Jari Isohanni

GPU-laskennan optimointi

Tietotekniikan pro gradu –tutkielma

Mobiilijärjestelmät

16. marraskuuta 2013

Jyväskylän yliopisto

Tietotekniikan laitos

Kokkolan yliopistokeskus Chydenius

Tekijä: Jari Isohanni

Yhteystiedot: jari.isohanni@gmail.com

Ohjaaja: Risto Honkanen

Työn nimi: GPU-laskennan optimointi

Title in English: Optimization of GPU-computing

Työ: Pro gradu -tutkielma

Suuntautumisvaihtoehto: Mobiilijärjestelmät

S**ivumäärä:** 92+10

Tiivistelmä: Näytönohjaimet, grafiikkasuorittimet, tarjoavat rinnakkaisen laskennan alustan, jossa voidaan suorittaa ohjelmakoodia satojen ydinten toimesta. Tämä alusta mahdollistaa matemaattisesti työläiden ongelmien ratkaisemisen tehokkaasti. Grafiikkasuorittimen rinnakkainen suoritusympäristö kuitenkin eroaa suuresti tietokoneen suorittimen peräkkäisestä suoritusympäristöstä. Ongelmien ratkaisemiseksi tehokkaasti rinnakkaisympäristössä on noudettava ohjelmointimenetelmiä, jotka soveltuvat erityisesti rinnakkaisympäristöön. Tässä työssä tarkastellaan rinnakkaisen laskennan perusteita, miten erilaiset ohjelmointimenetelmät vaikuttavat ohjelman suoriutumiseen grafiikkasuorittimella sekä miten voidaan saavuttaa nopein mahdollinen suoritusaika ohjelmalle grafiikkasuorittimella. Työssä laadittiin teoreettinen parametroitu malli grafiikkasuorittimen laskenta-ajan arviointiin. Lisäksi toteutettiin kaksi erilaista matriisikertolaskuja suorittavaa rinnakkaisen laskennan ohjelmaa. Työn tuloksissa verrataan teoreettista suoritusaika-arviota käytännössä saavutettuihin tuloksiin sekä esitetään grafiikkasuorittimella suoritettavilla ohjelmilla saavutettuja nopeutuksia eri parametriarvoilla.

Avainsanat: Grafiikkasuoritin, näytönohjain, CUDA, GPU, optimointi, rinnakkainen laskenta

Abstract: Graphics processing units offer platform that consist of hundreds of cores for programs to use in parallel computing. This platform can be used to solve efficiently mathematically heavy problems. Running programs in parallel computing environment differs

from running programs in serial environment. In order to achieve speedup from parallel environment, programmer must use programming methods suitable to parallel environment. In this work we explain some basics of parallel computing and examine how programming and optimization practices affect programs running in graphics processing unit. We also created our own theoretical parameterized model which is used to estimate performance of our GPU-programs. Our model is used to estimate performance of two different matrix multiplication programs, which are run in parallel environment. In results we compare theoretical performance values, calculated by using our own model, to ones measured in real run environment. Also we present how different parameters affect to gained speedup when using our GPU-programs.

Keywords: Graphics processing unit, GPU, CUDA, optimization, parallel computing

Termiluettelo

ALU	Arithmetic Processing Unit
BSP	Bulk-Synchronous Parallel
CU	Control Unit
CPU	Central Processing Unit
CTM	Close to Metal
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
FMA	Fused Multiply–Add
FPU	Floating Point Unit
GPU	Graphics Processor Unit
GPGPU	General Purpose computing on graphics processing unit
MPI	Message Passing Interface
MPIF	Message Passing Interface Forum
SFU	Special Function Unit
SIMD	Single Instruction, Multiple-Data
SIMT	Single-Instruction, Multiple-Thread
SPMD	Single-Program, Multiple-Data
PRAM	Parallel Random Access Machine
PVM	Parallel Virtual Machine
RAM	Random Access Machine

Sisältö

1 Johdanto

Tässä tutkielmassa käsitellään rinnakkaisen laskennan toteuttamista grafiikkasuorittimen avulla. Nykyaikainen grafiikkasuoritin rakentuu sadoista suoritinytimistä, jotka voivat suorittaa omaa käskyjonoaan itsenäisesti. Tämä tekee grafiikkasuorittimesta tehokkaan rinnakkaisen laskennan järjestelmän. Suoritinydinten avulla matemaattisesti haastavien, ja rinnakkaiseen suoritukseen sopivien ongelmien ratkaiseminen grafiikkasuorittimen avulla on huomattavasti tehokkaampaa kuin varsinaisella tietokoneen suorittimella. Kuitenkin vääräntyyppisellä, esim. paljon haarautumia sisältävällä, ongelmalla tai ongelman ratkaisulla grafiikkasuorittimen hyödyt jäävät olemattomiksi tai ohjelma voi toimia hitaammin kuin on peräkkäisesti toteutettuna. Tässä työssä tuodaan esille menetelmiä, joiden avulla grafiikkasuorittimesta saadaan mahdollisimman suuri nopeutus ongelman ratkaisemiseen.

Työssä esitellään aluksi malleja ja teorioita, joiden pohjalta rinnakkaisen laskennan paradigmoja voidaan suunnitella. Sekä tarkastellaan miten rinnakkaisen laskennan suorittamisen tehokkuutta voidaan arvioita ja eri algoritmeja tai järjestelmiä vertailla keskenään. Tämä tapahtuu käsittelemällä muutamia laajasti käytettyjä tai muuten merkittäviä rinnakkaisen laskennan malleja. Rinnakkaisen laskennan mallit jaetaan työssä osa-alueisiin riippuen siitä, millaisessa järjestelmässä ne toimivat tai mikä niiden käyttötarkoitus on.

Mallien käsittelyn jälkeen tarkastellaan varsinaista grafiikkasuorittimella suoritettavaa laskentaa ja grafiikkasuorittimen fyysistä rakennetta. Grafiikkasuorittimen fyysinen rakenne on osittain vain pintapuolista tarkastelua, sillä osa rakenteesta kuuluu liikesalaisuuksien piiriin. Näin ollen joudutaan osittain turvautumaan arvioihin ja suorituskykymittauksiin. Työ tarkastelee NVIDIAn kehittämää Fermi-arkkitehtuuria, joka on käytetyin grafiikkasuoritinarkkitehtuuri työn kirjoittamisen hetkellä. Grafiikkasuorittimen ohjelmoinnissa perehdytään niin ikään NVIDIAn kehittämään CUDA-ohjelmointiympäristöön. CUDA on grafiikkasuorittimille suunniteltu ohjelmointikieli ja suoritusympäristö. CUDA-ohjelmointikieli tarjoaa matalan kynnyksen ohjelmoijalle siirtyä perinteisestä peräkkäisohjelmoinnista grafiikkasuorittimella suoritettavaan rinnakkaiseen laskentaan. CUDA-ohjelmoinnista käydään läpi perusasioita sekä tarkastellaan CUDA-ohjelman rakennetta ja

sen suoriutumista grafiikkasuorittimella. Lisäksi perehdytään ominaisuuksiin joista ohjelmoijan tulee olla tietoinen, jotta ohjelma toimii tehokkaasti grafiikkasuorittimella.

Varsinainen tutkimustyö esitetään luvussa 4. Luvussa käsitellään menetelmiä, joiden avulla voidaan arvioida algoritmien suoritusaikaa grafiikkasuorittimella. Lisäksi luvussa 4 esitellään menetelmiä, joiden avulla suoritusaika saadaan mahdollisimman nopeaksi. Ratkaistavana ongelmana tutkimustyössä toimii neliömatriisien kertolaskun suorittaminen. Tutkimustyö tarkastelee miten ohjelma tulisi rakentaa, jotta se voi hyötyä parhaiten grafiikkasuorittimen tarjoamista resursseista kuten ytimistä ja erilaisista muisteista. Tutkimustyössä tutkitaan kahden erilaisen ohjelman suorituskykyeroja. Toinen käyttää grafiikkasuorittimen globaalimuistia ja toinen jaettua muistia. Molemmista ongelmista pyritään rakentamaan rinnakkaisen laskennan malli, jolla suorituskykyä voidaan ennustaa parametrein. Parametrien avulla pyritään löytämään optimaaliset suoritusarvot, joiden avuilla käytettävä grafiikkasuoritin suorittaa ohjelman mahdollisimman nopeasti.

Työn lopussa vertaillaan teoreettisesti laskettuja tuloksia oikeassa ympäristössä mitattuihin tuloksiin. Tuloksissa pyritään vertailemaan teoreettisia malleja ja käytännön suorituskykyä keskenään sekä löytämään syitä niiden eroihin ja yhteneväisyyksiin. Lisäksi tarkastellaan eri parametrien vaikutusta ohjelman suoritusaikaan oikeassa suoritusympäristössä. Oikeassa suoritusympäristössä pyritään löytämään parametrit, joiden avulla grafiikkasuorittimesta saadaan paras mahdollinen nopeutus ohjelman suoritusaikaan.

Työn on jäsennelty niin, että luku 2 esittelee rinnakkaisen laskennan malleja. Luku 3 käsittelee grafiikkasuoritinta ja GPU-laskentaa yleisellä tasolla. Luvussa 4 käsitellään GPU-laskennan optimointia. Luku 5 esittelee saavutettuja tuloksia ja luku 6 sisältää yhteenvedon työstä.

2 Rinnakkaisen laskennan malleista

Rinnakkaisen laskennan malleilla pyritään mallintamaan rinnakkaisen laskennan suorituskykyä. Rinnakkaisen laskennan mallit ovat usein korkean tason abstrakteja matemaattisia malleja. Abstrakteilla malleilla pyritään yksinkertaistamaan rinnakkaisen laskennan algoritmien kehitystyötä. Yksinkertaistaminen tapahtuu piilottamalla vähemmän tärkeitä yksityiskohtia laitteistopuolen toteutuksesta. Rinnakkaisen laskennan malli pyrkii luomaan yhteisen alustan, jolla voidaan mallintaa tärkeimpien osien toimintaa sekä mitata luotettavasti laitteiston suorituskykyä. [1]

Rinnakkaisen laskennan mallit muodostuvat yleensä useista parametreista, jotka kuvaavat laskentaa suorittavan järjestelmän eri osa-alueiden suorituskykyä tai suorituskyvyttömyyttä. Parametrien avulla mallilla voidaan laskea kustannusarvio suoritettavalle ohjelmalle. Tämä antaa ohjelmoijalle tietoa koodin monimutkaisuudesta ja arvion ohjelman suoritusajasta. Toisaalta malli voi olla myös ei-parametrisoitu, jolloin mallilla voidaan suunnitella rinnakkaisen laskennan paradigmoja. Jotkin mallit määrittelevät myös kirjastototeutuksia, joiden avulla rinnakkaista laskentaa voidaan suorittaa käytännössä. [2]

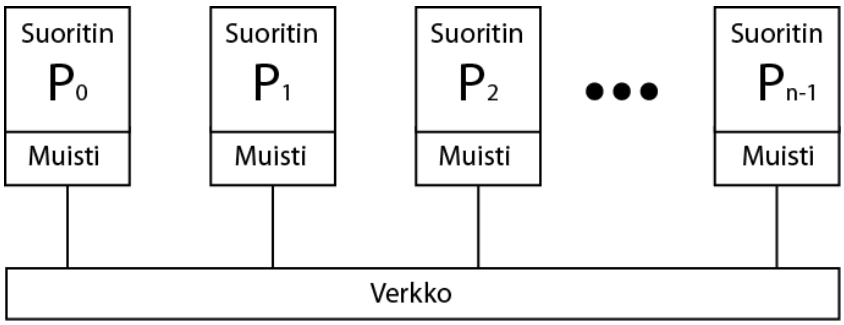
Rinnakkaisen laskennan malleilla voidaan kuvata niin tietokoneen grafiikkasuorittimessa sijaitsevien tietovirtasuorittimien suorituskykyä kuin laajan, hajautettujen tietokoneiden muodostaman, supertietokoneen suorituskykyäkin. Jotkin malleista soveltuvat parhaiten määriteltyihin käyttötarkoituksiin, kuten määrättyyn laiteympäristöön. Mallit voidaan jakaa kahteen laajempaan ryhmään, viestinvälitysmalleihin sekä yhteisen muistin malleihin. Suurimpana erona näiden kahden mallin välillä on muistin toimintatapa. Yhteisen muistin malleissa kaikilla suorittimilla on yhteinen muisti, kun taas viestinvälitysmalleissa jokaisella suorittimella on oma yksityinen muistinsa.

2.1 Viestinvälitysmallit

Viestinvälitysmallissa - kutsutaan myös hajautetun muistin malliksi - jokaisella suorittimella on oma yksityinen muistinsa. Kommunikaatio eri suorittimien muistien välillä tapahtuu viesteillä. Viestien avulla suoritin voi jakaa omaa tietoansa muille suorittimille tai pyy-

tää tietoa muilta suorittimilta. Viestinvälitysmallit usein olettavat suorittimien välisen verkon (interconnect network) hoitavan viestien reitityksen, tästä johtuen verkkoa voidaan käsitellä pisteestä pisteeseen verkkona (point-to-point communication). Pisteestä pisteeseen kommunikaatiossa viestien oletetaan kulkevan suoraan lähettäjältä vastaanottajalle. Kuva 1 esittää havainnollistetun esityksen viestinvälitysmallin laitteistosta. [3]

Viestinvälitysmallien eduiksi katsotaan niiden skaalautuvuus suuremmille määrille suorittimia sekä yksinkertaisempi ja halvempi toteutettavuus, verrattuna jaetun muistin malleihin. Viestinvälitysmallissa ohjelmoijan ei tarvitse huolehtia mahdollisista kilpailutilanteista, joita syntyy jaetun muistin malleissa eri suorittimien yrittäessä kirjoittaa/lukea samaa muistiosoitetta. Haittapuolena suoritin vastaavasti voi joutua odottamaan, että toinen suoritin ehtii lähettämään sille tietoa muististaan. Tämän lisäksi ohjelmointi käyttäen viestinvälitysmallia on usein haastavampaa. Tämä johtuu siitä, että ohjelmoija joutuu mm. ajoittamaan viestien lähetyksen ja paketoimaan/purkamaan tiedon. [4]



Kuva 1. Viestinvälitysmallin mukainen laitteisto [5].

2.1.1 Ei-kustannusparametroidut viestinvälitysmallit

Tässä työssä käsiteltävät ei-kustannusparametroidut viestinvälitysmallit ovat kirjastoja tai niiden määritelmiä. Mallien avulla ohjelmistokehittäjä voida luoda koodia, jota suoritetaan rinnakkaisesti. Malleilla ei voida mitata teoreettisesti rinnakkaisen laskennan suorituskykyä, vaan ne ovat käytännön toteutuksia. Mallit pyrkivät optimoimaan rinnakkaisen laskennan suoritusta parhaaksi näkemällään tavalla ja olemaan samanaikaisesti helposti käyttöönotettavia, tehokkaita ja skaalautuvia.

MPI-mallin (Message Passing Interface) ensimmäinen versio julkaistiin vuonna 1994, sen kehitystyötä vastasivat yli 40:nen eri toimijan, mm. yliopistojen ja laitteistovalmistajien edustajat. MPI-mallin kehittäjät perustivat myös Message Passing Interface Forum:in (MPIF). MPIF vastaa nykyään mallin ylläpidosta ja päivittämisestä. Uusin MPI:n versio 3.0 julkaistiin syyskuussa 2012. MPI-malli määrittelee rajapinnat, jotka toteuttavaa kirjastoa voidaan kutsua MPI-kirjastoksi. MPI-malli ei ota kantaa rajapintojen toteutukseen tai kirjaston ohjelmointikieleen. Tämä antaa sovelluskehittäjälle mahdollisuuden luoda rinnakkaisen laskennan koodia, joka on siirrettävissä eri alustojen MPI-kirjastojen välillä pienellä työmäärällä. [6]

MPI:n tavoitteena on tehokas ja luotettava viestintä, siirrettävyys, sekä säieturvallisuuden varmistaminen. Tehokas viestintä toteutetaan välttämällä muistista muistiin kopiointia, limittämällä laskentaa ja viestintää sekä käyttämällä viestintään erillistä suoritinta. Siirrettävyys saavutetaan määrittelemällä MPI-malli laitteistoriippumattomaksi, jolloin MPI-malli voidaan toteuttaa eri käyttöjärjestelmissä ja laitteistoissa. Laitteistoriippumattomuuden ansiosta MPI-malli toimii myös yhteisen muistin laitteistoissa. Luotettava viestintä tapahtuu MPI-mallin toimesta, jolloin ohjelmoijan ei tarvitse huolehtia viestinvälitysongelmista. Säieturvallisuus varmistetaan, kun jokainen prosessi käsittelee muistialueen tietoja niin, että taataan muiden prosessien turvallinen suorittaminen samanaikaisesti. [6]

MPI-malli muodostuu [7]:

- Joukosta $P = \{P_o, P_{1,\dots,} P_{n-1}\}$, prosesseja, jotka suorittavat varsinaiset laskentatehtävät.
- Viestintäryhmistä (communicator) $C = \{C_o, C_{1,\dots,} C_{n-1}\}$, jotka sisältävät prosessijoukon *P*. Viestintäryhmä hallitsee prosessijoukkoa niin, että saman viestintäryhmän *C* sisällä olevat prosessit voivat kommunikoida keskenään.
- Päästä-päähän verkosta, jossa prosessit voivat lähettää viestejä luotettavasti toisilleen.
- Tietotyypeistä, tietotyyppien avulla MPI-malli vähentää tiedonsiirtoon liittyviä kustannuksia sekä ylläpitää siirrettävyyttä.

MPI-mallissa määritellään mm. päästä-päähän kommunikaation toteutus, tietotyypit, kollektiiviset operaatiot ja prosessit sekä niiden rakenne ja ryhmittely. Malli ei ota kantaa operaatioihin, jotka vaativat laajempaa käyttöjärjestelmä tukea, debug-toimintoihin tai käytettäviin kehitystyökaluihin. MPI-malli on laajasti käytetty ja se on saatavilla C/C++ ja FORTRAN toteutuksin. MPI-mallissa eri käskyjä on käytettävissä yli 100, mutta peruskäytössä riittää kuusi eri käskyä. [7]

MPI-malli toteuttaa luotettavan viestinvälityksen päästä-päähän periaatteella. Viestinvälitys voi olla synkronista tai asynkronista. Molemmissa tapauksissa MPI-malli säilyttää välitettävien viestien järjestyksen niin, että prosessin A lähettäessä viestit 1 ja 2 viestit saapuvat viestit myös vastaanottajalle järjestyksessä 1 ja 2. MPI-malli mahdollistaa prosessien välisen tiedon jakamisen joko viesteillä lähde- ja kohdeprosessin välillä tai muistialueen etäkäytöllä. Etäkäytössä muistialueeseen voidaan tehdä aktiivisia tai passiivisia kutsuja. Jos muistialue on jaettu aktiivisesti, muistialueen omistava prosessi synkronoi muistialueeseen tulevat kutsut. Passiivisella kutsulla voidaan hakea tietoa kohdeprosessin muistialueesta ilman vuorovaikutusta kohdeprosessin kanssa. [8]

MPI-ohjelma muodostuu kolmesta pääosasta 1) MPI-ympäristön alustus, 2) rinnakkaisen laskennan työ ja viestintä sekä 3) MPI-ympäristön tuhoamien. MPI-malli jakaa suorituksen alussa suoritettavan laskentatehtävän T (Task) alitehtäviksi/prosesseiksi (Subtask) $P = \{P_o, P_{1,\dots,} P_{n-1}\}$. Jokaisella alitehtävällä on oma yksilöllinen arvo (Rank), jolla se voidaan tunnistaa. Lisäksi jokaisella alitehtävällä on oma laskentaryhmänsä *C* johon se kuuluu. Jos ohjelmoija ei määritä alitehtävälle laskentaryhmää, kuuluu se automaattisesti `MPI_COMM_WORLD` laskentaryhmään. [6]

Seuraavassa esimerkissä esitetään MPI-mallin kuuden tärkeimmän käskyn käyttöä. Esimerkistä on selkeyttämisen vuoksi jätetty pois muuttujien määrittelyt ja ei MPI-malliin liittyvää koodia. Esimerkin alussa MPI-ympäristö alustetaan `MPI_Init` -komennolla. Tämän jälkeen laskentatehtävä kysyy MPI-ympäristöltä, kuinka monta prosessia kuuluu koko MPI-ympäristöön sekä oman rank-arvonsa. Nämä tapahtuvat komennoilla `MPI_Comm_Size` sekä `MPI_Comm_rank`. Jos prosessin rank-arvo on 0, lähettää proses-

si muille MPI_COMM_WORLD laskentaryhmän suorittimille oman muistialueensa (send_buffer) tiedot MPI_INT -tietotyypissä. Jos rank-arvo ei ole nolla vastaanottaa prosessi omaan muistialueeseensa (receive_buffer) MPI_INT -tietotyyppiä olevan viestin, komennot MPI_Recv ja MPI_Send. Lopuksi MPI-ympäristö tuhotaan MPI_Finalize -kutsulla. [6]

