiä

Luvuissa 3.2 ja 4.2 mainittujen optimointitekniikoiden lisäksi on paljon muita optimointitekniikoita. Tässä luvussa esitellään muutamia muita mielenkiintoisia menetelmiä. Myöskin Hijma ym. (2023) artikkelissa esiintyy paljon tässä tutkielmassa mainitsemattomia menetelmiä.

Säikeiden välisen käyttöasteen maksimoimiseksi on olemassa erilaisia työnjakoalgoritmeja. Tehtäviä voi jakaa joko päälaite, kuten Chen ym. (2010) tapauksessa, tai hajautetusti käyttäen globaalia muistia (Tzeng, Patney ja Owens 2010). Hajautettuun työnjakoon on olemassa useita menetelmiä, kuten työn lahjoitus (engl. *Task donation*) ja työn varastus (engl. *Task stealing*), ja Tzeng, Patney ja Owens (2010) algoritmi hyödyntää molempia menetelmiä.

Päälaitteen suorittamia työnjakoalgoritmeja on lukemattomia ja ne ovat usein sovelluskohtaisia. Esimerkiksi OpenCL:ää hyödyntävässä yleislaskentasovelluksessa työnjako on ohjelmoijan itse määrittelemä.

Vektorisointi on myös yksi yleinen optimointimenetelmä. Vektorisoinnissa muutetaan joukko skalaarioperaatioita vektorioperaatioiksi (Nuzman, Rosen ja Zaks 2006). Näitä voidaan suorittaa esimerkiksi SIMD - laitteistolla. Eichenberger, Wu ja O'Brien (2004) tutkii vektorisointia tilanteissa, joissa data ei ole triviaalisti asemoitu muistiin vektorioperaatioita varten. (Nuzman, Rosen ja Zaks 2006) tutkii vektorisointia, kun data on muistissa limittäin paremman lokaalisuuden vuoksi.

Monitahoinen malli on optimoinnissa yleisesti hyödynnetty matemaattinen malli. Mallia hyödynnetään sisäkkäisen silmukoiden rinnakkaistamiseen ja tiedon parempaan lokaalisuuteen. Monitahoisen mallin idea on muodostaa operaatioista ja riippuvuuksista graafi, jota voidaan manipuloida hyödyntäen affiineja kuvauksia. Täten on mahdollista muodostaa algoritmi, joka on paremmin rinnakkaistettavissa, tai jossa on parempi tiedon lokaalisuus. Esimerkiksi Kong ym. (2013) artikkelissa hyödynnetään monitahoisia malleja sekä vektorisointia SIMD-pohjaisen koodin muodostamiseen.

Yksi varsin mielenkiintoinen tekniikka on hyödyntää päälaitteen keskussuoritinta algorit-

missa. Näin voidaan suorittaa ongelman sarjallinen osa nopeammalla laitteella. Esimerkiksi Fan ym. (2015) kuvan terävöitysalgoritmi hyödyntää keskusprosessoria osaan algoritmin vaiheista.

6 Yhteenveto

Rinnakkaisen laskennan ohjelmissa suurin pullonkaula on muistin käyttö. Täten kannattaa keskittyä järkevään muistinkäyttöön ja suosia nopeampaa muistia aina, kun on mahdollista. Myös hyvä lokaalisuus on hyvä pitää mielessä. Kannattaa myös hyödyntää yhdistettyä muistinhakua tietoa haettaessa hitaasta muistista.

Algoritmeja suunniteltaessa on myös varsin suotuisaa suunnitella algoritmit mahdollisimman suoraviivaisiksi. Vähähaarainen tai täysin haaraton koodi saa suurimman hyödyn käskytason rinnakkaisuudesta. Silloin jää kääntäjälle parempi mahdollisuus suorittaa erilaisia optimointeja.

On myöskin hyödyllistä pitää mielessä laitteiston rakenne ja sen mukana tulevat haasteet. Esimerkiksi SIMD - pohjaisen laitteen kanssa on erityisen suotuisaa suunnitella vähähaaraisia algoritmeja. Korkeamuistisen laitteen kanssa voi olla hyödyllistä siirtää paljon informaatiota paikalliseen muistiin.

On myös viisasta käyttää erilaisia profilointityökaluja mahdollisten pullonkaulojen löytämiseksi. Samalla voi löytää kohtia, jossa optimoinnilla voi saada varsin suurta hyötyä. Profilointia voi myös hyödyntää erilaisten optimointitekniikoiden hyperparametrien asettamiseen.

Lähteet

Bodík, Rastislav, Rajiv Gupta ja Mary Lou Soffa. 1997. “Interprocedural conditional branch elimination”. Teoksessa *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation,* 146–158. PLDI ’97. New York, NY, USA: Association for Computing Machinery. ISBN: 978-0-89791-907-4, viitattu 12. helmikuuta 2024. https://doi.org/10.1145/258915.258929. https://dl.acm.org/doi/10.1145/258915.258929.