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &me);
if(me == 0)
        MPI_Send (&send_buffer, 1, MPI_INT, 0, tag,
                    MPI_COMM_WORLD);
else
        MPI_Recv (&receive_buffer, 1, MPI_INT, 0, tag,
                    MPI_COMM_WORLD, &status);
MPI_Finalize();
```

MPI-mallin etuina ovat mahdollisuus ajoittaa tehtäviä staattisesti, sekä tehtävien suorittamisen täsmällinen rinnakkaisuus. Viestinvälityksessä MPI-malli takaa luotettavan tiedonsiirron ja mahdollistaa useita eri viestintätapoja. Viestinvälitys toimii MPI-mallissa myös synkronointikohtana, jolloin erillistä synkronointia ei välttämättä tarvita. Lisäksi MPI-malli pystyy hyödyntämään laitteisto-ominaisuudet täysmittaisesti, sillä MPI-kirjasto luodaan jokaiselle käyttöjärjestelmälle ja laitteistolle erikseen. [9]

MPI-mallin haittapuoliksi voidaan vastaavasti laskea mahdolliset suuret viiveet globaaleissa operaatioissa sekä viestienvälityksessä. Ohjelmoijan kannalta on haasteellista määrittää tehtäville oikea koko. Tehtävien liian pieni koko voi johtaa siihen, että suorittimet joutuvat lähettämään tietoa jatkuvasti toisilleen mikä saattaa tukkia viestinvälityksen. Lisäksi MPI-mallissa dynaamisen kuormituksen hallinta on vaikeaa. [9]

MPI-mallista on saatavilla useita toteutuksia mm. eri laitteistovalmistajien kuten HP:n, IBM:n ja Intelin tekeminä. Lisäksi saatavilla on ilmainen toteutus osoitteesta: http://www.mcs.anl.gov/research/projects/mpi/

PVM-malli on rinnakkaisen laskennan toteutus, jolla useita suorittimia/tietokoneita voidaan liittää yhdeksi rinnakkaisen laskennan koneeksi. PVM on kehitetty Yhdysvalloissa Oak Ridgen kansallisessa tutkimuslaitoksessa ja julkaistu ensimmäisen kerran marraskuussa 1991. PVM-mallissa jokaisella suorittimella on oma yksityinen muistinsa ja kommunikointi suorittimien välillä tapahtuu viesteillä. [10]

PVM-malli perustuu seuraaviin ominaisuuksiin: muutettavaan suoritinjoukkoon, laitteistotason ominaisuuksien hyödyntämiseen, tehtäväpohjaiseen suoritukseen, täsmälliseen viestinvälitykseen sekä kattavaan laitteistotukeen. Muutettavalla suoritinjoukolla ohjelmoija voi valita mitkä suorittimet osallistuvat laskennan suorittamiseen. Muutettava suoritinjoukko on myös tärkeä osa PVM-mallin vikasietoisuutta. Laitteistotason ominaisuuksilla ohjelmoija voi hyödyntää laitteiston erityispiirteitä ja valita parhaiten sopivan laitteiston suorittamaan valitun laskentatehtävän. Tehtäväpohjaisessa suorituksessa tehtävät suoritetaan itsenäisesti peräkkäisten käskyjen sarjoina, jolloin yksi prosessi voi sisältää monta tehtävää ja monta tehtävää voidaan suorittaa samalla suorittimella. Täsmällinen viestinvälitys antaa tehtäville mahdollisuuden kommunikoida viesteillä suoraan keskenään. Viestien kokoa rajoittaa vain käytettävissä oleva muisti. Kattava laitteistotuki mahdollistetaan kun PVM-malli hoitaa viestien välittämisen laitteiden välillä, jotka toimivat eri tiedonesitystavoilla. [10]

PVM toimii suoritusympäristönä rinnakkaisen laskennan järjestelmässä. Tämän ansioista ohjelman ei tarvitse huolehtia viestienvälityksestä, tietotyyppimuunnoksista tai tehtävien aikatauluttamisesta. PVM-ympäristössä suoritettava ohjelma muodostuu sarjoista tehtäviä, jotka ratkaisevat annettua ongelmaa yhdessä. Jokainen tehtävä pystyy PVM-ympäristön kautta kommunikoimaan muiden tehtävien kanssa, sekä käynnistämään ja pysäyttämään tehtäviä. PVM:llä voidaan suorittaa joko funktionaalista rinnakkaisuutta tai tietorinnakkaisuutta. Funktionaalisessa rinnakkaisuudessa jokaisella tehtävällä on oma funktionaalinen tarkoituksensa. Tehtävä voi esimerkiksi toimi sisään- tai uloskirjoittajana tai tehdä lasken-

taa. Tietorinnakkaisuudessa jokainen tehtävä suorittaa samaa käskyjonoa, mutta jokainen tehtävä suoritetaan eri osalle tietojoukkoa. Tätä suoritustapaa kutsutaan SPMD:ksi (single-program multiple-data). [11]

Varsinainen PVM-suoritusympäristö koostuu [10]:

- Pvmd3-ohjelmasta, joka toimii jokaisessa rinnakkaiseen laskentaan osallistuvassa tietokoneessa.
- PVM-kirjastosta, joka sisältää varsinainen rinnakkaisen laskennan hallintaan liittyvän koodin.
- PVM-ohjelmasta, joka on kokoelma peräkkäisiä käskyjä, jotka jaetaan suoritettavaksi eri rinnakkaisen järjestelmän koneissa.
- Lisäksi isäntäkoneella, joka hallitsee rinnakkaisen laskennan suorittamista, käytetään PVM-konsolia.

Jokainen PVM-ohjelma käännetään erikseen jokaiselle tietokoneelle/suorittimelle ja siirretään suoritettavaksi omalle tietokoneelleen/suorittimelle. Yksi ohjelma toimii käynnistävänä ohjelmana ja se käynnistetään manuaalisesti PVM-konsolista. Muissa tietokoneissa toimivat ohjelmat käynnistyvät automaattisesti käynnistävän ohjelman toimesta. Jokainen ohjelma voi käynnistää uusia tehtäviä, liittää niihin suoritettavan ohjelman, lähettää tietoa muille ohjelmille ja vastaanottaa tietoa muilta ohjelmilta. Lisäksi jokaisella ohjelmalla on oma yksilöllinen tunnus sekä tieto isäntäohjelman tunnuksesta. [11].

PVM-virtuaalikoneen tietokoneet yhdistyvät toisiinsa niin, että ne voivat kommunikoida keskenään TCP- ja UDP-protokollien kautta. TCP-protokollaa PVM-virtuaalikone käyttää, kun prosessi kommunikoi samassa laitteistossa olevan pvmd3-ohjelman kanssa. UDP-protokollaa käytetään kommunikoitaessa verkon yli. Varsinainen kommunikointi tietokoneiden välillä tapahtuu asynkronisesti. Tällöin lähettäjä luovuttaa viestin verkon välitettäväksi, mutta ei jää odottamaan vastausta vaan jatkaa omaa työtään. Verkko huolehtii viestin perille menosta ja siitä, että samalta lähettäjältä samalla vastaanottajalle lähetyt viestit säilyttävät järjestyksensä. [12]

Seuraavassa esimerkissä käydään läpi muutamia PVM-mallin peruskäskyjä. Esimerkin PVM-ohjelma kysyy PVM-ympäristöstä aluksi oman tunnuksensa `pvm_mytid` -käskyllä sekä isäntäohjelman tunnuksen `pvm_parent` –käskyllä. Tämän jälkeen ohjelma alustaa tiedonlähetyksen `pvm_initsend` –käskyllä ja paketoi omasta muistialueestaan (`buf`) tekstijonotyyppisen tiedon lähetystä varten, `pvm_pkstr` –kutsu. Viesti lähetetään `pvm_send` –kutsulla isäntäohjelmalle. Lähetyksen jälkeen ohjelma vastaanottaa paketin isäntäohjelmalta `pvm_recv` –kutsulla. Vastaanoton jälkeen ohjelma purkaa vastaanotetun paketin, joka sisältää kokonaislukutyyppistä tietoa muistialueeseen (`result`). Ohjelma lopettaa suorituksen `pvm_exit` –käskyllä. [10]

```
myTID = pvm_mytid();
ptid = pvm_parent();
pvm_initsend(PvmDataDefault);
pvm_pkstr(buf);
pvm_send(ptid, 1);
pvm_recv(ptid, 0);
pvm_upkint(&result, 1, 1);
pvm_exit();
```

PVM-mallin suurin etu on sen heterogeenisyys, jolloin PVM-ympäristö voidaan muodostaa keskenään erilaisista laitteistoista, tämä vaikuttaa merkittävästi mm. järjestelmän kustannuksiin. Dynaamisen prosessien hallinnan vuoksi PVM-ympäristö on vikasietoinen. Muihin malleihin verrattuna PVM-malli on helppo asentaa, konfiguroida ja ohjelmoida. Ohjelmoitavuutta helpottaa mm. edellä mainittu dynaaminen prosessien hallinta sekä automaattiset tietotyyppimuunnokset eri laitteistojen välillä. [11]

Toisaalta heterogeenisyyden vuoksi PVM-malli ei pysty täysin hyödyntämään kaikkea laitteiston suorituskykyä. PVM-mallissa työ jaetaan staattisesti prosesseille, jolloin jotkin prosessit voivat ylikuormittua. [13]:

PVM-ympäristö on saatavilla useille eri käyttöjärjestelmille osoitteesta: http://www.netlib.org/pvm3/index.html

2.1.2 Kustannusparametroidut viestinvälitysmallit

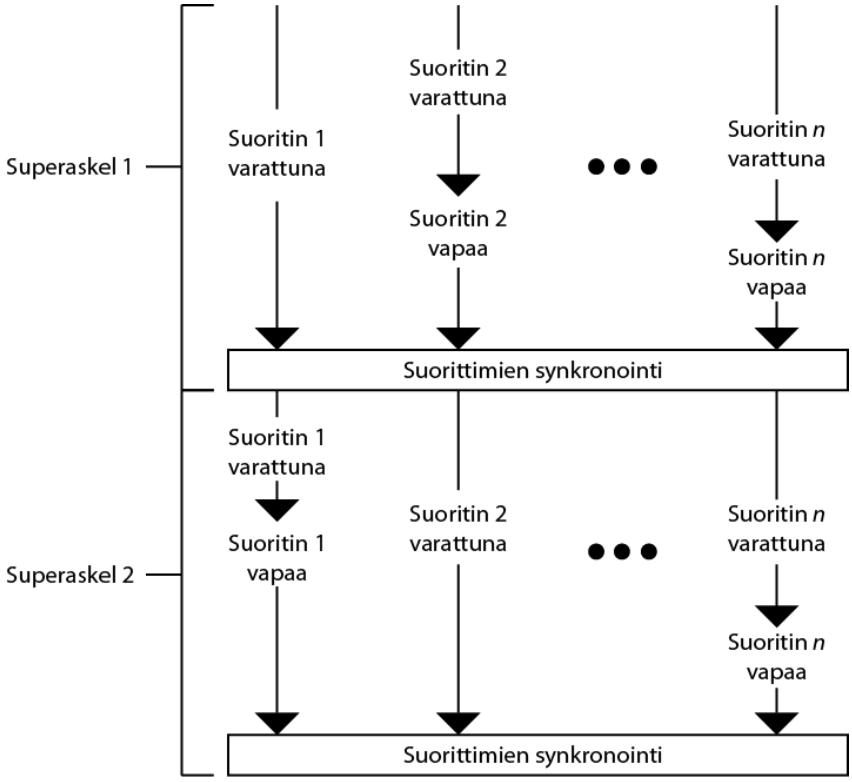
Kustannusparametroidut mallit ovat malleja, joilla voidaan mallintaa rinnakkaisen laskennan suoritusta ja ennustaa sen suorituskykyä. Kustannusparametroiduilla malleilla, muuttelemalla eri parametrien arvoja, voidaan mallintaa algoritmien tehokkuutta eri laitteistoissa. Tässä luvussa käsitellään BSP-mallia ja LogP-mallia.

BSP-malli on laskennallinen mallin rinnakkaisen laskennan paradigmojen suoritusaikojen arviointiin. BSP-mallissa parametreilla pyritään kuvaamaan ohjelmaa suorittavan laitteiston ominaisuuksia. Malli on kehitetty vuonna 1990 Leslie Valiantin toimesta. Valiantin mukaan mallin tulee olla yksinkertainen eikä keskittyä liikaa yksityiskohtiin. [1]

BSP-mallin määrittää monisuorittimen joka muodostuu [14]:

- Joukosta $P = \{P_o, P_{1,\dots,}P_{n-1}\}$ suorittimia, jotka suorittavat lasku- ja/tai muistioperaatioita.
- Verkosta, joka toimittaa viestejä suorittimien välillä.
- Synkronoijasta, joka synkronoi suorittimien ja verkon tilan.

BSP-mallissa laskennan suoritus jaetaan superaskeliin (superstep). Jokainen superaskel on jaettu kolmeen vaiheeseen. Ensimmäisen vaiheen aikana suoritin voi lukea ja kirjoittaa paikallista dataa. Toisessa vaiheessa suorittimet voivat lähettää ja vastaanottaa tietoa. Kolmannessa vaiheessa suorittimet synkronoidaan. Globaalin viestinnän vaiheessa suoritin voi lähettää tietoa muille suorittimille. Lähetetyt tiedot ovat vastaanottavan suorittimen saatavilla seuraavan superaskeleen alussa. Synkronointivaiheessa ei-valmiit suorittimet saavat uuden superaskeleen aikaa suorittaa tehtävänsä loppuun. Sillä aikaa kun ei-valmiit suorittimet suorittavat käskynsä loppuun muut suorittimet odottavat. Kuvassa 2 on esitetty BSP-mallin mukainen superaskeleen suoritus. [15]



Kuva 2. BSP-mallin superaskeleen suoritus [1].

BSP-mallin mukainen superaskeleen suoritusaika määräytyy kolmesta parametrista [14]:

- w = suurin paikallisesti suoritettava työ (local work).
- g = viive (gap), kuvaa verkon viestinvälityskykyä.
- L = latenssi (latency), kuvaa superaskeleen pienintä mahdollista suoritusaikaa.

Parametrien mukaan superaskeleen T_i suoritusaika voidaan lasketa kaavasta:

$$T_i = w_i + gh_i + L,$$

Kaavassa on lisäksi käytetty arvoa h_i, jolla kuvataan suurinta määrää viestejä, jotka suoritin voi lähettää tai vastaanottaa superaskeleen aikana. Vastaavasti koko ohjelman suoritusaika voidaan laskea laskemalla yhteen kaikkien superaskeleiden parametrit:

$$T = W + gH + LS,$$

missä $W = \sum_{i=0}^{S-1} w_i$ ja $H = \sum_{i=0}^{S-1} h_i$, missä S kuvaa kaikkien suoritettavien superaskeleiden määrää. Lyhin mahdollinen suoritusaika pyritään BSP-mallissa saavuttamaan minimoimalla: a) lähetettävien pakettien määrä, b) superaskeleiden määrä ja c) paikallisen työn määrä [16].

Jotta BSP-malli toimisi mahdollisimman tehokkaasti, tulee suorittimilla olla enemmän työtä tehtävänä kuin verkon latenssin aiheuttama viive. Tällaisen ratkaisun saavuttaminen voi olla jossain tapauksissa mahdotonta. [17].

BSP-mallin mukainen ohjelma skaalautuu hyvin erilaisille määrille suorittimia sekä voidaan siirtää eri alustoille. Vaikka BSP-mallin mukainen ohjelma siirretään eri alustalle, pysyy sen suoritus ennustettavana. BSP-mallin etuna on myös sen yksinkertaisuus, sekä tarkkuus. [18]

Ongelmaksi BSP-mallia käytettäessä voi muodostua laitteisto-ominaisuuksien tarkka huomioiminen, sillä malli käsittelee viestinvälityksen viiveitä yhtenä arvona. BSP-mallin huono puoli on myös se, että lähetetty viesti on saatavilla vasta lähetystä seuraavassa superaskeleessa. BSP-malli määrittelee myös erillisen synkronoijan joka varaa käyttöön yhden suorittimen. [19]

BSP-mallin toteuttaa esimerkiksi C-ohjelmointikirjasto BSPLib joka on saatavilla osoitteesta: http://www.bsp-worldwide.org/

LogP-malli on vuonna 1993 julkistettu malli, joka pyrkii arvioimaan BSP-mallia tarkemmin rinnakkaisen laskennan ohjelman kustannusarviota. LogP-malli muodostuu:

- Joukosta $P = \{P_o, P_{1,\dots,} P_{n-1}\}$ suorittimia.
- Verkosta, joka toimittaa viestejä suorittimien välillä.

LogP-malli ei ota kantaa suorittimien välisen verkon rakenteeseen. Sen sijaan, kuten BSP-mallissa, sen oletetaan olevan päästä-päähän tyyppinen. LogP-mallissa verkon tiedonsiirtokyvyn oletetaan olevan rajallinen niin, että verkon siirtokyvyn täyttyessä suoritin joutuu

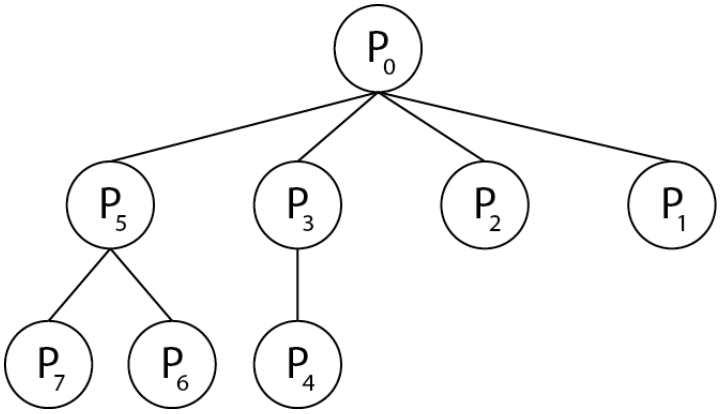
odottamaan verkon vapautumista. Mallin mukaan verkon viiveen vaihtelun vuoksi verkkoon lähetyt viestit voivat saapua eri järjestyksessä missä ne ovat lähetetty. [19]

LogP-mallin parametrit ovat [19]:

- L = verkkoviiveen maksimiarvo (latency), joka aiheutuu viestinvälityksestä lähettäjältä vastaanottajalle.
- o = yleiskustannus (overhead), aika jonka suoritin on sidottuna viestin lähetykseen tai vastaanottoon ja jona aika suoritin ei voi suorittaa muita operaatioita.
- g = viive (gap), vähimmäisaika peräkkäisten viestien lähetyksen/vastaanoton välillä.
- P = suorittimien (processor) määrä.

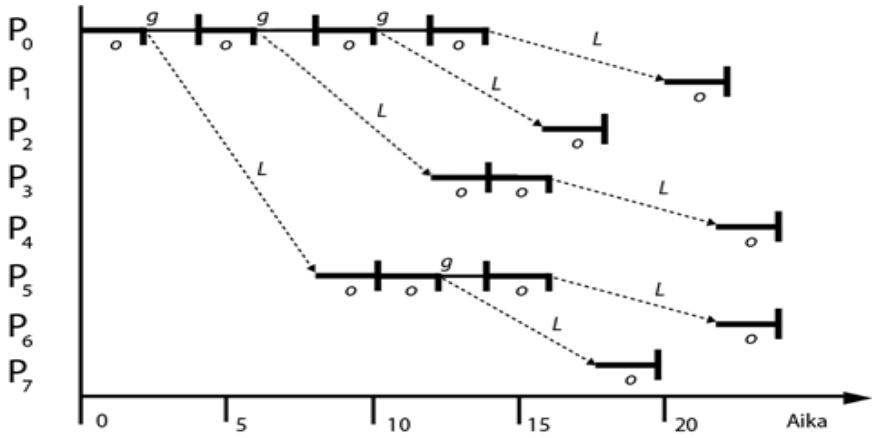
L-, *o*- ja *g*-arvot mitataan suorittimen kellojaksoina. *L*-, *o*- ja *g*-parametrit eivät ole painoarvoltaan samanarvoisia laskettaessa lopullista suoritusaikaa. Parametrien painoarvo riippuu suoritettavasta tehtävästä. Useissa tapauksissa jokin parametreista voidaan ohittaa. Esimerkiksi jossain tapauksissa yleiskustannus arvo on niin suuri, että se yliajaa verkon viiveen. Tällöin viive voidaan jättää huomioimatta. Jos sovellus taas lähettää tietoa harvoin suorittimien välillä voidaan *L* ja *g* arvot jättää pois suoritusaikaa laskettaessa. [19]

LogP-mallin toimintaa voidaan havainnollistaa seuraavasti. Ajassa $t = 0$, suoritin P_o aloittaa viestin M_1 lähetyksen suorittimelle P_1. Suorittimet on järjestetty niin, että jokainen suoritin vastaanottaa lähetetyn viestin vain kerran. Tällöin suorittimet muodostavat optimaalisen viestinvälityspuun (optimal broadcast tree). Optimaalinen viestinvälityspuu on melkein täydellinen binääripuu. Melkein täydellisessä binääripuussa jokainen taso, alinta tasoa lukuun ottamatta, on täynnä ja alimman tason lehdet on järjestetty vasemmalle [20]. Kahdeksan suorittimen melkein täydellinen viestinvälityspuu on havainnollistettu kuvassa 3.



Kuva 3. Melkein täydellinen viestinvälityspuu [19].

Ajassa $t = o$, suoritin P_o on saanut valmisteltua viestin M_1 välitettäväksi verkkoon, ja ajan g jälkeen suoritin P_o voi aloittaa viestin valmistelun seuraavaa suoritinta P_2 varten. Samalla kun suoritin P_o odottaa viiveen (g) kulumista, verkko välittää viestiä M_1 kohdeosoitteeseen P_1. Ajan L kuluttua viesti saapuu suorittimelle P_1. Suorittimelta P_1 kuluu aika o viestin vastaanottoon. Jos suorittimen P_1 pitää lähettää viesti eteenpäin, kuluu vielä aika o lähetyksen valmisteluun. Hetkestä $t = 0$ on siis kulunut $L+2o$ kellonjaksoa siitä kun ensimmäinen suoritin vastaanottaa viestin M_1. Kuvassa 4 on esitetty ylläkuvattu toiminta, arvoilla $P = 8$, $L = 6$, $g = 4$ ja $o = 2$. [19]



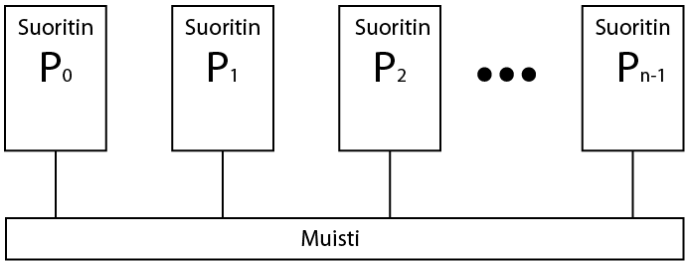
Kuva 4. Esimerkki LogP-ajoituskaaviosta [19].

LogP-malli pyrkii mahdollistamaan riittävän tarkan mallin mahdollisimman suppealla parametrimäärällä. Malli ei määrittele toteutuksen koodikieltä, laitteistoa tai suorittimien välistä verkkoa.

LogP-mallin eduiksi katsotaan laskennan ja viestinnän limittäisyys, joka mahdollistaa asynkronisuuden. Lisäksi eduksi voidaan katsoa myös toiminta tilanteissa, joissa yleiskustannus kasvaa suureksi. Vastaavasti mallin heikkouksia on se, että malli olettaa viestien olevan kooltaan pieniä sekä epädeterministinen toiminta osan parametreista suhteen. [21]

2.2 Yhteisen muistin mallit

Yhteisen muistin malleissa kaikilla prosesseilla on pääsy samaan yhteiseen muistialueeseen. Yhteisen muistin kautta suorittimet voivat kommunikoida keskenään sekä synkronoida tilansa. Abstraktio yhteisen muistin mallista on esitetty kuvassa 5.



Kuva 5. Yhteisen muistin malli [22].

Yhteisen muistin mallit pyrkivät tarjoamaan matalan siirtymiskynnyksen peräkkäisestä ohjelmakoodista rinnakkaiseen. Siirtyminen yhteisen muistin malleilla toteuttavaan rinnakkaiseen monisuoritin ohjelmointiin, yhden suorittimen peräkkäisohjelmoinnista, katsotaan olevan helpompaa kuin käyttämällä viestinvälitysmallia. Yhteisen muistin mallit toimivat myös väliaskeleena, jolloin yhteisen muistin malli toimii pohjana kehittää ohjelmaa toimivaksi monimutkaisemmissa rinnakkaisen laskennan malleissa. Yhteisen muistin mallit ovat helpompia siirtää muistiarkkitehtuurilta toiselle, koska yhteisen muistin mallit eivät ota kantaa itse muistin laatuun tai toteutukseen. [23]

2.2.1 Ei kustannusparametroidut yhteisen muistin mallit

PRAM-malli on vuonna 1978 esitetty teoria siitä, miten rinnakkaisen laskennan tehtävä ratkaistaan. Malli ei ota kantaa järjestelmän fyysisiin ominaisuuksiin, kuten konearkkitehtuuriin tai verkon rakenteeseen. PRAM-mallilla luotu tehtävä voidaan ratkaista millaisessa

rinnakkaisen laskennan järjestelmässä tahansa, mutta mallia voidaan joutua muuttamaan järjestelmään sopivaksi. [24]

PRAM-malli tarjoaa idealistisen näkemyksen rinnakkaiseen laskentaan ja piilottaa mm. viestinnästä ja synkronoinnista aiheuttavat kustannukset. PRAM-malli kuvaa [24]:

- Joukon $P = \{P_o, P_{1,\dots,} P_{n-1}\}$ RAM-suorittimia.
- Suorittimien yhteisen rajoittamattoman globaalin muistin.
- Verkon joka yhdistää suorittimet viiveettömään globaaliin muistiin.

RAM-suoritin on suoritin, joka suorittaa yhden käskyn yhdellä aikayksiköllä. Suorittimella oletetaan olevan rajaton paikallinen muisti. Yhdellä aikayksiköllä suoritettava käsky voi olla lukukäsky, jolloin suoritin lukee arvon yhdestä globaalimuistiosoitteesta paikalliseen muistiin. Kirjoituskäsky, jolloin suoritin kirjoittaa arvon paikallisesta muistiosoitteesta globaaliin muistiin. Laskentakäsky, jonka operandit ovat paikallisessa muistissa. PRAM-mallissa jokaiselle RAM-suorittimelle annetaan sama käskyjono, mutta jokainen suoritin operoi omalla tiedollaan eli malli on ns. SIMD-malli (Single Instruction, Multiple-Data). [25]

PRAM-mallissa RAM-suorittimet suorittavat käskyjä synkronisesti, jolloin jokainen suoritin suorittaa yhden käskyn yhtä kellojaksoa kohden. Kahden suorittimen välinen kommunikointi kestää näin ollen vähintään kaksi aikayksikköä (luku ja kirjoitus). PRAM-mallissa jokainen RAM-suoritin tunnistetaan yksilöllisellä tunnuksella sekä jokainen suoritin omistaa paikallisen pinomuistin. Malli olettaa käytettävissä olevan globaalin muistin olevan rajaton, verkon viiveetön ja käytettävissä olevien RAM-suorittimien määrän ääretön. [24]

PRAM-mallista on kehitetty erilaisia variaatioita. Jokainen variaatio määrittää erilaiset rajat sille, miten prosessit voivat käyttää muistia yhden aikayksikön aikana. EREW-muodossa (Exclusive Read Exclusive Write) kaksi prosessia eivät voi samanaikaisesti lukea tai kirjoittaa samaan muistiosoitteeseen. CREW-muodossa (Concurrent Read Exclusive Write) useampi prosessi voi lukea samasta muistiosoitteesta, mutta vain yksi voi kirjoittaa siihen. ERCW-muodossa (Exclusive Read Concurrent Write) useampi prosessi voi kirjoittaa samaan muistiosoitteeseen, mutta vain yksi prosessi voi lukea siitä. CRCW-muodossa (Concurrent Read Concurrent Write) useampi prosessi voi lukea ja kirjoittaa

samaan muistiosoitteeseen yhtä aikaa. Variaatioissa, joissa useampi prosessi voi kirjoittaa samaan muistialueeseen, muodostuu kilpailutilanne. Kilpailutilanne joudutaan ratkaisemaan, jotta voidaan valita minkä prosessin arvo muistiin kirjoitetaan. [26]

Kilpailutilanne voidaan ratkaista mm. seuraavilla määrityksillä. WEAK-määritelmässä suorittimet voivat kirjoittaa samanaikaisesti samaan muistipaikkaan, jos kaikki kirjoittavat arvon 0. COMMON-määritelmässä kirjoittaminen samanaikaisesti sallitaan, jos kaikki kirjoittavat saman arvon. ARBITRARY-määritelmässä kirjoitetaan muistiosoitteeseen satunnaisesti valittu arvo kirjoitettavista arvoista. PRIORITY-menetelmässä arvon saa kirjoittaa suoritin jolla on pienin prioriteetti. STRONG-menetelmä käyttää eri operaatioita määrittämään miten samanaikainen kirjoittaminen tapahtuu. Esim. STRONG ADD –menetelmässä kirjoitettavat luvut summataan muistipaikkaan. [27]

Eri PRAM-variaatioiden tehokkuutta voidaan vertailla keskenään niiden aika- ja työkompleksiivisuuden suhteen. Vertailussa mallit asettuvat seuraavaan järjestykseen heikoimmasta vahvimpaan: EREW, CREW, COMMON CRCW, ARBITRARY CRCW ja PRIORITY CRCW. [26]

PRAM-ohjelmaa suunniteltaessa käytetään korkean tason rinnakkaisohjelmointikieltä. PRAM-mallin ohjelmointikieli eroaa peräkkäisestä ohjelmointikielestä sillä, että se sisältää lauseen

$$for\ i\ \in S\ pardo\ P_i.$$

Tämä ns. pardo-lause käynnistää jokaisella indeksin *i* arvolla suorituksen P_i. Koska käynnistettäviä suorituksia on määrä *S*, tarvitaan myös suorittimia määrä *S*. Suorittimia voidaan tarvita enemmän, jos jokin proseduurikutsu P_i sisältää myös pardo-lauseita. Palautuminen pardo-lauseesta tapahtuu, kun jokainen suoritus P_i on suoritettu loppuun. Seuraavassa on esitetty esimerkki summan laskemisesta CRCW STRONG ADD-muotoisella PRAM-koneella sekä pardo-lauseen käyttö. [27]

```
procedure summa(n: integer, A: array 1…n of integer)
for i:=1 to n pardo A[n+1] := A[i]
return A[n+1]
```

Esimerkissä lasketaan *n*-alkioisen taulukon *A* summa muistipaikkaan *A*[*n*+1]. Pardo-lauseessa jokainen proseduuri summaa oman indeksinsä mukaisesta taulukon muistipaikasta numeron taulukon muistipaikkaan *A*[*n*+1]. Vastaavasti CREW-variaatiota käytettäessä summausalgoritmi voisi olla muotoa:

```
procedure summa(n: integer, A: array 1…n of integer)
if n = 1 then
        return A[1]
else
if n pariton then n:= n+1; A[n] := 0
for i:=1 to n/2 pardo A[i] := A[i] + A[i + n/2]
summa(n/2,A)
```

Tällä algoritmiversiolla tarvittavien suorittimen määrä on *n*/2, kun ensimmäisellä algoritmilla tarvittiin *n*-kappaletta suorittimia. Ensimmäinen algoritmi suoriutuu ajassa $\theta(1)$, koska pardo-lauseen suoritusaika on $\theta(1)$. Vastaavasti jälkimmäinen algoritmi suoriutuu ajassa $\theta(\log n)$. [27]

PRAM-mallista voidaan laatia myös matalamman tason ohjelmointikielen toteutus. Tällöin käytetään RAM-koneen konekäskykieltä, muutamilla PRAM-lisäyksillä. Aiemmin esitetty CRCW-variaation mukainen taulukon alkioiden arvojen summien laskenta on esitetty konekielisenä seuraavassa. Konekielinen koodi on yhden proseduurin P_i käskyjono. [27]

```
LOADINDEX
STORE   R[1]
LOAD    M[0]
ADD     1
STORE   R[2]
READ    M[R[1]]
ADD     M[R[2]]
```

Oheisessa koodissa suoritin lukee oman indeksinsä (`LOADINDEX`) ja tallentaa sen paikkaan `R[1]`. Taulukon lukujen määrä luetaan paikasta `M[0]` ja siihen lisätään yksi, jolloin

saadaan muistiosoite, jonka arvoon oman indeksin mukaisen taulukon alkion arvo lisätään. READ-käskyllä suoritin lukee oman indeksinsä mukaisen numeron taulukosta ja lisää sen ADD-käskyllä kohdemuistiosoitteeseen. [27]

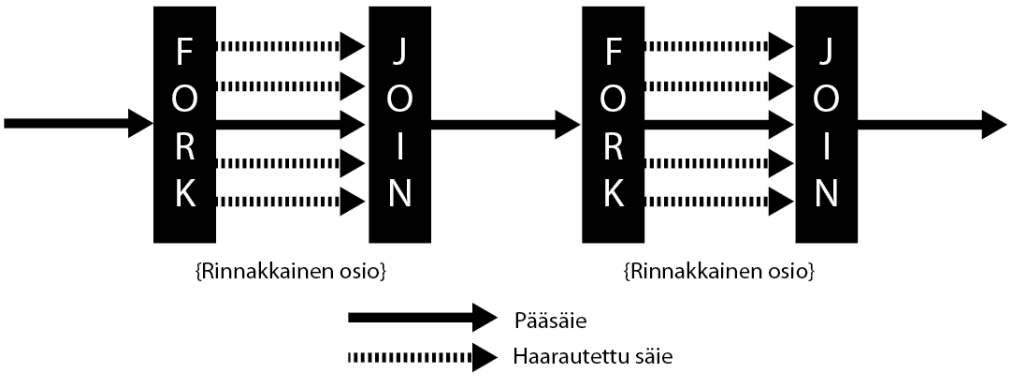
PRAM-mallin suorituskykyä voidaan laskea, kun tunnetaan prosessorien määrä *p*, algoritmin suoritusaika *T* sekä peräkkäisten PRAM-suoritusaskelten lukumäärä eli algoritmin työmäärä *W*. Jos $p = 1$, eli kyseessä on peräkkäislaskennan suorittaminen yhdellä suorittimella $T = W$. Yleensä PRAM-mallissa työmäärä $W <= pT$ ($W = T$ rinnakkaisen laskennan järjestelmässä vain, jos kaikkia suorittimia käytetään koko suorituksen ajan). PRAM-mallin suorituskyky voidaan laskea ns. Brentin lauseen avulla. Brentin lauseen mukaan algoritmin suoritusaika *p* määrän suorittimia sisältävässä järjestelmässä on $T_p = W/p+T$, jos algoritmin työ on *W* ja suoritusaika *T* jollakin suoritinmäärällä. [27]

PRAM-malli on teho perustuu sen yksinkertaisuuteen, malli jättää tarkoituksellisesti huomioimatta verkon ja synkronoinnin viiveet. PRAM-malli pyrkiikin tarjoamaan selkeän kuvauksen siitä mitä aikayksikön aikana ohjelmassa tapahtuu. Koska PRAM-malli on korkeantason suunnitteluun tarkoitettu, ovat sillä luodut mallit vakaita kuvauksia rinnakkaisen laskennan suoriutumisesta. Korkeantason suunnittelu mahdollistaa myös sen, että PRAM-mallilla suunniteltu ohjelma voidaan siirtää pienillä muutoksilla muihin malleihin. Vaikka PRAM-mallin mukainen laitteisto on mahdoton toteuttaa, voidaan PRAM-mallin mukaista ohjelmaa simuloida muissa rinnakkaisen laskennan ympäristöissä. [28]

PRAM-mallin huonoin puoli on sen käytännön toteutettavuus. Rajattoman määrän suorittimia on mahdoton viestiä vakioajassa verkon yli. Myös verkon fyysinen koko kasvaa, mittoihin joita ei voida käytännössä toteuttaa, suorittimien määrän kasvaessa. Malli ei myöskään ota huomioon verkon vaikutusta ohjelman suoriutumiseen. [28]

OpenMP-malli on vuonna 1997 kehitetty ohjelmointikirjasto, malli kehitettiin standardoimaan yhteisen muistin laitteiden ohjelmointia. OpenMP-mallia kehittää OpenMP Architecture Review Board, jossa ovat mukana suurimpien laitteisto- ja ohjelmistovalmistajien edustajat. OpenMP-mallin tuorein version on 3.1, joka julkaistiin heinäkuussa 2011. Versiota 4.0 odotetaan valmistuvaksi vuoden 2013 aikana.

OpenMP on saavuttanut lähes standardin arvoisen aseman yhteisen muistin malleja ohjelmoitaessa. Malli toimii C/C++ ja FORTRAN ohjelmointikielillä Unix ja Windows käyttöjärjestelmissä. OpenMP-ohjelmointikirjasto sisältää kaiken tarvittavan, jotta ohjelmoija voi hyödyntää sitä omassa koodissaan, mutta ei automaattisesti tee peräkkäisestä koodista rinnakkaista. OpenMP-mallissa ohjelman suoritus aloitetaan yhdellä pääsäikeellä, joka edustaa prosessia. Pääsäikeestä voidaan kutsua joukkoa rinnakkaisesti suoritettavia säikeitä, jotka suoriutuvat itsenäisesti. Kun säikeet ovat suorittaneet käskynsä, niiden tilat synkronoidaan ja pääsäikeen suoritusta jatketaan. Koodin suoritusta pääsäikeessä sekä rinnakkaisissa säikeissä on havainnollistettu kuvassa 6. OpenMP-mallin toteuttama koodinhaarautuminen on ns. fork-join mallin mukaista. [29]



Kuva 6. OpenMP-mallin mukainen (fork-join) haarautuminen [29].

Fork-join mallissa ohjelmaa suoritetaan aluksi yhdessä säikeessä, kuten normaalissa peräkkäisohjelmassa. Ohjelma voidaan kuitenkin haarauttaa fork-kutsulla. Fork-kutsua seuraavat käskyt (lohko) suoritetaan ennalta määrätyssä määrässä säikeitä. Kun lohko on suoritettu, jatketaan pääsäikeen suoritusta. Fork-join mallissa haarautunut säie voi myös haarauttaa lisää säikeitä. OpenMP-malli takaa haarautuvan koodin antamaan saman tuloksen kuin vastaava koodi suoritettuna peräkkäisenä koodina. [30]

OpenMP-mallissa oletetaan olevan jaettu muisti, johon kaikki säikeet voivat tallentaa ja josta säikeet voivat lukea. Lisäksi jokaisella säikeellä on ns. väliaikainen näkymä (tempo-

rary view) muistiin. Tätä näkymää säie voi käyttää tallentaessaan tietoa, jota muiden säikeiden ei tarvitse käsitellä. Säie voi siirtää tietoa jaetun muistin ja väliaikaisen näkymän välillä. Mutta kaksi säiettä eivät voi siirtää tietoa omien väliaikaisten näkymien välillä ilman, että tieto kulkee jaetun muistin kautta. Suorituksen aikana luodut muuttujat voidaan OpenMP-mallissa nimetä joko jaetuiksi tai yksityisiksi. Jaetut muuttujat ovat muiden säikeiden käytettävissä, kun taas yksityisiä muuttujia voi käyttää vain ne varannut säie. [31]

OpenMP-malli ei itsessään takaa, tilanteessa jossa useampi prosessi käsittelee samaa muistialuetta, että jaettu muisti toimii yhtenäisesti ja johdonmukaisesti. Sen sijaan malli jättää muistinkäsittelyn käyttöjärjestelmän vastuulle. Mallissa kuitenkin on flush-käsky, jolla ohjelmoija voi pakottaa muuttujien arvon kirjoittamisen muistiin. Flush-käskyn suorittaminen varmistaa, että muuttujien arvo on kirjoitettu muistiin ennen kuin muut säikeet voivat niitä lukea muistista. Flush-käsky on OpenMP-mallin ainoa kutsu, jolla voidaan taata muuttujien arvojen säilyminen, kun arvoja siirretään prosessien välillä. [31]

Koska OpenMP-mallissa säikeet toimivat ilman yhteistä kelloa, voidaan niiden välistä tilaa synkronoida myös muilla kuin flush-komennolla. Rajasynkronoinnissa säikeet odottavat rajakohdassa muita säikeitä ja jatkavat suoritusta vasta, kun kaikki säikeet saavuttavat rajan. Kriittisen lohkon suorituksessa ohjelmoija voi määrittää koodin käskyt, jotka vain yksi säie suorittaa kerrallaan. Lisäksi OpenMP-mallissa voidaan määrittää erikseen koodiosio, joka ajetaan vain pääsäikeessä sekä osio joka ajetaan vain yhden säikeen toimesta. [32]

OpenMP-mallin rinnakkaisuutta ohjataan ohjelmakoodista kääntäjäkomennoilla. C/C++ kielen syntaksissa, jokainen OpenMP-komento alkaa sanoilla `#pragma omp`. Tärkein rinnakkaisuuteen liittyvä käsky on `#pragma omp parallel`. Tällä käskyllä käsketään ajettavan koodin suorittaa seuraava lohko rinnakkaisina prosesseina. Rinnakkaisten prosessien määrä määräytyy suoritusympäristöön asetettujen arvojen mukaan. For-silmukka voidaan puolestaan jakaa suoritettavaksi eri prosesseille komennolla `#pragma omp parallel for`. Koska OpenMP-malli sisältää suuren määrän erilaisia käskyjä niitä ei tässä yhteydessä käsitellä syvemmin. Käskyjen kuvaukset löytyvät ”OpenMP Application Programming Interface” -kirjasta. [33]

Seuraavassa on esitetty OpenMP-mallin mukainen ohjelmakoodi, joka laskee taulukon *a* kahden peräkkäisen alkion keskiarvon taulukkoon *b*. For-silmukan lause suoritetaan eri prosesseissa jolloin taulukko *b* täyttyy satunnaisessa järjestyksessä, sitä mukaan kun prosessit saavat laskutoimituksensa tehtyä. [33]

```
void a1(int n, float *a, float *b)
{
int i;
#pragma omp parallel for
for (i=1; i<n; i++)
b[i] = (a[i] + a[i-1]) / 2.0;
}
```

OpenMP-mallin suurin etu, mallin laajan laitteistotuen lisäksi, on matala siirtymiskynnys peräkkäisestä koodista rinnakkaiseen laskentaan. Helpoimmillaan peräkkäinen koodi saadaan rinnakkaiseksi lisäämällä kääntäjäkomennot sekä OpenMP-kirjasto. Kääntäjäkomentojen avulla koodi pysyy myös siirrettävänä, sillä jos ympäristö ei tue OpenMP:tä tulkitaan kääntäjäkomennot kommenteiksi. Samoin kääntäjäkomennoilla voidaan rinnakkaistaa vain osa koodia. Siirtymistä peräkkäisestä koodista rinnakkaiseen OpenMP:hen edesauttaa se, ettei ohjelmoijan tarvitse huolehtia säikeiden hallinnasta vaan OpenMP hoitaa säikeiden hallinnan. [34]

OpenMP mallin suurin heikkous on sen skaalautuvuus suurille prosessimäärille. Alustasta riippuen OpenMP-mallilla voidaan saavuttaa suorituskykyhyötyä vain tiettyyn prosessimäärän asti. Huonoiksi puoliksi voidaan laskea myös prosessien suorittaminen satunnaisessa järjestyksessä, mikä voi aiheuttaa lisätyötä jossain ohjelmissa. Lisätyötä ja huolellisuutta vaatii myös se, että OpenMP-mallissa ohjelmoijan tulee huolehtia säikeiden tilojen synkronoinnista [9, 35]

OpenMP:n ajantasainen dokumentaatio, esimerkkikoodeja ja kirjastot eri käyttöjärjestelmil le ovat saatavissa osoitteesta: http://www.openmp.org.

2.2.2 Kustannusparametroidut yhteisen muistin mallit

QSM-malli (Queuing Shared Memory) edustaa yhteisen muistin malleissa samanlaista mallia kuin BSP- ja LogP-mallit hajautetun muistinmalleissa. QSM-malli pyrkii kahdella parametrilla mallintamaan yhteisen muistin mallin toteuttamaa järjestelmää. QSM-mallin mukainen järjestelmä koostuu [36]:

- Joukosta $P = \{P_o, P_{1,\dots,} P_{n-1}\}$ identtisiä suorittimia.
- Suorittimien yhteisestä rajoittamattomasta jaetusta muistista.
- Verkosta joka yhdistää suorittimet jaettuun muistiin.

QSM-mallissa suorittimet suorittavat synkronoidusti vaiheita. Jokaisessa vaiheessa suoritin voi kopiota yhteisestä muistista tietoa omaan yksityiseen muistiin, kirjoittaa yhteiseen muistiin tai suorittaa käskyjä paikallisessa muistissa olevilla tiedoilla. Vaiheen aikana luettu tieto on käytettävissä paikallisessa muistissa vain seuraavan vaiheen ajan. Suoritin voi lukea tai kirjoittaa valittuun muistialueeseen vaiheen aikana, mutta ei tehdä molempia. Jos suoritin kirjoittaa useamman kuin yhden kerran muistialueeseen, satunnaisen kirjoituskerran arvo valitaan muistialueen arvoksi. [23]

Vaiheen aikana useampi suoritin voi kirjoittaa tai lukea samaa muistialuetta, mutta QSM-malli määrittää kustannuksen tällaisessa kilpailutilanteessa. Kilpailutilanteessa QSM-malli muodostaa, nimensä mukaisesti, jonon muistialuetta kohden, jolloin yhden vaiheen kustannusarvioksi tulee muistialuetta käsittelevien suorittimien lukumäärää sekä paikallisen työmäärän kustannus. Jos vaiheessa yksikään suoritin ei kirjoita tai lue muistiosoitetta annetaan vaiheella kustannusarvoksi yksi. [23]

Malli määrittää myös verkkoviiveen (gap), jolla voidaan kuvata paikallisen työmäärän ja viestinvälityksen viiveen suhdetta. Yhden vaiheen kustannus voidaan laskea kun tiedetään samasta muistialueesta kilpailevien suorittimien lukumäärä x, suurin määrä paikallisia operaatioita m_{op} ja suurin luku/kirjoituskäskyjen määrä m_{rw}. Näiden suureiden avulla vaiheen kustannusarvio saadaan kaavasta

$$C = \max(m_{op}, m_{rw}, x). \tag{3}$$

Vastaavasti koko algoritmin kustannusarvio on vaiheiden kustannusarvioiden summa, ja lasketaan kaavalla

$$C_{total} = \sum_{x=0}^{n-1} C_i. \quad (4)$$

[37]

QSM-mallin yksikertainen esitys pyrkii kahdella parametrilla piilottamaan toissijaisesti suorituskykyyn vaikuttavat yksityiskohdat järjestelmää mallinnettaessa, jolloin ohjelmoija pystyy keskittymään algoritmin suorituksen ja muistinkäytön optimointiin. QSM-mallin on todettu tuottavan lähelle todellista suoritusaikaa kuvaavia arvioita suurillakin tietomäärillä. Mutta koska malli ei huomioi verkon viivettä tarpeeksi tarkasti, aiheutuu mallin kustannusarvion ja todellisen suorituskyvyn välille eroja, jos verkko toimii hitaasti. [36]

P-PRAM -malli (Parameterized PRAM) toimii PRAM-mallin jatkeena. PRAM-mallilla voidaan suunnitella suoritettavaa rinnakkaisen laskennan paradigmaa. Kun taas P-PRAM-mallilla voidaan laskea suunnitellun paradigman kustannusarviota. P-PRAM-mallissa kustannusarviota lasketaan viidellä parametrilla:

- 1, paikallisen operation kustannusarvio.
- α. EREW-muistinkäsittelyn kustannusarvio.
- β, CRCW-muistinkäsittelyn kustannusarvio.
- μ, Multiprefix-operaation kustannusarvio.
- δ, Globaalisynkronoinnin kustannusarvio.

P-PRAM-malli määrittää paikalliselle operaatiolle kustannusarvion yksi. Paikallisen operaation oletetaan olevan aina nopein suoritettava käsky, riippumatta suorittavasta laitteistosta. Koska paikallinen operaatio on nopea, myös mitattuna kellojaksoina, määrittää paikallisen operaation nopeus P-RAM-mallille tarkan kellon kustannusarvioiden laskemiseen. EREW-muistinkäsittely on kustannusarvio tilanteesta, jossa vain yhden prosessin sallitaan samaan aikaan kirjoittavan tai lukevan muistialuetta. EREW-muistinkäsittely kuvastaa siis kilpailutilannetta suorittimien lukiessa tai kirjoittaessa samaan jaettuun muistipaikkaan. CRCW-muistinkäsittelyssä useampi prosessi voi samaan aikaisesti lukea tai kirjoittaa samaan muistipaikkaan. Multiprefix-operaatioihin kuuluvat esim. prioritisoidut muistikäsitte-

lyoperaatiot. Prioritisoidussa muistinkäsittelyssä prosessi jolla on alhaisin prioriteetti saa kilpailutilanteessa käsitellä muistia. [38]

P-PRAM-mallilla voidaan laskea myös kustannusarviota PRAM-mallin eri variaatioille. PRAM-mallin EREW-variaation kustannusarvio voidaan laskea, kun käytetään $\alpha = 1$, $\delta = 0$ ja muiden parametrien kustannusarvion oletetaan olevan ääretön. Vastaavasti CRCW-variaation kustannusarvio voidaan laskea kun käytetään $\alpha = 1$, $\beta = 1$, $\delta = 0$ sekä $\mu = \infty$. [38]

Seuraavassa esimerkissä esitetään PRAM-mallin mukaisen algoritmin kustannusarvion laskeminen. Esimerkkiohjelma testaa esiintyykö *n*-kokoisessa positiivisten kokonaislukujen taulukossa *A*, sellaista lukua joka esiintyy yli puolessa taulukon indekseistä.