Chen, Long, Oreste Villa, Sriram Krishnamoorthy ja Guang R. Gao. 2010. “Dynamic load balancing on single- and multi-GPU systems”. Teoksessa *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS),* 1–12. ISSN: 1530-2075. Huhtikuu. Viitattu 29. huhtikuuta 2024. https://doi.org/10.1109/IPDPS.2010.5470413. https://ieeexplore.ieee.org/abstract/document/5470413.

Eichenberger, Alexandre E., Peng Wu ja Kevin O’Brien. 2004. “Vectorization for SIMD architectures with alignment constraints”. *SIGPLAN Not.* 39 (6): 82–93. ISSN: 0362-1340, viitattu 29. huhtikuuta 2024. https://doi.org/10.1145/996893.996853. https://dl.acm.org/doi/10.1145/996893.996853.

Fan, Mengran, Haipeng Jia, Yunquan Zhang, Xiaojing An ja Ting Cao. 2015. “Optimizing Image Sharpening Algorithm on GPU”. Teoksessa *2015 44th International Conference on Parallel Processing,* 230–239. ISSN: 0190-3918. Syyskuu. Viitattu 30. huhtikuuta 2024. https://doi.org/10.1109/ICPP.2015.32. https://ieeexplore.ieee.org/document/7349578.

Filipovič, Jiří, Matúš Madzin, Jan Fousek ja Luděk Matyska. 2015. “Optimizing CUDA code by kernel fusion: application on BLAS” [kielellä en]. *J Supercomput* 71, numero 10 (lokakuu): 3934–3957. ISSN: 1573-0484, viitattu 28. huhtikuuta 2024. https://doi.org/10.1007/s11227-015-1483-z. https://doi.org/10.1007/s11227-015-1483-z.

Herlihy, Maurice, ja J. Eliot B. Moss. 1993. “Transactional memory: architectural support for lock-free data structures”. Teoksessa *Proceedings of the 20th annual international symposium on computer architecture,* 289–300. ISCA ’93. New York, NY, USA: Association for Computing Machinery. ISBN: 978-0-8186-3810-7, viitattu 29. maaliskuuta 2024. https://doi.org/10.1145/165123.165164. https://dl.acm.org/doi/10.1145/165123.165164.

Hijma, P., S. Heldens, A. Sclocco, B. Van Werkhoven ja H.E. Bal. 2023. “Optimization Techniques for GPU Programming” [kielellä English]. *ACM Computing Surveys* 55 (11). ISSN: 0360-0300. https://doi.org/10.1145/3570638.

Khan, A., M. Al-Mouhamed, A. Fatayar, A. Almousa, A. Baqais ja M. Assayony. 2014. “Padding free bank conflict resolution for CUDA-based matrix transpose algorithm”. Teoksessa *15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD),* 1–6. Kesäkuu. Viitattu 9. huhtikuuta 2024. https://doi.org/10.1109/SNPD.2014.6888709. https://ieeexplore.ieee.org/document/6888709.

Kivioja, Markus, Sanna Mönkölä ja Tuomo Rossi. 2022. “GPU-accelerated time integration of Gross-Pitaevskii equation with discrete exterior calculus” [kielellä eng]. Accepted: 2022-08-17T07:53:39Z Publisher: Elsevier BV, *Computer Physics Communications* 278. Viitattu 22. huhtikuuta 2024. https://doi.org/10.1016/j.cpc.2022.108427. https://jyx.jyu.fi/handle/123456789/82636.

Kong, Martin, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet ja P. Sadayappan. 2013. “When polyhedral transformations meet SIMD code generation”. Teoksessa *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation,* 127–138. PLDI ’13. New York, NY, USA: Association for Computing Machinery. ISBN: 978-1-4503-2014-6, viitattu 29. huhtikuuta 2024. https://doi.org/10.1145/2491956.2462187. https://dl.acm.org/doi/10.1145/2491956.2462187.

Kreahling, William, David Whalley, Mark Bailey, Xin Yuan, Gang-Ryung Uh ja Robert van Engelen. 2003. “Branch Elimination via Multi-variable Condition Merging” [kielellä en]. Teoksessa *Euro-Par 2003 Parallel Processing,* toimittanut Harald Kosch, László Böszörményi ja Hermann Hellwagner, 261–270. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. ISBN: 978-3-540-45209-6. https://doi.org/10.1007/978-3-540-45209-6_40.

Kreahling, William C., David Whalley, Mark W. Bailey, Xin Yuan, Gang-Ryung Uh ja Robert van Engelen. 2005. “Branch elimination by condition merging” [kielellä en]. _Eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.627, *Software: Practice and Experience* 35 (1): 51–74. ISSN: 1097-024X, viitattu 7. huhtikuuta 2024. https://doi.org/10.1002/spe.627. https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.627.