```
1. sort the array A
for all procs i in parallel do
2.1 Found[i]:=0
2.2 if A[n mod 2] = A[i] then
2.3 Found[i]:=1
3. total := sum of values in Found
4.1 if total > n/2 then
4.2 majority:= A[n div2]
```

Esimerkkiohjelman rivi 1 voidaan suorittaa, käytettäessä PRAM EREW-variaatiota, ajassa $log\, n$. Tällöin P-PRAM-mallin mukainen kustannusarvio on $\alpha\, log\, n$. Rivi 2.1 on EREW–operaatio, jolloin sen kustannusarvio on α. Rivillä 2.2 tapahtuu rinnakkainen lukeminen muistipaikasta *A*[*n mod 2*], eksklusiivinen lukeminen paikasta *A*[*i*] sekä eksklusiivinen kirjoittaminen paikkaan *Found*[*i*]. Rivin 2.2 kustannusarvio on $2\alpha + \beta$. Rivi 3 toteuttaa multiprefix-operaation kustannusarviolla μ. Viimeinen rivi 4.2 voidaan suorittaa yhdellä suorittimella EREW-operaationa, jolloin lukeminen ja kirjoittaminen tapahtuvat kustannusarviolla 2α. Koska suorittimet tulee synkronoida taulukon lajittelun jälkeen, lisätään 1. rivin kustannus arvioon δ. Tästä seuraa koko ohjelman kustannusarvio $((\alpha + \delta)\, log\, n + \beta + \mu)$. [38]

P-PRAM mallilla voidaan laskea kustannusarvioita niille ohjelmille, joiden algoritmi on laadittu PRAM-mallia noudattaen. Itse P-PRAM ei siis määrittele tiettyjä laitteistorakennetta tai koodikieltä, vaan sen tarkoitus on määrittää kustannusarvio eri PRAM-operaatioille. P-PRAM malli määrittää selkeät parametrit erilaisille rinnakkaisen laskennan operaatioille, mutta on toisaalta riippuvainen PRAM-mallin mukaisesta laitteisto- ja ohjelmatoteutuksesta.

2.3 Pohdintaa malleista

Edellä on kuvattu rinnakkaisen ohjelmoinnin malleja, jotka ovat yleisiä tai tämän työn kannalta merkityksellisiä. Mallit jaettiin viestinvälitysmalleihin sekä yhteisen muistin malleihin riippuen niiden laitteistoarkkitehtuurista.

Käsitellyistä ei-kustannusparametroiduista viestinvälitysmalleista (PVM ja MPI) löytyy paljon samankaltaisuuksia mutta myös eroja. Tehokkuudessa suurin ero löytyy siitä, että MPI-kirjasto pystyy hyödyntämään tehokkaammin laitteiston ja käyttöjärjestelmän tarjoamat hyödyt. PVM-malli taas tarjoaa matalamman siirtymiskynnyksen peräkkäisestä ohjelmakoodista rinnakkaiseen suoritukseen. Siirtymiskynnys PVM-ympäristöön on matala varsinkin, jos käytössä on heterogeeninen ympäristö. Rinnakkaisen laskennan dynaamisuuden suhteen PVM-malli on perinteisesti ollut MPI-mallia edellä, mutta MPI-mallin uusimmat versiot ovat tuoneet mallit tässä suhteessa tasavertaisiksi. Rinnakkaisessa laskennassa, jossa laskutoimitukset voivat kestää päiviä tai viikkoja, vikasietoisuus on tärkeä ominaisuus. Vikasietoisuudessa PVM-malli on MPI-mallia parempi, sillä PVM-mallissa prosessin ”kuollessa” tai epäonnistuessa voi toinen prosessi saada tiedon tapahtuneesta ja reagoida tilanteeseen. Yleisesti ottaen paras valinta käytettävään kirjastoon tulee valita käyttötapauskohtaisesti. Ohjenuorana voidaan kuitenkin pitää sitä, että PVM-malli toimii parhaiten, kun rinnakkaista laskentaa suoritetaan heterogeenisessä ympäristössä ja MPI-malli parhaiten, kun kyseessä on ympäristö jossa prosessit ajetaan samankaltaisissa laitteistoissa. [39, 40]

Kustannusparametroidut viestinvälitysmallit LogP ja BSP –mallit eivät ota kantaa siihen, miten prosessien välinen verkko muodostuu, mutta tunnustavat molemmat verkon vaikut-

tuvan merkittävästi rinnakkaisen laskennan tehokkuuteen. Molemmat mallit pyrkivät eri parametrin ennustamaan rinnakkaisen laskennan toimintaa ympäristössä, jossa parametrit ovat vakioita. LogP ja BSP -mallien suurin ero on prosessien synkronoinnissa ja viestienvälitystavassa. BSP-malli vaatii prosessien tilan synkronoinnin, kun taas LogP-mallissa tarkkaa prosessien synkronointia ei suoriteta. LogP sen sijaan tarjoaa paremman hallinnan laitteiston ominaisuuksiin sekä pyrkii kuormittamaan verkkoa vähemmän. Malleista LogP soveltuu paremmin ennustamaan toimintaa tilanteissa, joissa prosessien välinen verkko muodostuu laskennalle hidasteeksi. Koska LogP-mallissa voidaan käyttää useampia parametreja verkon toiminnan kuvaamiseen. LogP toimii kuitenkin hyvin ainoastaan kun verkon suorituskyky on staattinen. Yksinkertaisempi BSP-malli vastaavasti sopii paremmin data-rinnakkaisen laskennan tehokkuuden arviointiin sekä verkon ja laskennan keskinäisen suorituskyvyn vertailuun. [41]

Yhteisen muistin malleista käsiteltiin ei-kustannusparametroitujen mallien osiossa PRAM-mallia sekä OpenMP-mallia. PRAM-malli valittiin käsittelyyn, koska se on yksi rinnakkaisen laskennan perusmalleista. PRAM-malli on teoreettinen malli, ja se on toiminut pohjana useille muille malleille ja laitteistototeutuksille. PRAM-johdannaisilla malleilla pyritään toteuttamaan alkuperäistä PRAM-mallia lähellä oleva malli, joka toimii käytännössä toteutettavissa olevassa laitteistossa. Vastaavasti laitteistokehityksessä on pyritty säilyttämään PRAM-malli, mutta toteutettu mahdollisimman lähelle PRAM-mallin teoreettista suorituskykyä pystyvä laitteisto. PRAM-mallia voidaankin käyttää lähinnä suunniteltaessa rinnakkaisen laskennan paradigmoja tai laitteistoja korkealla tasolla. Tarkempaan toteutukseen kannattaa valita joku muu yhteisen muistin malli. OpenMP sen sijaan toteuttaa kirjastot, joilla rinnakkaista laskentaa voidaan suorittaa. OpenMP malli suorittaa ohjelmaan aluksi yhdessä pääsäikeessä, kuten myös PRAM malli, pää säikeestä ohjelmoija voi käynnistää itse rinnakkaisesti toimivat haarat. OpenMP-mallissa koodin rinnakkaistaminen toimii fork-join komennoilla ja PRAM-mallissa pardo-lauseella. Saatavuutensa ansiosta OpenMP-malli on varteenotettava vaihtoehto, kun harkitaan siirtymistä rinnakkaiseen laskentaan. OpenMP-mallin avulla voidaan ohjelmasta myös tehdä rinnakkaista pala palalta. [42, 3, 43]

Käsitellyistä kustannusparametroista yhteisen muistin malleista QSM ja P-PRAM edustavat erilaisia tapoja kustannusarvioiden laskemiseen. P-PRAM mallilla arvioidaan jo tehdyn PRAM-algoritmin tehokkuutta erilaisin parametrein. QSM-malli taas vastaa PRAM ja P-PRAM mallin yhteiskäyttöä, sillä sen avulla voidaan suunnitella koko rinnakkaisen laskennan ohjelma ja arvioida sen suorituskykyä. Parametreilla mitattuna QSM-malli on P-PRAM mallia kevyempi, sillä se pyrkii laskemaan kustannusarvion kahdella parametrilla. PRAM-mallissa taas on käytössä viisi parametria. Kumpikaan esitetyistä malleista ei määrittele tarkkaa parametria verkon viiveelle vaan laskevat kustannuksia vain paikallisten operaatioiden ja muistinkäsittely operaatioiden kustannusten perusteella.

Yhteisen muistin mallien ja viestinvälitysmallien lisäksi on kehitetty myös ns. hybridimalleja, joissa yhdistetään yhteisen muistin ja viestinvälitysmallin toiminta. Yksi tällainen malli on MPI- ja OpenMP-mallin yhdistelmä, jossa OpenMP-mallia käytetään korkeamman tason rinnakkaisessa koodissa ja MPI-mallilla ohjataan varsinaista laskentaa tekevää koodia. Hybridimalleilla saadaan rinnakkaislaskennan paradigma vastaamaan tarkemmin järjestelmää, jossa rinnakkaisen laskennan koodi suoritetaan. Tässä työssä hybridimalleja ei tarkemmin käsitelty, koska ne eivät työn kannalta ole oleellisia. [44, 45]

Rinnakkaisen laskennan mallien kehittäminen ja parantaminen on suosittu tietokonetekniikan tutkimusala ja useita tässä työssä esitettyjä malleja on parannettu kapea-alaisempiin käyttötarkoituksiin soveltuviksi. Lisäksi on olemassa useita rinnakkaisen laskennan malleja, jotka eivät ole tämän työn tai rinnakkaisen laskennan tutkimuksen kannalta merkittäviä, lisätietoa eri malleista löytyy mm. rinnakkaisen laskennan konferenssijulkaisuista. Seuraavassa listassa on malleja, jotka jätettiin tässä työssä käsittelemättä tilan puutteen vuoksi:

- Cilk/Cilk++ [46].
- Posix Threads (pThreads) [47].
- OpenACC [48].
- Postal model [49].
- QRQW Asynchronous PRAM Model [50].
- Asynchronous PRAM [28].
- Phase PRAM [28].

- Thread Building Blocks (TBB) [51].
- OpenCL [52].
- BrookGpu [53].

3 GPU-laskenta

Tässä luvussa esitellään yleisesti grafiikkasuorittimella suoritettavaa laskentaa. Luvun alussa käydään läpi grafiikkasuorittimen historiaa, rooli tietokoneen arkkitehtuurissa sekä grafiikkasuorittimen fyysistä rakennetta. Seuraavaksi perehdytään grafiikkasuorittimen ohjelmointiin tarkoitettuun NVIDIAn CUDA-kehitysympäristöön. Lopuksi esitellään ominaisuudet, joita grafiikkasuoritin tarjoaa laskentakäyttöön.

3.1 Johdanto

Grafiikkasuoritin (GPU, Graphical Processing Unit) on tietokoneen suorittimesta (CPU, Central Processing Unit) erillinen komponentti. Grafiikkasuoritin vastaa mm. kuvien, videoiden, 3D-grafiikan ja käyttöliittymän piirrosta näytölle. Piirtämisen lisäksi grafiikkasuoritin hoitaa piirtoon liittyvän laskennan [54]. Rinnakkaisen rakenteensa vuoksi nykyaikainen grafiikkasuoritin pystyy suorittamaan sille räätälöityjä laskutoimituksia tehokkaammin kuin tietokoneen varsinainen suoritin. [55]

Tässä työssä käsitellään vuonna 2009 julkaistuun NVIDIAn Fermi-arkkitehtuuriin pohjautuvia grafiikkasuorittimia. Vuoden 2012 alussa NVIDIA julkaisi uuden Kepler-arkkitehtuurin. Lisäksi ATI:lla on oma Cypress-arkkitehtuurinsa. Koska Fermi-arkkitehtuuri on näistä yleisin, käsitellään tässä työssä sen avulla tehtävää laskentaa.

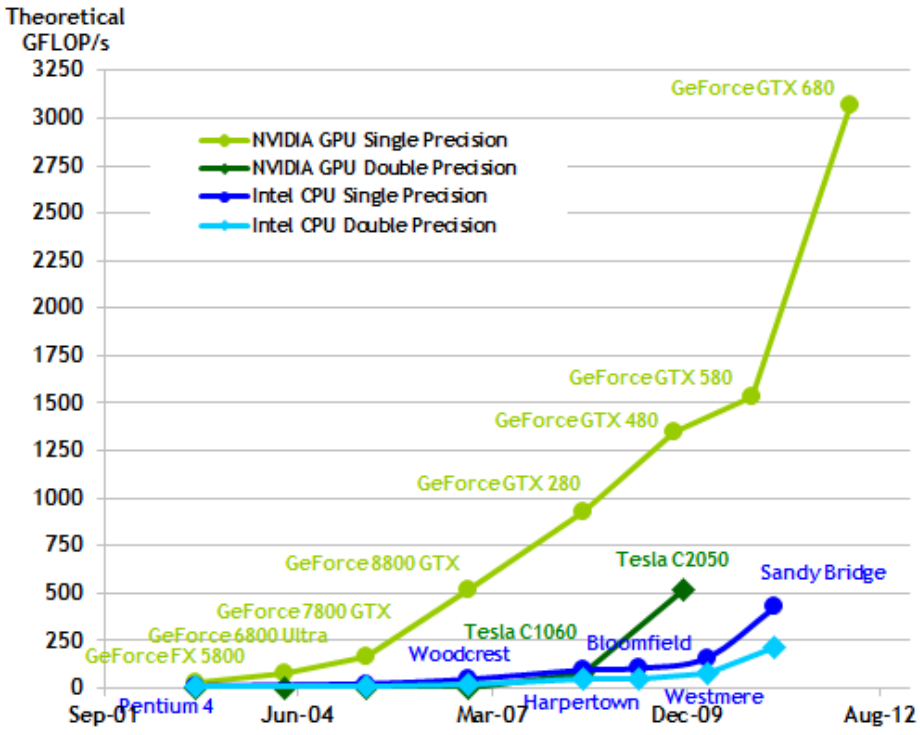
Grafiikkasuorittimen käyttäminen yleiseen laskentaan katsotaan alkaneen vuonna 1999 NVIDIAn julkaisemasta GeForce256 näytönohjaimesta. GeForce256 oli ensimmäinen laite, jota kutsuttiin varsinaisesti grafiikkasuorittimeksi, aikaisempia laitteita kutsuttiin grafiikan kiihdyttimiksi. Grafiikkakiihdyttimet olivat laitteita, jotka suorittivat etukäteen määriteltyjä funktioita. Funktioiden tarkoituksena oli nopeuttaa kuvan piirtämiseen liittyvää laskentaa. Sen sijaan nykyaikainen grafiikkasuoritin on tehokas rinnakkaisen laskennan laite, jossa sadat tai tuhannet suoritinytimet hoitavat laskentaa. Grafiikkasuorittimen laskentaa voidaan ohjelmoida useilla ohjelmointikielillä useissa ympäristöissä, kuten esim. C/C++ kielellä. [56]

Grafiikkasuorittimen hyödyntäminen tieteelliseen laskentaan sai alkunsa, kun grafiikkakiihdyttimien grafiikkarajapintoja alettiin käyttämään erilaisten ei-graafisten laskutoimituksien suorittamiseen. Tätä tekniikka kutsutaan GPGPU:ksi (General Purpose computation on GPU). GPGPU ohjelmointi hyödynsi alun perin OpenGL ja DirectX rajapintoja, joiden päälle tutkijat rakensivat omia funktioita ja julkaisivat niitä kirjastoiksi. [57]

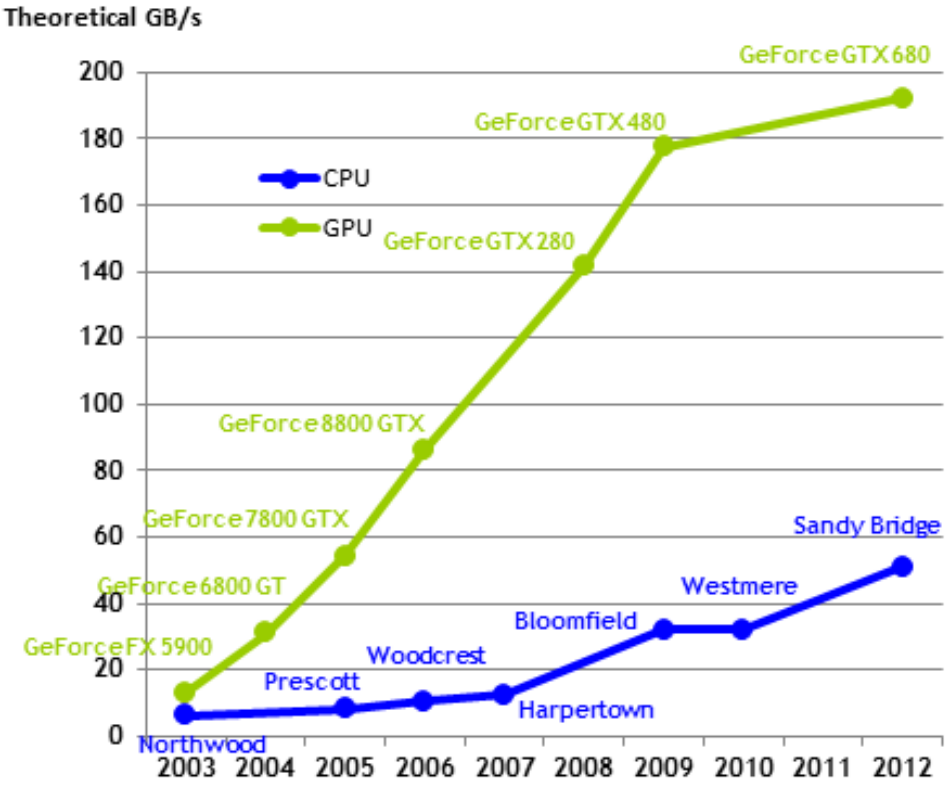
Varsinainen laskennan suorittaminen grafiikkasuorittimella tuli täysimittaisesti saataville marraskuussa 2006, kun ATI julkisti Close To Metal (CTM) ohjelmointiympäristön. CTM tarjosi matalan tason ohjelmointiympäristön, kääntäjän, debuggerin ja kirjastoja. Vuonna 2007 NVIDIA julkaisi vastineen CTM:lle, tätä ympäristöä kutsutaan CUDA:ksi (Compute Unified Device Architecture). CUDA toi grafiikkasuorittimen ohjelmoinnin saataville korkeamman tason ohjelmointikielille, sillä se mahdollisti grafiikkasuorittimen ohjaamisen käyttämällä C-kieltä. CUDA:n ja CTM:n avulla grafiikkasuorittimen tehoja pystyttiin hyödyntämään aikaisempaa yksinkertaisemmin. Ohjelmointiympäristöt lunastivat nopeasti paikkansa mm. tieteellisessä laskentatyössä.

Nykyään molemmat suuret grafiikkasuoritinvalmistajat, NVIDIA ja ATI, tarjoavat ohjelmointiympäristöjä grafiikkasuorittimen ohjaamiseen. Ohjelmointiympäristöt ovat tulleet tarjolle myös muille ohjelmointikielille, kuten Pythonille. Lisäksi grafiikkasuoritinta voidaan hyödyntää suoraan useista laskentaohjelmista kuten MATLAB-ohjelmasta ja erilaisista kuvankäsittelyohjelmista. Tämä on johtanut siihen, että grafiikkasuoritinta voidaan helposti käyttää erilaisissa laskutehoa vaativissa tehtävissä.

Grafiikkasuorittimen ja suorittimen välisiä suorituskykyeroja voidaan teoreettisella tasolla vertailla kahdella eri mittarilla. Teoreettinen laskutoimituskyky mittaa, kuinka monta liukulukulaskutoimitusta suoritin kykenee teoreettisesti suorittamaan sekunnissa (GFLOP/s). Kaistanleveys taas mittaa kuinka monta gigatavua tietoa siirtyy grafiikkasuorittimen ja sen muistin välillä sekunnissa (GB/s). Kuvassa 7 on esitetty grafiikkasuorittimen ja suorittimen laskutoimituskyvyn kehitys viime vuosien aika. Kuvassa 8 on vastaavasti esitetty grafiikkasuorittimen ja suorittimen muistin kaistanleveyden kehitys. Vertailussa on käytetty NVIDIAn ja Intelin kehittimiä suorittimia.

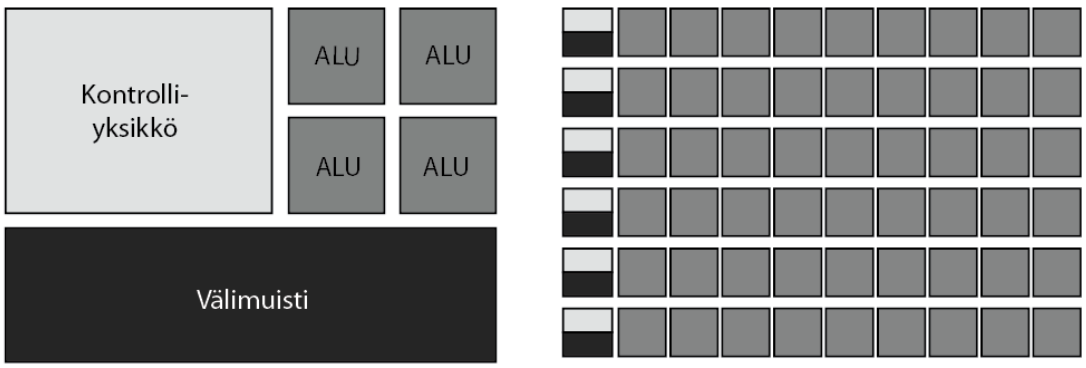


Kuva 7. CPU ja GPU suorittimien laskutoimitustehon kehitys [58].



Kuva 8. CPU ja GPU suorittimien kaistanleveyden kehitys [58].

Kaistanleveydellä ja laskutoimituskyvyllä mitattuna grafiikkasuoritin vaikuttaa olevan ylivertainen CPU:hun verrattuna. Grafiikkasuorittimen ja CPU:n arkkitehtuurit ovat kuitenkin täysin erilaiset. Suoritin on suunniteltu suorittamaan yksi käsky mahdollisimman nopeasti (low-latency). Lisäksi suurella välimuistilla varustettu suoritin on optimoitu suorittamaan haarautuvaa ja epäsäännöllistä koodia. Grafiikkasuoritin puolestaan toimii parhaiten, kun sen suorittamat käskyt ovat yksinkertaisia eivätkä käskyt tarvitse useita muistihakuja. Suorittimen arkkitehtuurissa koodin suoritusta ohjaava kontrolliyksikkö (CU) sekä iso välimuisti vievät suuren tilan suorittimesta. Tämän vuoksi suoritin sisältää vähemmän aritmeettis-loogisia yksiköitä (ALU), jotka suorittavat käskyjen operaatiot. Grafiikkasuorittimessa vastaavasti kontrolliyksikkö ja välimuisti vievät vähemmän tilaa, jolloin grafiikkasuorittimen pinta-alalle mahtuu enemmän aritmeettis-loogisia yksiköitä. Suorittimen ja grafiikkasuorittimen arkkitehtuurin ero on kuvattu kuvassa 9. [59]

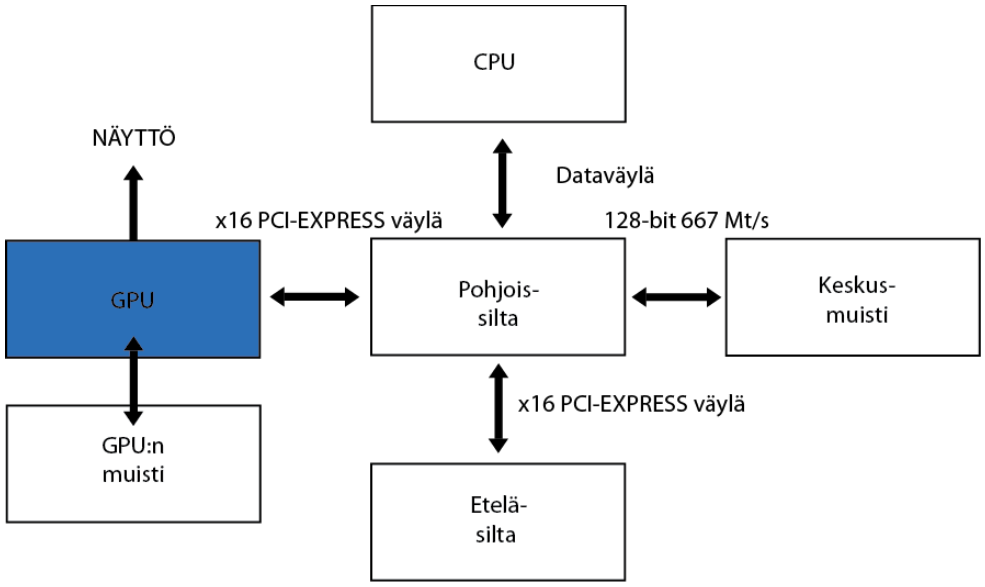


Kuva 9. CPU (vasemmalla) ja GPU -arkkitehtuurien ero [59].

3.2 GPU:n fyysinen rakenne

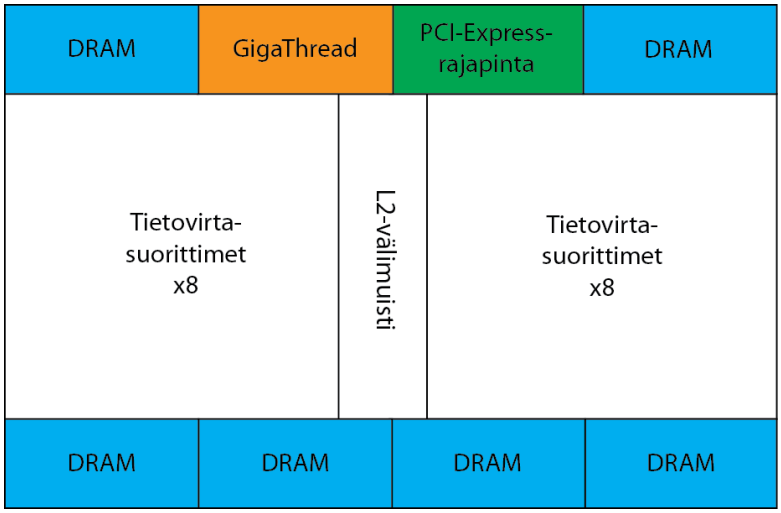
Tässä luvussa tarkastellaan grafiikkasuorittimen fyysistä rakennetta, sekä sen sijaintia tietokoneen arkkitehtuurissa. Paikoittain joudutaan tyytymään grafiikkasuorittimen ominaisuuksien pintapuoleiseen tarkasteluun, koska yksityiskohtaisia tietoja kaikista ominaisuuksista ei ole saatavilla. Grafiikkasuoritin voi olla joko integroitu emolevyyn tai sijaita omassa laajennuspaikassaan.

Grafiikkasuoritin on yhdistetty tietokoneen emolevyllä varsinaiseen suorittimeen pohjoissillan (North Bridge) kautta. Siirtoväylänä suorittimen ja grafiikkasuorittimen välillä toimii nykyaikaisessa tietokoneessa PCI-Express x16 -väylä. Käytössä olevat PCI-Express-väylät kykenevät siirtämään tietoa korkeintaan nopeudella 8GB/s suuntaan. Grafiikkasuorittimen sijaintia tietokoneen arkkitehtuurissa on havainnollistettu kuvassa 10. Kuvassa esitetään CPU, muisti ja GPU sekä väylät jotka yhdistävät ne toisiinsa. Lisäksi kuvassa on esitetty tavanomaisimpia väylien toteutustapoja ja nopeuksia. [54]



Kuva 10. GPU:n sijainti Intelin PC-rakenteessa [54].

Itse grafiikkasuoritin (Fermi-arkkitehtuuri) koostuu useasta osasta, jotka on esitetty kuvassa 11. Fremi-arkkitehtuuri määrittää grafiikkasuorittimen koostuvan 16 tietovirtasuorittimesta (SM, streaming processor), kuudesta DRAM-muistiyksiköistä, säiehallintamoottorista (GigaThread™) sekä rajapinnasta PCI-Express –väylään (Host Interface). [60]



Kuva 11. NVIDIAn Fermi-arkkitehtuuri [60].

Fermi-arkkitehtuurissa grafiikkasuoritin suorittaa suuren määrän säikeitä samanaikaisesti. Säikeiden hallinta perustuu SIMT–arkkitehtuuriin (Single Instruction, Multiple Thread). SIMT-arkkitehtuuri kuvataan tarkemmin myöhemmin. Fermi-arkkitehturissa säikeitä hallitaan erillisellä GigaThread™-moottorilla, joka hoitaa muun muassa säikeiden välityksen tietovirtasuorittimille. [60]

Fermi-arkkitehtuurissa grafiikkasuorittimella on kuusi 64-bittistä muistiosiota, jotka voivat sisältää yhteensä 6 GB GDDR5 DRAM-muistia [60]. DRAM-muistista käytetään nimitystä laitemuisti (device memory). Laitemuisti on jaettu useampaan osaan, joita tarkastellaan tarkemmin CUDA-ohjelmoinnin yhteydessä. Laitemuistin hakuajat ovat 400 – 800 kellojakson luokkaa. Grafiikkasuoritin piilottaa hakuajan viiveen (latenssin) suorittamalla tuhansia säikeitä samaan aikaan, jolloin jokin säie on aina valmiina suoritettavaksi, vaikka muut säikeet odottaisivatkin tietoa muistista. Sekä käyttämällä välimuistia tiedon varastoimiseen nopeaa käyttöä varten. [59]

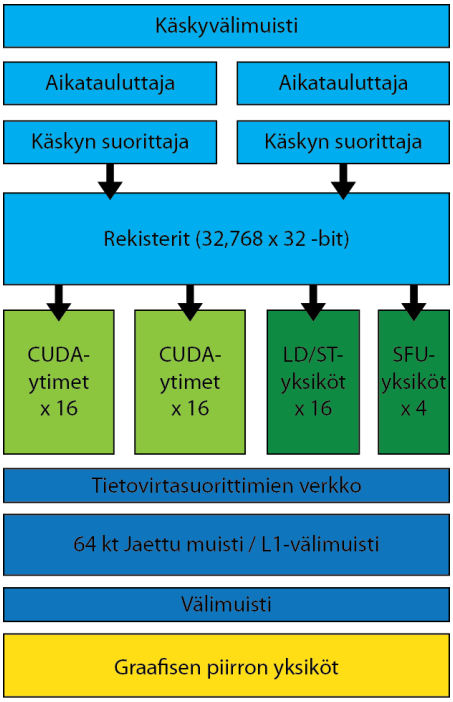
DRAM-muisti jaetaan neljään osaan: globaaliin muistiin, paikalliseen muistiin, tekstuurimuistiin ja vakiomuistiin. Näistä rinnakkaislaskennan kannalta merkittävimmät ovat globaali- ja paikallinen muisti. Globaalin muistin avulla voidaan jakaa tietoa eri säikeiden välillä tietovirtasuorittimien yli, koska globaaliin muistiosaan on kaikilla säikeillä luku- ja kirjoitusoikeus. Paikallinen muisti jaetaan käyttöön säikeiden kesken, jolloin pelkästään

säikeellä on luku- ja kirjoitusoikeus omaan muistialueeseensa. Paikallista muistia säie käyttää, jos sen tarvitsemat tiedot eivät mahdu muihin, tietovirtasuorittimella sijaitseviin, nopeampiin muistialueisiin. [61]

Laitemuistin globaalia muistia käytetään pääasiassa näytönohjaimen työmuistina. Työmuistiin kopioidaan tarvittava tieto ennen sen suorittamista grafiikkasuorittimella. Vastaavasti työn valmistuttua tulokset voidaan hakea laitemuistista. Globaalimuisti on ainoa keino, funktiokutsujen lisäksi, jolla grafiikkasuoritin ja suoritin voivat kommunikoida keskenään. [62]

Nopea tiedonjako tietovirtasuorittimien välillä varmistetaan sillä, että kaikki tietovirtasuorittimet sijaitsevat L2-välimuistin ympärillä. L2-välimuisti toimii välimuistina laitemuistille ja pyrkii vähentämään muistihakuja laitemuistista [63]. Tarkkoja tietoja L2-välimuistin hakuajoista ei ole valmistajan puolesta saatavilla, mutta tutkimusten mukaan ne ovat n. 300 kellojakson luokkaa. L2-välimuistin koko on Fermi-arkkitehtuurissa 768 kt. [64]

Fermi-arkkitehtuurissa tietovirtasuorittimet on jaettu kahteen osaan, molemmissa osissa on 8 tietovirtasuoritinta. Tietovirtasuoritin on yksikkö joka suorittaa varsinaiset laskentatehtävät ja niihin liittyvät käskyt. Tietovirtasuoritin koostuu käskyjen hallintaan erikoistuneista yksiköistä, käskyjä suorittavista yksiköistä sekä erilaisista muisteista. Kuvassa 12 on esitetty yhden tietovirtasuorittimen rakenne. Kuvassa ylimpänä on käskyvälimuisti, joka sisältää tietovirtasuorittimen suoritettavana olevan koodin. Suorituksen aikana käskyt haetaan suoraan käskyvälimuistista. [65]



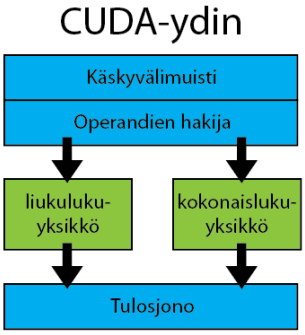
Kuva 12. Fermi-arkkitehtuurin tietovirtasuorittimen rakenne [54].

Kuten edellä mainittiin, tietovirtasuoritin suorittaa säikeitä SIMT-arkkitehtuuriin perustuen. SIMT-arkkitehtuurissa jokainen tietovirtasuorittimen säie suorittaa saman käskyn samalla kellojaksolla rinnakkain. Jokainen säie operoi kuitenkin eri datalla. Jos jonkin säikeen käsky eroaa muiden säikeiden käskyistä, suoritetaan eroavat käskyt aluksi, jonka jälkeen palataan rinnakkaiseen suoritukseen. [61]

Fermi-arkkitehtuuri toteuttaa SIMT-arkkitehtuurin suorittamalla 32 säiettä rinnakkain. Tätä 32 säikeen joukkoa kutsutaan nipuksi (warp). Lisäksi jossain yhteyksissä, kuten muistihakujen yhdistämisessä, käytetään puolinippua (half-wrap) joka sisältää 16 säiettä. Tietovirtasuoritin saa niput grafiikkaprosessorin GigaThread™-moottorilta. GigaThread™-moottori paketoi säienipun säielohkoista. Säielohkot jaetaan nippuihin niin, että jokainen nippu sisältää peräkkäisen tunnuksen sisältävät säikeet. Tietovirtasuoritin toimii tehokkaimmillaan, kun säienipun säikeet suorittavat käskyjä joiden muistihaut kohdistuvat peräkkäisiin osoitteisiin muistialueella. [58]

Jokainen tietovirtasuoritin sisältää kaksi aikataulutinta (warp scheduler) sekä käskyn suorittajaa (instruction dispatch unit), jolloin kaksi säienippua voidaan ajaa samanaikaisesti. Aikatauluttaja valitsee jokaiselle kellojaksolle molemmista säienipuista yhdet käskyt suori-

tettavaksi. Aikatauluttajan tärkein tehtävä on muistiviiveiden piilottaminen, tämä tapahtuu valitsemalla suoritettavaksi käskyjä joiden ei tarvitse odottaa muistinhakujen valmistumista. [57]



Kuva 13. CUDA-ydin Fermi-arkkitehtuurissa. [60].

Nippujen käskyt voidaan suorittaa CUDA-ytimissä, 16:ta lataus/tallennusyksikössä (LD/ST) tai neljässä erikoislaskentayksikössä (SFU, Special Function Unit). Lataus/tallennusyksiköt laskevat lähde/kohde –osoitteen 16 säikeelle kellojaksolla ja voivat ladata/tallentaa tietoa välimuistiin tai DRAM-muistiin. Erikoislaskentayksiköt suorittavat esimerkiksi sini- tai kosinilaskutoimituksia. Erikoislaskentayksikkö suorittaa sille määrätyt käskyt itsenäisesti hidastamatta muuta käskyjonoa. SFU-yksiköt suorittavat parhaimmillaan säienipun kahdeksassa kellojaksossa. [60]

Jokaisella CUDA-ytimellä, kuva 13, on suoritusliukuhihna kokonaislukulaskentaan (ALU, Arithmetic Processing Unit) sekä liukulukulaskentaan (FPU, Floating Point Unit). CUDA-yksikkö voi kuitenkin suorittaa vain jommankumman laskentakäskyn kellojaksolla. Fermi-arkkitehtuurissa liukulukulaskenta täyttää IEEE 754-2008 liukulukustandardin, mikä lisää liukulukulaskennan tarkkuutta aikaisempiin grafiikkasuorittimiin verrattuna. Vaikka tarkkuus liukulukulaskennassa on parantunut ei laskennan suorituskyky ole kärsinyt. Liukulukulaskennan tarkkuus vaikuttaa merkittävästi mm. tieteellisiä laskutoimituksia suoritettaessa, tätä tarkastellaan myöhemmin ohjelmointiominaisuuksista puhuttaessa. [60]

Tietovirtasuorittimen sisäinen muisti koostuu rekistereistä ja datamuistista, lisäksi tietovirtasuorittimilla on luku- ja kirjoitusoikeus grafiikkasuorittimella sijaitsevaan DRAM-laitemuistiin. Fermi-arkkitehtuuri määrittelee rekisterin kooksi 32768 32-bittistä rekisteriä

tietovirtasuoritinta kohden. Rekisterit jaetaan säikeiden käyttöön ja niiden käsittely on osa suoritusliukuhihnaa. Datamuisti koostuu jaetusta muistista sekä L1-välimuistista. Osion jakoa voidaan muuttaa ohjelmallisesti kahden asetuksen (16/48kt tai 48/16kt) välillä. Datamuistin toteutus on tehty pankeilla, joita on yhtä monta kuin säieryhmän säikeitä (32 kpl) [60].

Tietovirtasuorittimen sisäisistä muisteista nopeampi on 32-bittinen rekisteri. Rekisterin luku- tai kirjoitustapahtuma kestää noin 20 kellojaksoa [66]. Koska rekisterit jaetaan kaikkien tietovirtasuorittimen säikeiden kesken, vaikuttaa niiden käyttö suoraan yhtä aikaa ajettavien säikeiden määrään. [61]

Datamuistin jaon avulla voidaan määrittää, kuinka paljon muistia ajettava ohjelma saa käyttöönsä (jaettu muisti, shared memory), ja kuinka paljon muistia tietovirtasuorittimen sisäiset toiminnot saavat omaan käyttöönsä (L1-välimuisti, L1-cache). Jaetun muistin tarkoitus on tiedon jakaminen tietovirtasuorittimen säikeiden välillä. Käyttämällä jaettua muistia säikeiden tarvetta käyttää hitaampaa DRAM-muistia voidaan vähentää. Datamuisti toimii parhaimmillaan yhtä nopeasti kuin rekisterit. L1-välimuisti taas nopeuttaa globaalinmuistin käsittelyä tallentamalla haettua tietoa. Ohjelmasta riippuen L1- ja jaetun muistin suhteella voi olla merkittäviä vaikutuksia ohjelman suorituskykyyn. [54, 58].

Tietovirtasuorittimessa on myös muistialueet tekstuuri- ja piirtopintamuisteille. Näiden muistien tarkoituksena on optimoida 2D- ja 3D-piirtoa. Niiden käsittely ohitetaan tässä työssä koska ne eivät ole työn kannalta oleellisia.

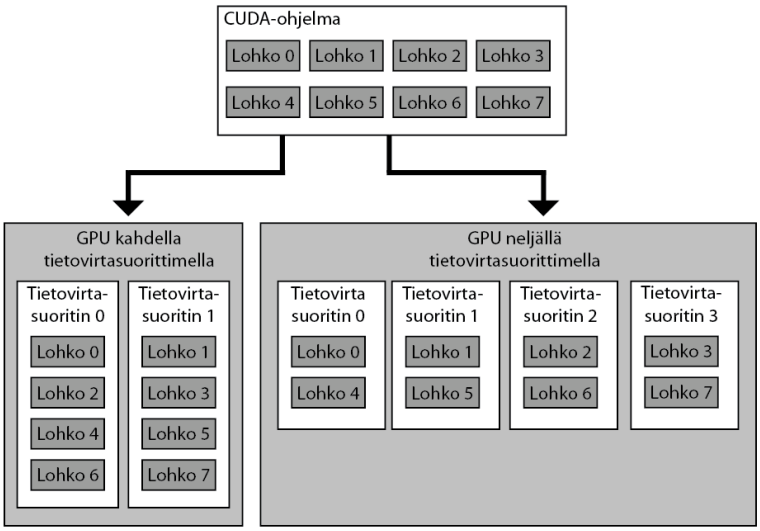
3.3 CUDA-ohjelmointi

NVIDIA julkaisi ensimmäisen version rinnakkaisen laskennan CUDA-kehitysympäristöstä vuonna 2006. CUDA:n tarkoitus on mahdollistaa monimutkaisten ongelmien ratkaisu käyttämällä hyväksi grafiikkasuoritinta. CUDA on rakennettu C/C++ ohjelmointikielen päälle, jolloin puhdasta C/C++ koodia voidaan ajaa CUDA ympäristössä. Laajennuksena C/C++ kirjastoihin NVIDIA on tuonut omat CUDA-kirjastot. CUDA:n rinnakkaisen ohjelmoinnin malli on suunniteltu niin, että C-ohjelmointikielestä siirtyminen CUDA-ohjelmointiin olisi mahdollisimman helppoa. CUDA toimii myös muissa ohjelmointiympäristöissä, kuten

JAVA, Fortran, Python, DirectCompute ja OpenACC. Lisäksi CUDA:lla on toteutettu useita kirjastoja, joilla päästään hyödyntämään grafiikkaprosessorin tehoja esimerkiksi MATLAB-ohjelmassa. Tässä työssä käytetään CUDA:n C/C++ laajennusta. [58]

CUDA-ohjelmointiin NVIDIA tarjoaa CUDA Toolkit -paketin. Kehitysympäristö voidaan asentaa OS X, Linux ja Windows -käyttöjärjestelmiin. Kehitysympäristön lisäksi tarvitaan CUDA-yhteensopiva grafiikkasuoritin. Tieto CUDA-yhteensopivista grafiikkasuorittimista löytyy NVIDIAn verkkosivuilta: http://developer.nvidia.com/cuda-gpus. CUDA Toolkit-kehitysympäristö tarjoaa ohjelmoijalle kaiken mitä grafiikkasuorittimen ohjelmointiin tarvitaan, kuten kirjastot sekä otsikkotiedostot, kääntäjän, debuggerin, profilerin ja muistintarkistajan. CUDA Toolkit on saatavissa osoitteesta: www.nvidia.com/getcuda.

CUDA C-ohjelmointi perustuu kolmeen yksinkertaistettuun asiaan: säienippuihin, jaetun muistin käyttöön sekä suorituksen synkronointiin. Näiden kolmen ominaisuuden avulla ohjelmoija voi hallita hienojakoisesti datarinnakkaisuutta ja säikeiden rinnakkaisuutta omassa koodissaan. Ominaisuudet myös ohjaavat ohjelmoijaa paloittelemaan ongelman aliongelmiin, jotka voidaan ratkaista itsenäisesti säienipuissa. Aliohjelmat voidaan vielä paloitella pienempiin paloihin, jotka säienipun säikeet ratkaisevat yhdessä. Jako aliohjelmiin mahdollistaa säikeiden välisen yhteistyön ratkaistaessa ongelmaa. Jokainen säienippu voidaan välittää suoritettavaksi mille tahansa grafiikkasuorittimen tietovirtasuorittimelle. Tämä mahdollistaa CUDA-koodin suorittamisen millä tahansa CUDA-yhteensopivalla grafiikkasuorittimella, koska ohjelmoijan ei tarvitse tietää kuinka monta tietovirtasuodinta grafiikkasuorittimessa on. Kuvassa 14 on esitetty CUDA-ohjelman suoritus kahdella eri grafiikkasuorittimella. Kuvassa CUDA-ohjelma on jaettu kahdeksaan säienippuun, vasemmanpuoleinen grafiikkasuoritin suorittaa säieniput kahdella tietovirtasuorittimella ja oikeanpuoleinen neljällä tietovirtasuorittimella. Säienippujen jako tapahtuu automaattisesti CUDA:n GigaThread™ -moottorin toimesta. [58].



Kuva 14. CUDA-ohjelman suoritus kahdella eri GPU:lla [58]

Varsinainen CUDA-kielen syntaksi sisältää muutamia laajennuksia tavalliseen C/C++ kieleen. CUDA-kielessä määritellään ytimiä (kernel), käyttämällä funktiolla määritettä `__global__`. Kaikki ytimen määrittelevät funktiot ovat tyypiltään paluuarvoja palauttamattomia (void). Ytimen määrittelevät funktiot suoritetaan aina asynkronisesti, eli ohjelman suoritus palautuu kutsujalle, vaikka ytimen määrittelemää toimintoa suoritetaan. [67]

Ydinfunktion suorittava säie on tietoinen omasta indeksistään suorituksen aikana erityisen `threadIdx` -muuttujan kautta. `ThreadIdx` on kolmiulotteinen vektori, joka määrittää säikeen indeksiarvon. Kolmiulotteisuutta voidaan käyttää hyväksi laskettaessa vektoreita, pinta-aloja tai tilavuuksia. Lisäksi ydinfunktiossa voidaan käyttää `blokcIdx`-muuttujaa, joka on kaksiulotteinen vektori, `blockIdx`-muuttujan avulla funktio on tietoinen siitä, missä lohkossa se suoritetaan. `BlockIdx`-muuttujan lisäksi kaksiulotteinen vektori `blockDim` sisältää tiedon, kuinka monta lohkoa on suoritettavana. Seuraavassa ytimessä lasketaan matriisisummaa. Summa lasketaan niin, että jokainen säie hakee lähdetaulukosta oman indeksinsä mukaiset arvot ja summaa ne tulosmatriisiin. Oman indeksinsä ydin selvittää `threadIdx`-muuttujan `x`- ja `y`-jäsenmuuttujista. Syötteenä ytimellä ovat float-

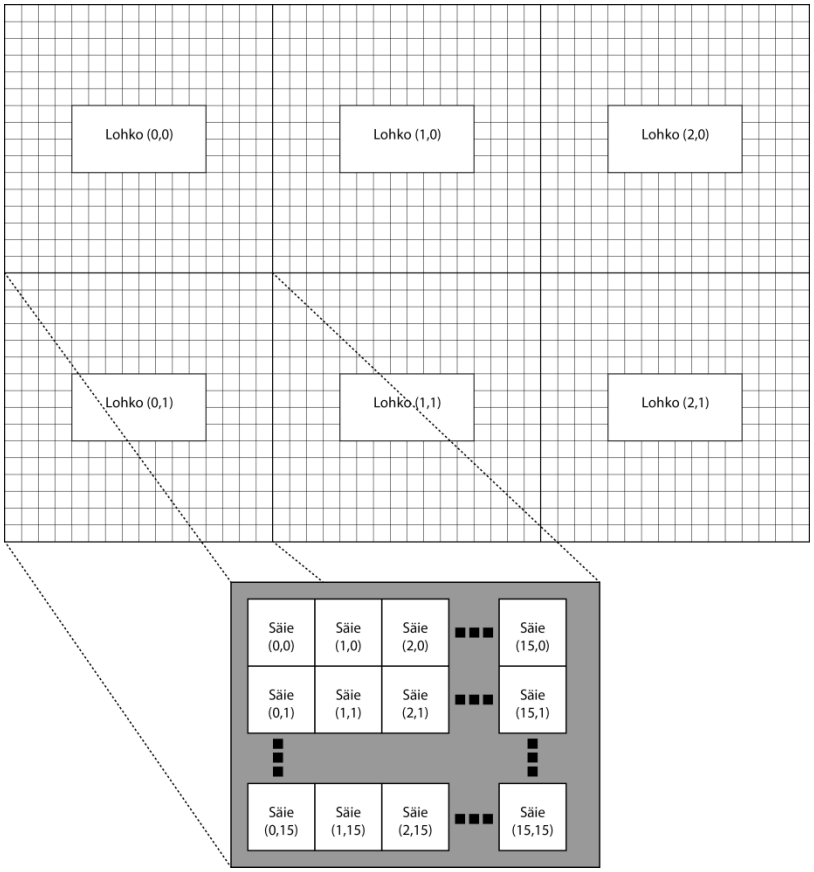
tyyppiset taulukot, joista C-taulukko on kohdetaulukko ja A- ja B-taulukot lähdetaulukoita. [58]