Lee, Jaekyu, Hyesoon Kim ja Richard Vuduc. 2012. “When Prefetching Works, When It Doesn’t, and Why”. *ACM Trans. Archit. Code Optim.* 9 (1): 2:1–2:29. ISSN: 1544-3566, viitattu 24. huhtikuuta 2024. https://doi.org/10.1145/2133382.2133384. https://dl.acm.org/doi/10.1145/2133382.2133384.

Mueller, Frank, ja David B. Whalley. 1995. “Avoiding conditional branches by code replication”. *SIGPLAN Not.* 30, numero 6 (kesäkuu): 56–66. ISSN: 0362-1340, viitattu 4. helmikuuta 2024. https://doi.org/10.1145/223428.207116. https://dl.acm.org/doi/10.1145/223428.207116.

Murthy, Giridhar Sreenivasa, Mahesh Ravishankar, Muthu Manikandan Baskaran ja P. Sadayappan. 2010. “Optimal loop unrolling for GPGPU programs”. Teoksessa *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS),* 1–11. ISSN: 1530-2075. Huhtikuu. Viitattu 6. helmikuuta 2024. https://doi.org/10.1109/IPDPS.2010.5470423. https://ieeexplore.ieee.org/document/5470423.

Myllykoski, Mirko. 2015. “On GPU-accelerated fast direct solvers and their applications in image denoising” [kielellä eng]. Accepted: 2015-09-01T12:01:03Z ISBN: 9789513962777 Publisher: University of Jyväskylä, *Jyväskylä studies in computing,* numero 218, viitattu 29. tammikuuta 2024. https://jyx.jyu.fi/handle/123456789/46733.

Nuzman, Dorit, Ira Rosen ja Ayal Zaks. 2006. “Auto-vectorization of interleaved data for SIMD”. *SIGPLAN Not.* 41 (6): 132–143. ISSN: 0362-1340, viitattu 29. huhtikuuta 2024. https://doi.org/10.1145/1133255.1133997. https://dl.acm.org/doi/10.1145/1133255.1133997.

Ramamoorthy, C. V., ja H. F. Li. 1977. “Pipeline Architecture”. *ACM Comput. Surv.* 9 (1): 61–102. ISSN: 0360-0300, viitattu 2. huhtikuuta 2024. https://doi.org/10.1145/356683.356687. https://dl.acm.org/doi/10.1145/356683.356687.

Ryoo, Shane, Christopher I. Rodrigues, Sam S. Stone, John A. Stratton, Sain-Zee Ueng, Sara S. Baghsorkhi ja Wen-mei W. Hwu. 2008. “Program optimization carving for GPU computing”. *Journal of Parallel and Distributed Computing,* General-Purpose Processing using Graphics Processing Units, 68, numero 10 (lokakuu): 1389–1401. ISSN: 0743-7315, viitattu 5. maaliskuuta 2024. https://doi.org/10.1016/j.jpdc.2008.05.011. https://www.sciencedirect.com/science/article/pii/S0743731508000968.

The OpenCL™ Specification. 2023. Specification, joulukuu. Viitattu 27. maaliskuuta 2024. https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html.

Tzeng, Stanley, Anjul Patney ja John D. Owens. 2010. “Task Management for Irregular-Parallel Workloads on the GPU” [kielellä en], viitattu 29. huhtikuuta 2024. https://doi.org/10.2312/EGGH/HPG10/029-037. https://escholarship.org/uc/item/9r15d4zk.

Wang, Guibin, YiSong Lin ja Wei Yi. 2010. “Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU”. Teoksessa *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing,* 344–350. Joulukuu. Viitattu 28. huhtikuuta 2024. https://doi.org/10.1109/GreenCom-CPSCom.2010.102. https://ieeexplore.ieee.org/abstract/document/5724850.

Vespa, Lucas, Alexander Bauman ja Jenny Wells. 2015. “Algorithm Flattening: Complete branch elimination for GPU requires a paradigm shift from CPU thinking”. Teoksessa *2015 IEEE High Performance Extreme Computing Conference (HPEC),* 1–6. Syyskuu. Viitattu 9. helmikuuta 2024. https://doi.org/10.1109/HPEC.2015.7322477. https://ieeexplore.ieee.org/document/7322477.

Yang, Yi, Ping Xiang, Jingfei Kong ja Huiyang Zhou. 2010. “A GPGPU compiler for memory optimization and parallelism management”. *SIGPLAN Not.* 45 (6): 86–97. ISSN: 0362-1340, viitattu 22. huhtikuuta 2024. https://doi.org/10.1145/1809028.1806606. https://dl.acm.org/doi/10.1145/1809028.1806606.