```
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
        int i = threadIdx.x;
        int j = threadIdx.y;
        C[i][j] = A[i][j] + B[i][j];
}
```

Rinnakkaisuuden hallinta CUDA-kielessä suoritetaan erityisellä suoritusmääritteellä. Suoritusmääre määrittää, kuinka monessa lohkossa ja kuinka monessa säikeessä ydin suoritetaan. Suoritusmäärite kirjoitetaan syntaksilla *<<< Dg, Db, Ns, S >>>* missä Dg lohkojen määrän, Db määrittää lohkon säikeiden määrän, Ns määrittää kuinka monta tavua jaettua muistia varataan tämän funktion lohkoa kohti, staattisesti varatun muistin lisäksi sekä S määrittää tietovirran. Parametreista Ns ja S ovat optionaalisia. Seuraava koodi suorittaa edellä määritellyn ytimen yhdessä lohkossa ja N-määräll säikeitä, jolloin jokainen säie suorittaa yhden vektorin alkion yhteenlaskun. [58]

```
int main()
{
    MatAdd <<<1, N>>>(A, B, C);
}
```

Koska lohkon jokainen säie suoritetaan saman tietovirtasuorittimen sisällä, säikeiden määrä lohkoa kohden on rajallinen. Fermi-arkkitehtuurissa raja on 1024 säiettä lohkoa kohden. Rajoitus johtuu siitä, että säikeet jakavat tietovirtasuorittimen sisäiset resurssit keskenään. Kuvassa 15 on esitetty ydinfunktion suoritus kuudessa lohkossa, jossa jokaisessa lohkossa suoritetaan 15x15 = 225 säiettä.



Kuva 15. Lohkojen ja säikeiden suoritus CUDA-ympäristössä [67].

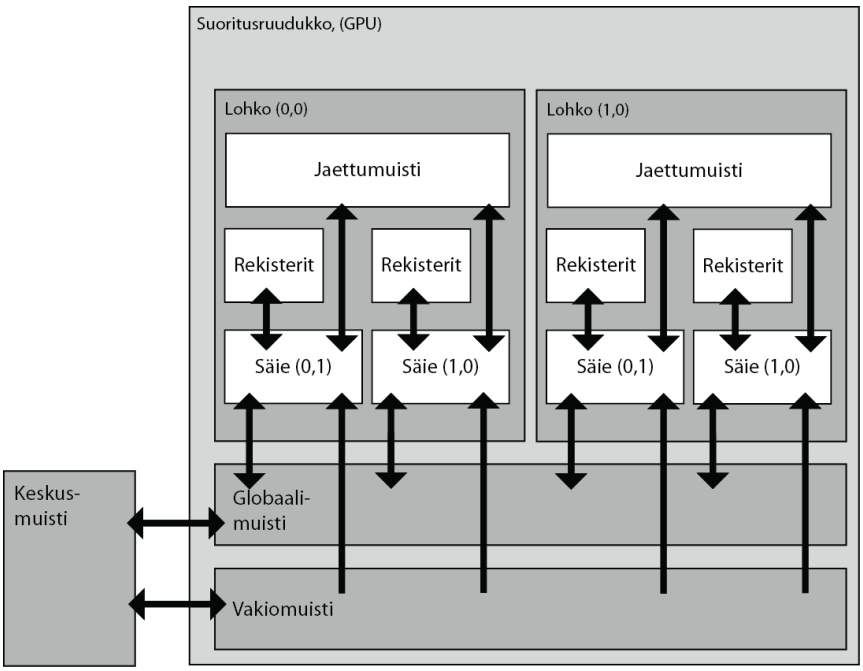
Suoritettavilla säikeillä on vaihtoehtoja, mistä muisteista ne voivat hakea ja tallentaa tietoja suorituksen aikana. Säikeen nopein käytettävissä oleva muisti on tietovirtasuorittimen sisäinen rekisteri ja toiseksi nopein säikeen yksityinen muisti. Säikeen yksityiseen muistiin tallennetaan niitä muuttujia, jotka eivät mahdu säikeen rekistereihin. Jaettumuisti on lohkon säikeiden käyttöön tarkoitettu muisti ja sijaitsee lähellä jokaista tietovirtasuoritinta. Jaettu muisti on nopein säikeen käytettävissä oleva tietovirtasuorittimen ulkopuolinen muisti. Jaetun muistin elinaika on sama kuin sen omistavan lohkon elinaika. Lohkon jaettuun muistiin voidaan alustaa tietoja joita kaikki lohkon säikeet tarvitsevat. Jaettua muistia voidaan käyttää myös säikeiden väliseen tietojen vaihtoon ja laskentatulosten tallentamiseen. Lisäksi säikeen käytössä on myös laitteen globaali DRAM-muisti. CUDA-kielessä määritellään parametrit, joilla voidaan vaikuttaa siihen mihin muistiin muuttujat halutaan

sijoittaa. Määrite `__shared__` sijoittaa muuttujan jaettuun muistiin, `__constant__` vakiomuistiin ja `__device__` laitemuistiin. Esimerkiksi lause:

```
__shared__ float cache[256];
```

varaa jaetusta muistista 256-alkoisen liukulukutaulukon käyttöön kaikille lohkon säikeille. [68]

Kuvassa 16 on esitetty kuvaus siitä, miten säikeiden, lohkojen ja koko ohjelman muistiavaruudet ovat käytettävissä. Lohkolla ja lohkon säikeillä on käytettävissä tietovirtasuorittimen rekisterit sekä jaettumuisti, nämä resurssit jaetaan kaikkien lohkon säikeiden kesken. Lisäksi kaikilla säikeillä on luku- ja kirjoitusoikeus globaalinmuistiin sekä lukuoikeus vakiomuistista.



Kuva 16. Eri muistialueiden käyttöoikeudet [69].

Globaalin muistin dynaaminen hallinta CUDA-kielessä tapahtuu `cudaMalloc()`-kutsulla, jolloin muisti varataan GPU-suorittimen muistista. `cudaMalloc()`-kutsun ensimmäinen parametri on osoitin varattuun muistialueeseen ja toinen on varattavan muistialueen koko. Kutsu palauttaa `cudaError_t` enumeraation, jossa on tieto kutsun onnistumista tai epäonnistumisesta. Palautettua osoitinta voidaan käyttää myös CPU-puolen koodissa rajoitetusti, kuten välittää eteenpäin ja suorittaa osoitinlaskentaa. Mutta CPU ei

voi lukea tai kirjoittaa suoraan GPU:n varaamaan muistiin. `cudaMalloc()`−kutsulla varattu muisti vapautetaan `cudaFree()`−kutsulla. GPU:n muistissa olevaa tietoa voidaan kopioida CPU:n muistiin ja päinvastoin `cudaMemcpy()`−kutsulla. Kutsun parametrit ovat, kohdeosoite, lähdeosoite, kopioitavan datan pituus ja kopiointisuunta. Kopiointisuuntia ovat `cudaMemcpyDeviceToHost`, joka kertoo että kohdeosoite sijaitsee CPU:n muistissa ja lähdeosoite GPU:n muistissa. `cudaMemcpyHostToDevice` toimii päinvastoin. Kopioitaessa laitemuistiosoitteesta toiseen laitemuistissa sijaitsevaan osoitteeseen käytetään `cudaMemcpyDeviceToDevice` parametria. [68, 58]

Käytettäessä CUDA:n omia globaalimuistin varaus- ja vapautusfunktiota estetään käyttöjärjestelmää sivuttamasta muistia. Tämä takaa sen, että varattu muisti pysyy aina samassa muistipaikassa. Samalla grafiikkasuoritin voi käyttää oikosiirtoa (DMA, Direct Memory Access) muistinkäsittelyyn. Oikosiirrossa muistiin voidaan kopioida tai siitä voidaan kopioida tietoa ilman, että tieto kiertää CPU:n kautta. Sivutuksen estolla on kuitenkin se haittavaikutus, että CPU ei voi siirtää muistia levylle. Tarve siirtää muistia levylle tapahtuu jos grafiikkasuorittimen omasta muistista loppuu tila. Tästä johtuen `cudaMalloc()` ja standardi C-kielen `malloc()`-kutsuja tulee käyttää tapauskohtaisesti harkiten. [67]

Suoritettavien säikeiden eli ydinfunktioiden tulee olla itsenäisesti suoritettava, koska CUDA-ympäristö suorittaa ne parhaaksi näkemässään järjestyksessä mm. suorituksen optimoimiseksi. Säikeet voivat lohkon sisällä jakaa tietoa jaetun muistin kautta tai synkronoida lohkon sisäistä tilaa `__syncthreads()`−kutsulla. Säikeiden välinen synkronointi on tärkeää jaetun muistin ympäristössä, koska sillä voidaan tahdittaa säikeiden välistä suoritusta kilpailutilanteessa. Kilpailutilanne voi syntyä kun säie *A* kirjoittaa muistipaikkaan *X*, ja ohjelmassa halutaan säikeen *B* lukevan muistipaikasta *X* vasta kun *A* on saanut kirjoituksen valmiiksi. Synkronointikutsu toimii siten, että säikeen kutsuttua synkronointikutsua jatkaa säie suoritusta vasta kun kaikki muutkin lohkon säikeet ovat kutsuneet synkronointikutsua. [67]

Edellä on esitetty CUDA:n ominaisuuksia, jotka ovat tämän työn kannalta tärkeitä. Näiden ominaisuuksien lisäksi CUDA-kirjastossa on rajapintoja mm. GPU-laitteen ominaisuuksien

kyselyyn. Kattavan katsauksen CUDA-kirjastoon tarjoaa mm. NVIDIAn julkaisema ”Introduction to CUDA C” kirja.

3.4 GPU-ohjelmoinnin ominaisuudet

Koska GPU on suunniteltu ensisijaisesti graafisen tiedon näyttämisen toteuttamiseen, tulee GPU-ohjelmoinnissa huomioida muutamia seikkoja, jotta sillä saataisiin aikaan nopeutusta. Vaikka CUDA-ohjelma saadaan toimimaan oikein millä tahansa CUDA-yhteensopivalla grafiikkasuorittimella, sen suoritusnopeus saattaa vaihdella. Grafiikkasuorittimen resurssien oikea käyttö mahdollistaa CUDA-ohjelman suorittamisen mahdollisimman tehokkaasti kohdenäytönohjaimella. Suurimmat resurssien hallintaan liittyvät suuntalinjat painottuvat siihen miten näytönohjaimen erilaisia muisteja hyödynnetään parhaalla mahdollisella tavalla.

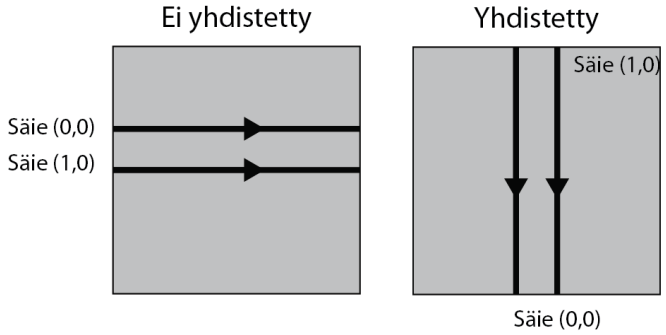
Jotta grafiikkasuoritin toimisi mahdollisimman tehokkaasti, pyritään saavuttamaan tila, jossa tietovirtasuoritin suorittaa mahdollisimman montaa säiettä yhtäaikaisesti. Säikeillä tulisi myös olla jatkuvasti tehtävänään työtä, joka edistää koodin suoritusta. Tämä tarkoittaa, että säie suorittaa sille määrättyjä käskyjä eikä ole odottamassa tietoa muistista. Koska jokaisen säienipun säikeet suorittavat samaa käskyjonoa, joudutaan kaikkien säienipun säikeiden koodi suorittamaan haarautumien kertojen määrän verran. Esimerkiksi, jos käskyjonossa on `if-then-else` rakenne ja osa säikeistä suorittaa `if`-osion ja osa säikeistä `else`-osion, suoritetaan erikseen `if`-osion suorittavat säikeet ja `else`-osion suorittavat säikeet odottavat. Vastaavasti `else`-osion säikeiden suorituksen aikana `if`-osion säikeet odottavat. Samanlainen tilanne tapahtuu myös silmukkarakenteita suoritettaessa. Jos silmukka tulee suorittaa osalla säikeitä x-kertaa ja osalla säikeitä y-kertaa, niin että esim. $y = x+1$, suorittavat kaikki säikeet silmukan aluksi x-kertaa. Tämän jälkeen vaaditaan kaksi suorituskertaa, y-määrän suorittavat säikeet suorittavat silmukan ja toinen suorituskerta suorittaa ne säikeet, jotka eivät suorita silmukkaa. [69]

Koska grafiikkasuoritin sijaitsee erillään tietokoneen keskusmuistista ja suorittimesta, tulee GPU-ohjelmoinnissa ottaa huomioon suorittimen, keskusmuistin ja grafiikkasuorittimen yhdistävän väylän kapasiteetti sekä grafiikkasuorittimen sisäisten tiedonsiirtoväylien kapa-

siteetti. grafiikkasuoritinta ohjelmoitaessa tuleekin erityisesti tarkastella tiedonsiirron tarpeellisuutta GPU:n ja CPU:n muistien välillä sekä sitä, kuinka usein ja millaisissa paloissa tietoa siirretään muistien välillä. Muistihakuviiveiden piilottamisella pyritään suorittamaan muita säikeitä sillä aikaa, kun jonkin säie odottaa tietoa globaalista muistista. Muistiviiveiden piilottaminen on mahdollista silloin, kun suoritettavia säikeitä on enemmän kuin suorittavia CUDA-ytimiä. Suoritettavan säikeen vaihtaminen toimii nopeasti NVIDIAn Fremi-arkkitehtuurissa. Arkkitehtuuri tukee myös samanaikaisesti suoritettavia säikeitä aina 1536 säikeeseen asti. Tällöin yksi tietovirtasuoritin voi suorittaa esim. 6:tta säienippua, joissa jokaisessa on 256 säiettä. Tietovirtasuoritin jakaa kuitenkin omat resurssinsa, kuten muistin, sillä yhtäaikaisesti suoritettavien säikeiden kesken. Intensiivinen paikallisen muistin tai rekisterien käyttäminen vähentää näin ollen yhtäaikaisesti suoritettavien säikeiden määrä. Muistihakujen määrää voidaan vähentää myös käyttämällä järkevästi hyväksi tietovirtasuorittimen jaettua ja vakiomuistia. Vaikka järjestelmä käyttääkin jaettua muistia ja vakiomuistia oletusarvoisesti hyväkseen, ohjelmoijan kannattaa usein käyttää `__shared__` ja `__constant__` -määrityksiä muistin käytön ohjaamiseen. [69]

Jos säikeet kuitenkin joutuvat käyttämään grafiikkasuorittimen globaalimuistia, tulee ottaa huomioon, että vaikka tietovirtasuorittimien yhteys globaalimuistiin on nopea, voivat sadat säikeet kilpailla saman muistiosoitteen tiedosta. Globaalimuisti on NVIDIAn Fermi-arkkitehtuurissa toteutettu DRAM-muistina, jossa erittäin herkät kondensaattorit tallentavat tiedon oman muistiosoitteensa bittiarvosta. Bittiarvon lukemiseksi käytetään sensoria, joka tunnistaa onko kondensaattorin lähettämä arvo 0 vai 1. Koska lukuprosessi yhdestä DRAM-muistipaikasta on hidas, piilottavat DRAM-muistit viiveen suorittamalla peräkkäistä lukua. Kun yhdestä DRAM-muistipaikasta luetaan tietoa, luetaan automaattisesti myös peräkkäisten muistipaikkojen tiedot. Tämän jälkeen kaikki luetut tiedot välitetään nopeaa siirtoväylää pitkin tietovirtasuorittimelle. Näin saavutetaan mahdollisimman lähelle DRAM-muistin optimaalista tiedonsiirtokykyä vastaava toiminta. Fermi-arkkitehtuurissa maksimaalinen väylän käyttö saavutetaan, kun säienipun kuusitoista säiettä suorittavat saman käskyn peräkkäisille muistipaikoille. Toisin sanoin, jos säienipun säie 0 käsittelee muistia osoitteessa N, säie 1 muistiosoitteessa $N+1$, säie 2 muistiosoitteessa $N+2$, jne. Tällöin muistihaut voidaan niputtaa yhdeksi muistihauksi. Kuvassa 17 on esitetty kaksi eri tilannetta, jossa säikeet käsittelevät muistia eri tavoin. Vasemmanpuoleinen kuva esittää

tilannetta, jossa säikeet suorittavat operaatioita muistialueen eri riveillä sijaitseville muistiosoitteille. Tällöin säikeiden muistihakuja ei voida yhdistää. Oikeanpuoleisessa kuvassa säikeet suorittavat operaatioita muistialueen peräkkäisille muistiosoitteille, mikä mahdollistaa muistihakujen yhdistämisen.



Kuva 17. Muistihakujen yhdistäminen [69].

Jos muistihakujen yhdistäminen ei ole mahdollista, esim. algoritmin logiikan vuoksi, voidaan tietovirtasuorittimen jaettua muistia hyödyntää muistihakujen nopeuttamiseen. Jaettua muistia voidaan käyttää nopeasti ilman, että muistihakuja tarvitsee yhdistää. Tällöin tieto DRAM-muistista kopioidaan lohkoissa jaettuun muistiin. Kopiointi lohkoissa tehdään niin, että muistihaut voidaan yhdistää. Kopioinnissa jokainen säie hakee oman lohkonsa indeksin mukaisen tiedot DRAM-muistista. Koska lohko täytetään rivi riviltä, vasemmalta oikealle, voidaan rivin täyttämisen muistihaut voidaan yhdistää. Kun kaikki säikeet ovat suorittaneet muistihaut, voivat säikeet jatkaa suorittamaan varsinaista laskentaa käyttäen jaetussa muistissa olevaa lohkoa. [69]

Globaalimuistin tiedonsiirtokyvyn rajoitteita voidaan kiertää myös hakemalla tietoa samaan aikaan, kun sitä käsitellään. Tämä tapahtuu niin, että ytimet aloittavat tiedon haun seuraavan kierroksen ohjelmalohkoa varten ennen kuin suorittavat nykyisen ohjelmalohkon. Oletuksena on, että kun seuraavaa ohjelmalohko suoritetaan, on globaalimuisti ehtinyt palauttaa haettavan tiedon. Tämä vaatii oikeanlaista ohjelmalogiikkaa sekä säikein synkronointia. [69]

Oikeanlaisen muistinhallinnan lisäksi ohjelmoijan tulee huolehtia siitä, että tietovirtasuorittimen resurssit käytetään hyödyksi parhaalla mahdollisella tavalla. Fermi-arkkitehtuurissa voidaan suorittaa kerralla korkeintaan 65535 säielohkoa ja jokainen säielohko voi sisältää korkeitaan 1024 säiettä [67]. Yhdeltä tietovirtasuorittimelta voidaan kerralla varata tilaa korkeintaan 1536 säikeelle sekä kahdeksalle säielohkolle [70]. Maksimaalisten säie- ja lohkomäärien saavuttamien on kuitenkin usein turhaa, sillä lohko- ja säiekoko tulee valita ongelman tehokkaan ratkaisun vaatimusten mukaan, tätä tarkastellaan myöhemmin. Suoritettavien säikein määrää tietovirtasuoritinta kohden rajoittaa merkittävästi muistinkäyttö ydintä kohden. Tämän vuoksi ohjelmoijan tulee olla tietoinen miten muuttujien määrittely vaikuttaa siihen mistä muistista tila muuttujalle varataan. Taulukko 1 kertoo mistä muistista muuttujan tila varataan, muuttujan näkyvyyden ja elinajan. [69]

Taulukko 1. Tietoja muuttujien määrittelystä [69].

Muuttujan määrittely	Muisti	Näkyvyys	Elinaika
Automaattiset muuttujat, ei taulukot	Rekisteri	Säie	Ydin
Automaattiset taulukot	Paikallinen	Säie	Ydin
__device__, __shared__, int SharedVar;	Jaettu	Lohko	Ydin
__device__, int GlobalVar;	Globaali	Suoritusjoukko	Ohjelma
__device__, __constant__, int ConstVar;	Vakio	Suoritusjoukko	Ohjelma

Rekisteriä ja paikallista muistia käyttävät muuttujat ovat käytettävissä vain ne varaavan säikeen toimesta, koska säikeen elinaika on sama kuin ytimen elinaika myös säikeen varaamat muuttujat saavat saman elinajan. Jaetun muistin muuttujat sen sijaan ovat kaikkien lohkon säikeiden käytettävissä, myös niillä on ytimen elinaikaa vastaava elinaika. Sen sijaan globaali- tai vakiomuistista varatut muuttujat ovat kaikkien säikeiden käytettävissä riippumatta siitä missä lohkossa ne sijaitsevat, näiden muuttujien elinaika on myös yhtä pitkä kuin koko ohjelman elinaika. Tämä tarkoittaa sitä, että ne ovat käytettävissä myös suoritettavan ydinfunktion ulkopuolella.

Tietovirtasuorittimen sisäisiin resursseihin kuuluu mm. rekisterit, joita yhdessä tietovirtasuorittimessa on 32768 kappaletta. Yhteen rekisteriin voi tallentaa 32-bittisen arvon [60].

Kun tiedetään kuinka monta rekisteriä suoritettava ydin käyttää, voidaan laskea kuinka monta säiettä ja säielohkoa voidaan suorittaa tietovirtasuoritinta kohden. Jos suoritetaan 16x16 säiettä yhdessä lohkossa ja ydin käyttää 32 rekisteriä, tulee lohkoa kohden käytettäväksi 16x16x32 = 8192 rekisteriä. Tällöin yksi tietovirtasuoritin voi, ainakin rekisteri tarpeen mukaan, suorittaa neljä lohkoa ja 1024 säiettä. Jos käytettäviä rekistereitä ydintä kohden onkin 33, vaadittaisiin 33792 rekisteriä neljän lohkon suorittamiseen yhdellä tietovirtasuorittimella. Tällöin tietovirtasuoritin muuttaa automaattisesti kerralla yhtäaikaisesti suoritettavien lohkojen määrää alaspäin. Tällöin vaaditaan 25344 rekisteriä, mikä jää alle maksimirajan, mutta suoritettavien säikeiden määrä tippuu kuitenkin 16x16x2 = 768:aan. Tässä tapauksessa tippuu samalla tietovirtasuorittimella yhtäaikaisesti suoritettavien säikeiden määrä 25%, vain koska ydin käyttää yhden rekisterin enemmän. [69]

Fermi-arkkitehtuuri määrittää grafiikkasuorittimelle 64 kilotavua vakiomuistia, jota ohjelmoija voi käyttää määritteillä. Vakiomuisti varataan globaalimuistista, mutta sitä käsitellään välimuistin kautta. Vakiomuistista voidaan varata tietoa, joka pysyy muuttumattomana ytimen suorituksen aikana. Tällä tavoin voidaan vähentää globaalimuistin käyttöä. Vakiomuisti toimii niin, että säikeen lukiessa tietoa vakiomuistista luetaan tieto tietovirtasuorittimen välimuistiin, jolloin se on käytettävissä myös muilla säikeillä välimuistin kautta. Vakiomuistilla on myös ominaisuus lähettää säikeen lukema tieto muille säikeille. Tällöin yhden säikeen lukiessa tietoa vakiomuistista myös saman säienippupuolikkaan säikeet saavat saman tiedon käyttöönsä, jolloin säästetään 15 lukukertaa (puolikkaan säienipun koko on 16 säiettä). Vakiomuistin käytöllä voidaan vähentää liikennettä tietovirtasuorittimen ja globaalimuistin välillä, mutta ohjelmoijan tulee huolella valikoida miten vakiomuistia käytetään. Edellä mainituista ominaisuuksista johtuen, jos puolikkaan säienipun säikeet lukevat kaikki eri vakiomuistin osoitetta, lähetetään luetut tiedot myös puolikkaan muille säikeille mistä aiheutuu tarpeetonta liikennettä. Vakiomuistista luku on myös peräkkäistä, jolloin säikeiden lukupyyntöjä ei voida yhdistää kuten muuta globaalimuistia käytettäessä. Vakiomuistista on hyötyä mm. simulaatioissa jolloin siihen voidaan tallentaa valmiiksi laskettua tietoa. [58]

Ytimen paikallinen muisti on pelkästään ytimen käytettävissä olevaa muistia. Kääntäjä päättää, varataanko paikalliselle muistille osa tietovirtasuorittimen rekistereistä vai käyte-

täänkö paikallisena muistina globaalimuistista varattua tilaa. Paikallista muistia käytetään myös silloin, kun ytimen varaamat muuttujat eivät mahdu säikeelle varattuihin rekistereihin. Parhaimmillaan paikallinen muisti on siis nopeaa rekistereiden käytön ansiosta, ja pahimmassa tapauksessa välimuistin kautta kulkevaa globaalimuistia. [65, 58]

Tietovirtasuorittimen oma 64 kilotavun muisti jaetaan L1-välimuistin ja jaetun muistin välillä joko suhteessa 16/48kt tai 48/16kt. Vaikka ohjelmoijalla on pääsy myös L1-välimuistiin suositellaan, että L1-välimuisti jätetään grafiikkasuorittimen hallintaan parhaan suorituskyvyn aikaansaamiseksi. Jos ohjelman muistinkäyttö on hyvin tiedossa voidaan L1-välimuistin kooksi määrittää 16kt. Jos taas ohjelman muistinkäytöstä ei ole varmuutta, kannattaa L1-välimuisti pitää 48kt:ssa, jotta globaalimuisti toimii nopeammin. Jaetun muistin käyttö vaikuttaa siihen, kuinka monta säiettä tietovirtasuoritin voi suorittaa. Koska tietovirtasuoritin voi suorittaa kahdeksan säielohkoa kerralla voi yksi säielohko käyttää maksimissaan 48 / 8 = 6 kt jaettua muistia, tai jos L1-välimuistille on määritetty 48kt niin 16 / 8 = 2kt. Jos säielohkot käyttävät enemmän muistia vähentää järjestelmä automaattisesti tietovirtasuorittimella yhtäaikaisesti suoritettavien säikeiden määrä. Jaettu muisti on toteutettu samankokoisina muistimoduuleina, joita voidaan käsitellä yhtäaikaisesti. Tämä lisää paikallisen muistin kaistanleveyttä. Paikallisen muistin kaistanleveys on *n*-kertainen yhden muistimoduulin kaistanleveyteen, jos *n*-kappaletta muistinkäsittelypyyntöjä suoritetaan jokainen eri muistiosoitteeseen. Kuitenkin jos samaan muistiosoitteeseen tulee useampi pyyntö samanaikaisesti tapahtuvat pyynnöt peräkkäisesti. [69, 58]

Jotta ohjelmoija voi saavuttaa optimaalisen säikeiden lukumäärän NVIDIA tarjoaa CUDA Occupancy Calculator Excel-taulukon, jolla voidaan arvioida, http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

NVIDIAn CUDA-arkkitehtuuri tarjoaa ohjelmoijan käyttöön lisäksi kaksinkertaisen tarkkuuden liukulukulaskennan, joka ei kuitenkaan vaadi suoritinytimiltä merkittävästi enempää suoritusaikaa verrattuna yksinkertaisen tarkkuuden liukulukulaskentaan. Fermi-arkkitehtuurin liukulukulaskenta täyttää IEEE 754-2008 standardin. Vuonna 2008 julkaistussa standardissa määritellään yhdistetty kerto-summa operaatio (FMA, fused multiply-add). FMA menetelmä eroaa aikaisemmista menetelmistä sillä, että siinä suoritetaan vain yksi pyöristäminen kerto-summa operaatiota kohden. Aikaisemmissa toteutuksissa pyöris-

tyksiä on tehty kaksi molemmille operaatioille, kertominen ja summa, erikseen. Kaikki nykyiset CPU-suorittimet eivät tue FMA-operaatiota, joten se antaa Fermi-arkkitehtuurille edun tarkkaa liukulukulaskentaa vaativissa ohjelmissa. [71]

Edellä mainittujen ohjelmointiominaisuuksien johdosta, ohjelmoijan tulee tapauskohtaisesti harkita, soveltuuko ohjelma ratkaistavaksi grafiikkasuorittimella ja onko grafiikkasuorittimen tuottama mahdollinen nopeutus oikeassa suhteessa työmäärään. Lisäksi ohjelmoijan tulee olla hyvin tietoinen ohjelman muistinkäytöstä, koska sillä on huomattava vaikutus ohjelman suoritukseen grafiikkasuorittimella. Väärillä ohjelmointitavoilla grafiikkasuorittimella voidaan saada aikaan hidas ohjelma, joka oikeilla ohjelmointitavoilla voisi nopeutua huomattavasti. Usein käykin niin, että ensimmäinen ohjelmointiyritys ei välttämättä ole paljon CPU:lla suoritettavaa ohjelmaa nopeampi, mutta kun ohjelmaa optimoidaan saadaan grafiikkasuorittimesta täysi hyöty ja ohjelma nopeutuu merkittävästi.

4 Optimointi GPU-laskennassa

Yleisimmät syyt siihen miksi koodi ei suoriudu tehokkaammin grafiikkasuorittimella johtuvat siitä, että joukko säikeitä joutuu odottamaan yksittäisiä säikeitä. Tai siitä, että säikeet joutuvat odottamaan muistista haettavaa tietoa. Tällaiset tilanteet ovat mahdollisia mm. silloin kun globaalisynkronointia käytetään väärin tai kun koodi haarautuu tarpeettomasti. Muistihakujen suhteen tilanteet muodostuvat ongelmiksi silloin kun tietoa haetaan satunnaisesti tai turhaan.

4.1 Optimoinnin yleisiä periaatteita

Grafiikkasuorittimella suoritettava koodi suoriutuu tehokkaimmin kun seuraavia käytäntöjä noudatetaan [72]:

- Grafiikkasuorittimella on jatkuvasti suoritettavana satoja säikeitä.
- Tietovirtasuorittimien muistin käyttö on maksimaalista.
- Muistihakuja tehdään yhdistetysti.
- Laajaa synkronointia vältetään.

Saavuttaakseen tilanteen, jossa tietovirtasuorittimella on jatkuvasti säikeitä suoritettavana, tulevat suoritusympäristön parametrit olla oikeita. Samoin ytimien koodin tulee olla optimoitua niin, että tietovirtasuorittimen resurssit riittävät mahdollisimman suurelle joukolle säikeitä. Muistinkäytössä tulee huomioida globaalimuistin käytön minimoiminen käyttämällä jaettua muistia tehokkaasti hyväksi. Jotta jaettu muisti saadaan parhaaseen mahdolliseen käyttöön, voidaan esilasketun tiedon määrää vähentää ja laskea tietoa suorituksen aikana, jos tiedon sisältö sen mahdollistaa. Jos globaalimuistia kuitenkin käytetään, tulee muistinkäyttö optimoida. Globaalin muistin käytön optimointi tehdään niin, että muistihaut ovat yhdistettyjä ja mahdollisimman tehokkaita. Jaettua muistia käytettäessä tulee kiinnittää huomiota tilanteisiin, joissa monta säiettä käsittelee samanaikaisesti saman muistimoduulin tietoja. Nämä tilanteet aiheuttavat viiveitä säikeiden suorittamisessa.

Optimoitaessa koodia CUDA-ympäristöön tulee pitää mielessä muutamia yleiskustannuksia, joita aiheutuu koodia suoritettaessa. Kopioitaessa tietoa CPU:n ja GPU:n muistien välillä (`cudaMemcpy`-komento) saavutetaan täysi kaistanleveys vasta kun tarpeeksi paljon tietoa on kopioitu. Kun tietoa kopioidaan vähän, muistinkäyttö ei saavuta tarpeeksi suurta intensiteettiä, eikä muistihakuja voida toteuttaa tehokkaasti esim. yhdistämisen suhteen. Globaalin muistin käytössä on myös tietty yleiskustannus, yleiskustannuksen takia muistinkäyttö on kustannustehokkaampaa isoilla tietomäärillä. Isoilla tietomäärillä yleiskustannuksen suhde itse kopioinnin kustannukseen jää olemattoman pieneksi. Kaistanleveys, tai sen mahdollinen vajaus, tulee ottaa huomioon määriteltäessä pienintä mahdollista ratkaistavaa ongelmaa. Koska pienellä ongelmalla hitaampi tiedonsiirto voi aiheuttaa tilanteen, jossa suorittaminen GPU:lla ei ole kannattavaa. Toinen yleiskustannus aiheutuu CUDA-ympäristön alustamisesta. CUDA-ympäristö alustetaan, kun ensimmäinen CUDA-kutsu tehdään. Tästä aiheutuva yleiskustannus on kymmenistä millisekunneista satoihin millisekunteihin. Alustan alustaminen on kuitenkin toimenpide, jota ei voida välttää. Alustan alustamisesta aiheutuva kustannus vaikuttaa myös siihen, kuinka pieniä ohjelmia grafiikkasuorittimella kannattaa suorittaa sekä kustannus tulee ottaa huomioon tehtäessä mittauksia. Kolmas yleiskustannus aiheutuu jokaisen uuden ytimen suorittamisesta. Tämä viive on kuitenkin vain mikrosekuntien luokkaa ja voidaan näin ollen sisällyttää ytimen suoritusaikaan. [64]

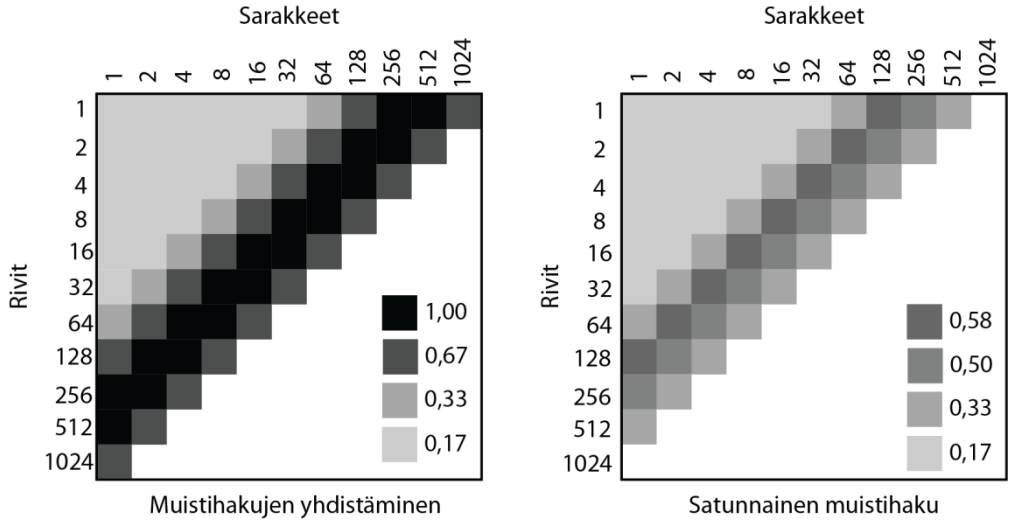
Muistihaun kokonaiskesto muodostuu muistihaun viiveestä sekä muistin kaistaleveydestä. Muistihakuihin tulee liittää yleiskustannus, jonka arvo on kuitenkin huomaamattoman pieni varsinaiseen hakuaikaan verrattuna. Globaalia muistia käytettäessä grafiikkasuoritin kohtainen raja-arvo sille, kuinka monta ydintä voi hakea tietoa globaalista muistista samanaikaisesti, joudutaan selvittämään erikseen. Globaalin muistin ominaisuuksista johtuen globaalimuistia voidaan käyttää tiettyyn raja-arvoon asti niin, että se nopeuttaa laskentaa. Raja-arvon jälkeen globaalin muistin suorituskyky laskee, koska sille osoitettuja pyyntöjä ei voida enää yhdistää tai välimuisti täyttyy. Tällöin pyynnöt joudutaan lähettämään peräkkäisesti. Raja-arvo ei kuitenkaan päde, silloin jos globaalimuisti pystyy edellisessä luvussa kuvattujen ominaisuuksien mukaan yhdistämään muistipyyntöjä. Tämä on mahdollista, kun puolikkaan säienipun muistipyynnöt kohdistuvat peräkkäisiin muistiosoitteisiin. Jaetun muistin kohdalla samaa ongelmaa ei esiinny, mutta jaetun muistin käyttöä rajoittaa kerralla

tietovirtasuorittimelle osoitettujen ydinten määrä sekä ydinten käyttämien rekisterien määrä. Jotta optimaaliseen jaetun muistin käyttöön päästään, tulee tietovirtasuorittimella olla niin paljon ytimiä osoitettuna, että jaettu muisti on jatkuvasti käytössä. [64, 73]

Ytimen suoritusaikaa voidaan pienentää suunnittelemalla ytimen koodia niin, että tietovirtasuoritin pystyy optimoimaan suorituksen tehokkaasti. Ytimen suoritukseen vaikuttaa muun muassa se, riippuvatko käskyt toisistaan. Kaksi toisistaan riippumatonta käskyä voidaan samasta säienipusta suorittaa samalla kellojaksolla. Riippumattomia käskyjä suoritettaessa, grafiikkasuorittimen ominaisuuksista riippuen, käskyjen suoritusnopeus kasvaa tiettyyn rajaan asti. Jos käskyt taas riippuvat toisistaan, CUDA-ympäristö pystyy optimoimaan itsenäisesti laskennan tehokkuutta käytettäessä pieniä silmukkarakenteita. Tätä menetelmää kutsutaan "loop unrolling”:ksi, jolloin CUDA-kääntäjä optimoi suoritettavia käskyjä niin, että ne suoriutuvat parhaimmalla mahdollisella tavalla tietovirtasuorittimen rakenteessa. [64, 73]

Vaikka ytimen käskyjono ja muistinkäyttö saataisiin optimoitua mahdollisimman pitkälle vaikuttavat suoritusaikaan suuresti myös suorituksen aikaiset parametrit. Fermi-arkkitehtuuri mahdollistaa korkeitaan 1024 säiettä säielohkoa kohden, 1536 säiettä tietovirtasuorinta kohden sekä kahdeksan yhtä aikaa suoritettavaa säielohkoa tietovirtasuorinta kohden. Jotta tietovirtasuorittimien ytimet saadaan työllistettyä mahdollisimman tehokkaasti, tulee säielohkojen koko ja lohkojen määrä valita tapauskohtaisesti oikein. Erilaiset säielohkokoot tuottavat erilaisen käyttöasteen tietovirtasuorittimille. Säielohkojen säiemäärän valinnassa tulee muistaa, että jos käytetään esim. 8x8 säiettä lohkoa kohden, ja tietovirtasuoritin voi suorittaa korkeintaan 1536 säiettä kerralla, vaaditaan 24 lohkoa tietovirtasuoritinta kohden. Tämä ei kuitenkaan ole mahdollista, koska yhdellä tietovirtasuorittimella voidaan suorittaa korkeintaan kahdeksan säielohkoa samanaikaisesti. Jos käytettäisiin 16x16 säiettä lohkoa kohden, maksimaalinen säiemäärä saavutettaisiin käyttämällä kuutta lohkoa. Tällöin kaikki lohkot voidaan ajaa samanaikaisesti yhdellä tietovirtasuorittimella. Säielohkon kokoa valittaessa tulee muistaa, että säiemäärä pitäisi olla jaollinen 32:lla, jotta kaikille tietovirtasuorittimen suorittimille riittää työtä. Jotta tietovirtasuorittimet saadaan työllistettyä täysin, tulee tietovirtasuorittimen maksimi säiemäärän olla jaollinen säielohkon säikeiden määrällä sekä lohkon säiemäärän tulee olla jaollinen 32:lla. Lisäksi lohko-

koon tulee mahdollistaa mahdollisimman monta yhdellä tietovirtasuorittimella suoritettavaa lohkoa, ei kuitenkaan enempää kuin kahdeksan. Säielohkon koolla on myös suora vaikutus muistihakujen tehokkuuteen. Kuten luvussa 3 todettiin, muistinhakujen yhdistämisellä on suuri vaikutus suorituskykyyn. Suorituskykyyn vaikuttaa oleellisesti myös tietovirtasuorittimien käyttöaste. Kuvassa 18 esitetään muistihakujen vaikutus tietovirtasuorittimien käyttöasteeseen matriisikertolaskua suoritettaessa. Kuvien asteikot vasemmalla ja ylhäällä edustavat käytettävän lohkokoon rivien ja sarakkeiden määrää. Vasemman puoleinen kuva esittää tilannetta, jossa ytimien muistihaut voidaan yhdistää globaalimuistia käytettäessä, toisin sanoen muistihaut kohdistuvat peräkkäisiin muistiosoitteisiin. Oikeanpuoleisessa kuvassa taas ytimet kopioivat tietoa globaalista muistista satunnaisista muistipaikoista. Valkoiset alueet kuvissa kuvaavat arvoja joita ei voida suorittaa Fermi-arkkitehtuurissa, eli säikeiden määrä lohkoa kohden on liian suuri. Oikeanpuoleisessa kuvassa myös 1024 säikeen lohko koko aiheuttaa tietovirtasuorittimen käyttöasteen tippumisen nollaan, koska satunnaishakuun liitetty algoritmi käyttää liikaa rekistereitä lohkoa kohden. Käytettäessä alle 32 säikeen lohkokokoa (vaalean harmaa alue), molemmissa tapauksissa jää käyttöaste alle 20%:n, tämä johtuu siitä, että tietovirtasuorittimelta jää vapaaksi ytimiä. Isommilla lohkokooilla käyttöaste nousee, mutta satunnaista hakua käyttävä ohjelma ei koskaan saavuta 100% käyttöastetta. Tämä johtuu siitä, että ytimet joutuvat odottamaan tietoa globaalista muistista. Käytettäessä yhdistettyjä hakuja tietovirtasuorittimet voivat saavuttaa 100% käyttöasteen, jos säielohkon koko on valittu oikein. [74]



Kuva 18. Yhdistetyn ja satunnaisen muistihaun vertailu [74].

4.2 Optimoinnin maksimaalinen nopeutus

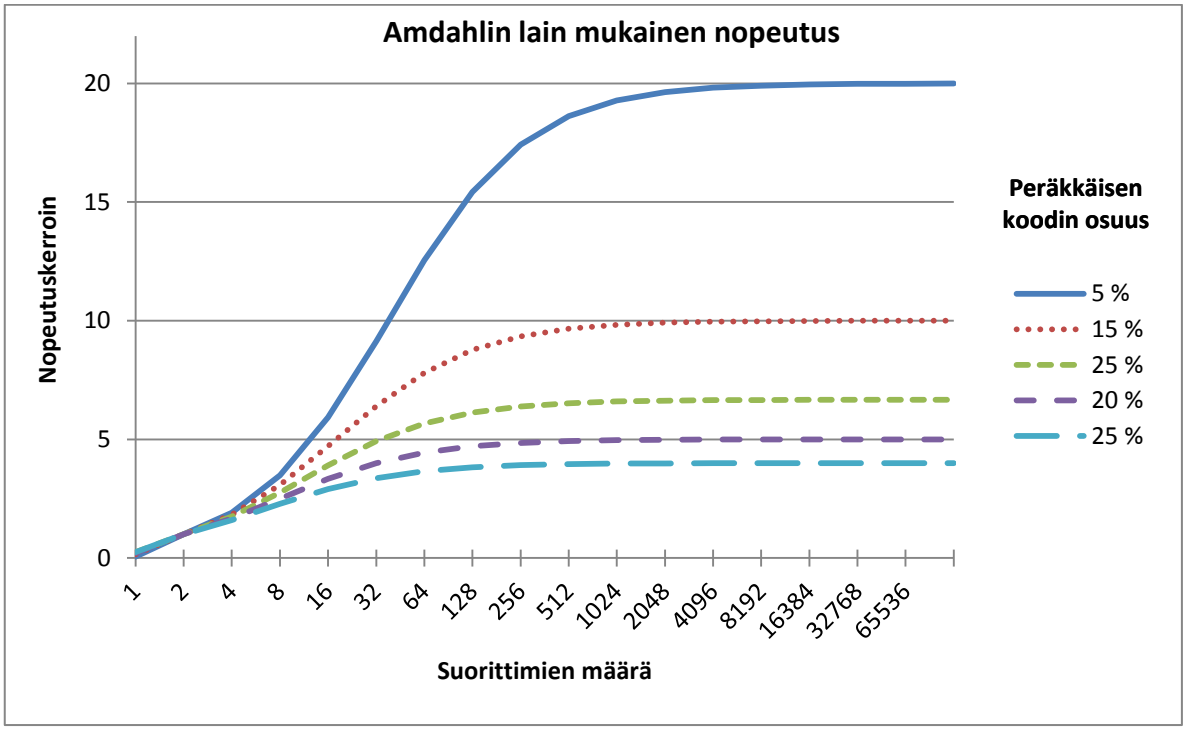
Käyttäen edellä mainittuja ominaisuuksia tehokkaasti hyväksi voidaan saavuttaa mahdollisimman suuri parannus rinnakkaisesti suoritettavan koodin suoritusaikaan. Vaikka koodi optimoitaisiin mahdollisimman hyvin niin, että suoritettavia ytimiä olisi maksimimäärä ja kaikki tietovirtasuorittimet olisivat täysin työllistettyjä, suoritusaikaa ei ole mahdollista pienentää jatkuvasti kasvattamalla käyttävien suorittimien tai säikeiden määrä.

Maksimaalisen optimoinnin muutamia perustutkimuksia ovat vuonna mm. 1967 julkaistu Amdahlin laki sekä 1988 julkaistu Gustafsonin laki. Amdahlin laki on kirjoitettu lähinnä vastustamaan rinnakkaisen laskennan kehitystä. Sen mukaan tulisi keskittyä suorittimien nopeuttamiseen. Gustafsonin laki taas puhuu rinnakkaisen laskennan puolesta.

Amdahlin lain mukaan algoritmin nopeutus, kun osa siitä muutetaan rinnakkaiseksi, voidaan laskea kaavasta [75]:

$$S = \frac{1}{(F + \frac{1-F}{N})} - O_N,$$

missä *S* kuvaa suorituskyvyn muutosta, *F* peräkkäisen koodin osuutta, *N* suorittimien määrää ja O_N rinnakkaistamisen aiheuttamaa yleiskustannusta. Amdahlin lain mukaan peräkkäisen laskennan osuuden lähestyessä nollaa nopeutuu algoritmi suoraan suhteessa käytettävien suorittimien määrään. Lain mukaan tietyn kokoisen ongelman ratkaisemista ei voida nopeuttaa rajattomasti lisäämällä suorittimia, eikä suoritusnopeus kasva samassa suhteessa lisättävien suorittimien määrään. Tämä johtuu siitä, että ongelmassa on olemassa aina kohtia joita ei voida suorittaa rinnakkaisesti. Lisäksi rinnakkaistaminen aiheuttaa yleiskustannusta joka hidastaa suoritusta. Yleiskustannus koostuu säikeiden luonnista ja tuhoamisesta, synkronoinnista sekä muistinkäsittelyn hallinnasta. Yleiskustannus ja koodin peräkkäinen osa määrittävät ohjelman minimisuoritusajan. Kuvassa 19 esitetään Amdahlin lain mukaan laskettu nopeutus muutamilla eri peräkkäisen laskennan osuuden arvoilla, esitys olettaa että rinnakkaistaminen ei aiheuta kustannuksia.

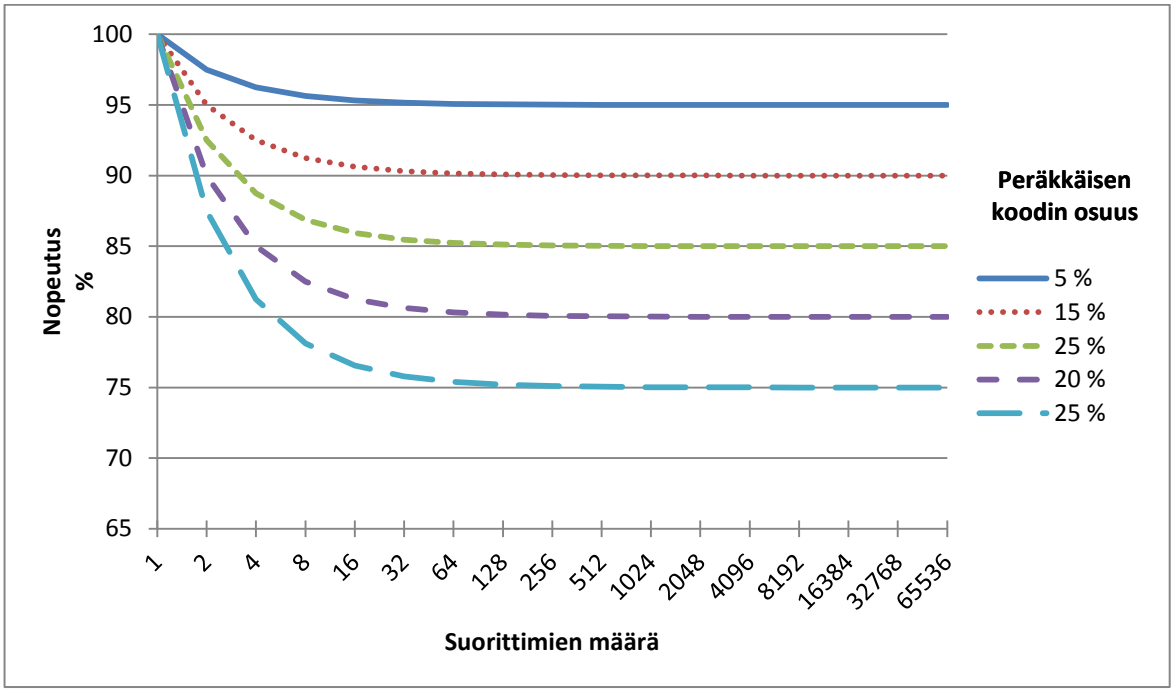


Kuva 19. Amdahlin lain mukainen nopeutus

Siinä missä Amdahlin laki olettaa ongelman koon pysyvän samana, Gustafsonin laki toteaa, että suorittimien määrän lisääntyessä rinnakkaisesti suoritettavan ohjelman koko kasvaa, mutta peräkkäisen osuuden koko pysyy vakiona. Gustafsonin lain mukaan työmäärän ja suorittimien määrän kasvaessa nopeutuu suoritus tietyllä vakiokertoimella. Gustafsonin lain mukaan nopeutus määräytyy kaavasta [75]:

$$S = \frac{S+N(1-S)}{S+(1-S)} - O_N,$$

missä N on suorittimien määrä, S on peräkkäisesti suoritettavan koodin osuuden prosentuaalinen osuus ja O_N rinnakkaistamisen aiheuttama yleiskustannus N-määrällä suorittimia. Gustafsonin lain mukaan koodi nopeutuu aina kun uusi suoritin lisätään mukaan, mutta nopeutuksen tehokkuus laskee. Kuva 20 esittää, Gustafsonin lain mukaan, suorittimien määrän vaikutuksen nopeutuksen tehokkuuteen eri peräkkäisen koodin osuuksilla. Arvoja laskettaessa oletettiin, että suorittimien lisäämisestä ei aiheudu yleiskustannusta. [75]



Kuva 20. Gustafsonin lain mukainen nopeutus.

Vaikka rinnakkaistamisen tuomaa nopeutusta voidaan teoreettisesti arvioida erilaisilla kaavoilla, aiheuttavat käytännössä toteutettavissa olevat ratkaisut rajoituksia nopeutukseen. Näistä voimakkaimmin vaikuttavat säikeiden luominen ja tuhoaminen, säikeiden synkronointi, lukkojen hallinta muistinkäytössä sekä työmäärän epätasainen jakautuvuus. Ongelmasta riippuen on olemassa jokin määrä suorittimia, jolla ohjelma saadaan ratkaistua nopeimmin. Tämän rajan jälkeen suorittimien lisääminen aiheuttaa kustannuksia edellä mainituista operaatioista siinä määrin, että kokonaissuoritusaika ei laske vaan saattaa jopa kasvaa. [75, 76]

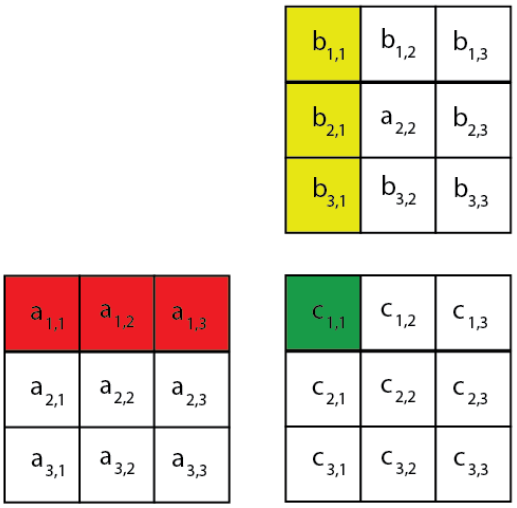
4.3 Esimerkkiongelma

Tässä luvussa esitellään esimerkkiongelma sekä pohditaan sen suorittamista CUDA-ympäristössä. Esimerkkiongelmaksi on valittu kahden neliömatriisiin matriisitulon toteuttaminen. Matriisitulo on suorituksena yksinkertainen, mutta vaatii suorittimelta paljon työtä laskemisen ja muistihakujen suhteen. Kahden matriisin kertolasku on myös hyvin sopiva CUDA-ympäristössä suoritettavaksi lähinnä siksi, että matriiseissa voidaan käyttää CUDA-ohjelmointiin sopivaa lohkojaottelua.

Matriisien *A* ja *B* matriisitulo on matriisi *C* = *AB*. Jos molemmat matriisit ovat kokoa *m x m*, on myös tulosmatriisin koko *m x m*. Matriisien *A* ja *B* matriisitulo määräytyy kaavan

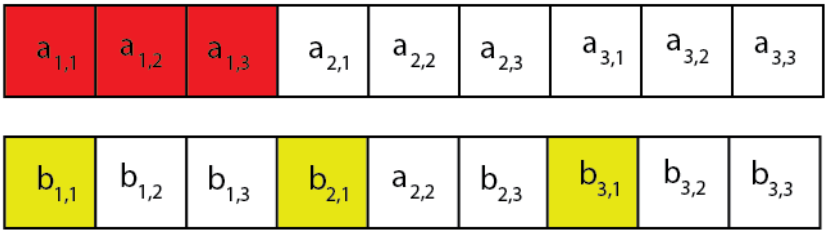
$$c_{i,j} = \sum_{k=1}^{m} a_{i,k} b_{k,j}$$

mukaan. Kaavan mukaan kohdealkioon lasketaan indeksin *i* mukaisen rivin ja indeksin *j* mukaisen sarakkeen alkioiden tulojen summa. Tämä on esitetty kuvassa 21, jossa *C* tulosmatriisin alkioon (1,1) lasketaan matriisin *A* rivin 1 sekä matriisin *B* sarakkeen 1 alkioiden tulojen summa. Tämä edellyttää *A*:n koko rivin 1 alkioiden sekä *B*:n koko sarakkeen 1 alkioiden läpikäymistä. [27]



Kuva 21. Matriisitulon laskeminen.

Jotta matriisikertolaskun suoriutumisesta tietokoneessa saadaan selvempi käsitys esittää kuva 22 matriisien *A* ja *B* alkioiden sijainnin muistissa. Käytäessä läpi matriisin *A* alkiota sijaitsevat alkiot peräkkäisissä muistiosoitteissa. *B*:n alkiot taas sijaitsevat *m* (matriisin leveys) muistiosoitteen päässä toisistaan. Alkioiden sijainnin vaikutusta muistihakuihin tarkastellaan myöhemmin.



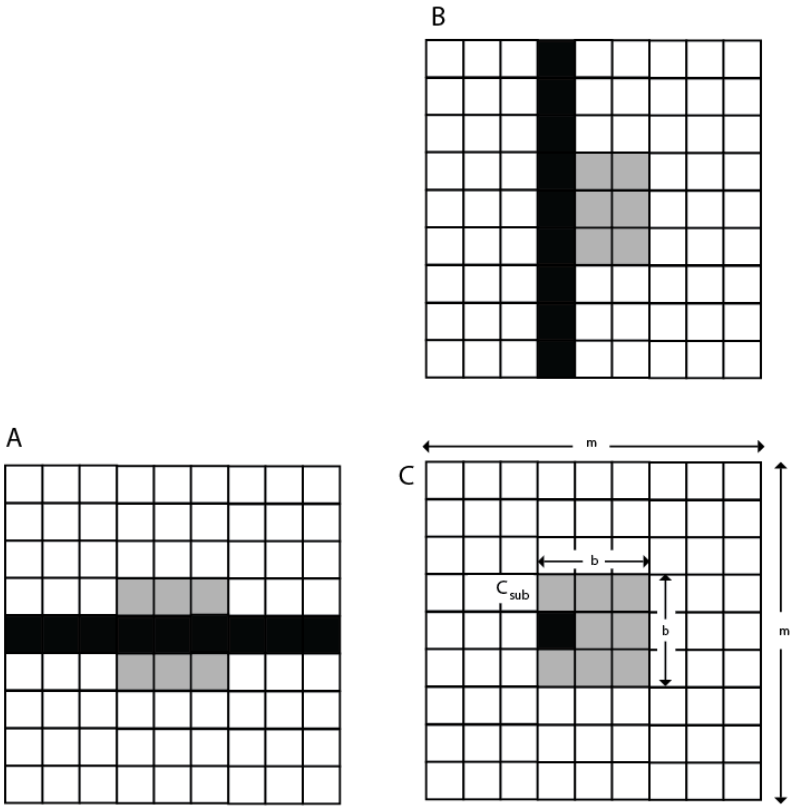
Kuva 22. Matriisien alkioiden sijainti muistissa.

Esimerkkiongelma on helppo ohjelmoida suoritettavaksi CPU:lla, jos ei oteta huomioon koodin optimoimista. Matriisien kertolaskun suoritus vaatii $\theta(m^3)$ ajan peräkkäisprosessorilla. Matriisikertolaskun optimointia on tutkittu paljon, ja sen laskentaa voidaan optimoida. Matriisien kertolaskun uskotaan yleisesti olevan mahdollista ajassa $\theta(m^2)$, tosin sellaista menetelmää ei vielä ole keksitty. Tässä työssä käsitellään peruslaskentaa sen ohjelmoitavuuden takia. [77]

Algoritmisen optimoinnin sijaan tarkastellaan, miten saman algoritmin toteuttava koodi suoriutuu niin CPU:lla kuin grafiikkasuorittimella. Grafiikkasuorittimella optimointia suoritetaan vain muistinkäsittelyn ja säikeiden suorituksen suhteen. Algoritmina käytetään edellä mainittua perusalgoritmia matriisien kertolaskun suorittamiseksi. CPU:lla yhden suorittimen ympäristössä ajettava koodi on esitetty liitteessä A. CPU-ohjelma suorittaa jokaisella alkiolle 4*m* kertolaskua, 4*m* yhteenlaskuoperaatiota sekä 3*m* muistikäsittelyoperaatiota, missä *m* on matriisin sivun pituus

CPU:lla suoritettavaa matriisien kertolasku verrataan aluksi GPU:lla suoritettavaan perusversioon kertolaskusta. Perusversio alustaa matriisit grafiikkasuorittimen ulkopuoliseen muistiin, josta ne kopioidaan grafiikkasuorittimen muistiin. Liitteessä C esitetään koodi ytimelle, joka suorittaa perusversion matriisikertolaskusta. Liitteessä B vastaavasti esitetään GPU-ohjelman pääohjelman koodi, jossa rinnakkainen laskenta alustetaan ja aloitetaan. Perusversiossa yhtä tulosmatriisin, kooltaan *m x m*, alkiota kohden suoritetaan 2*m* kappaletta globaalista muistista hakuja sekä yksi kirjoittaminen globaalimuistiin. 3*m* + 3 kappaletta kertolaskuja ja 4*m* + 3 yhteenlaskuoperaatiota. Suoritettaessa ohjelmaa jokaista tulosmatriisin alkiota kohden ajetaan yksi säie.

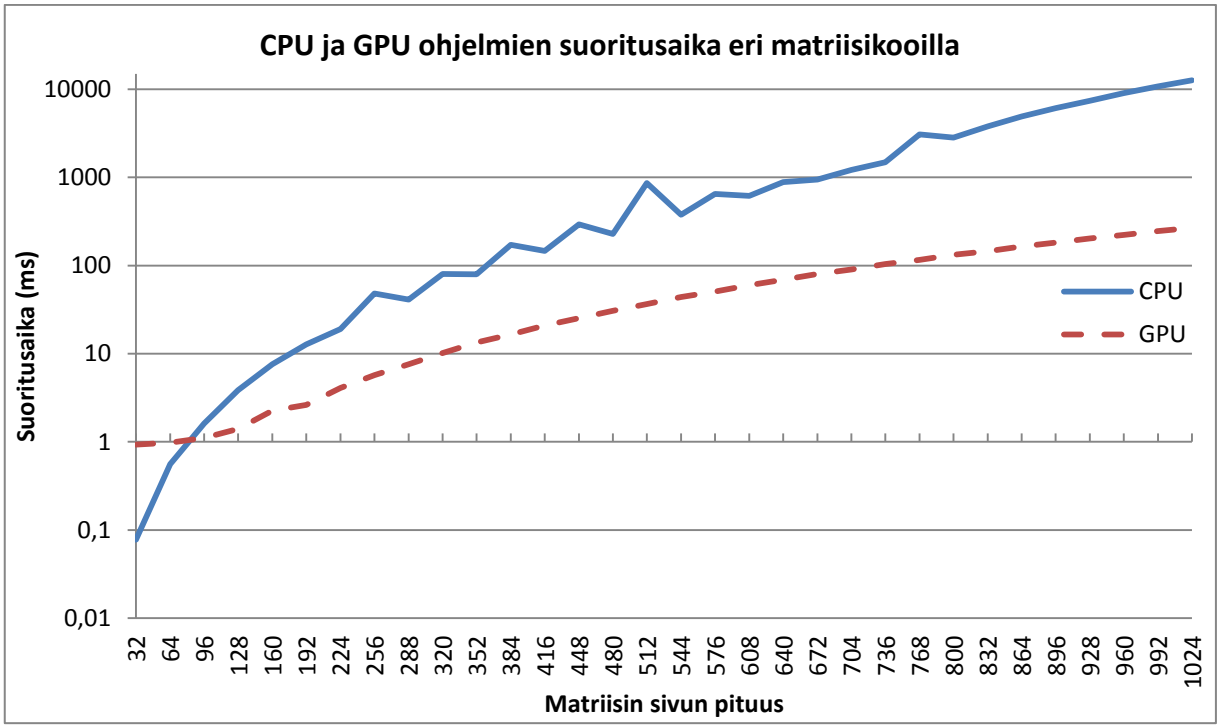
Suoritettaessa matriisien kertolaskua GPU:lla suoritusruudukon koko valittiin niin, että jokaiselle tulosmatriisin alkiolle suoritetaan yksi säie. Suoritusruudukon koon määrittely on esitetty kuvassa 23, kuvassa *m* kuvaa matriisin sivun pituuden, *b* lohkon sivun pituuden ja harmaa alue lohkon C_{sub}. Musta alue kuvaa *C*:n tulosalkiota (lohkon säie) ja sen läpikäymiä *A*:n ja *B*:n alkioita. Verrattaessa CPU ja GPU -ohjelmien eroja käytetään *b*:n arvoa yksi.



Kuva 23. Matriisien kertolaskun suorittamisen jakaminen lohkoihin.

Vertailussa käytettiin AMD:n 2,80 GHz:n suoritinta ja NVIDIAn GeForce GT640 grafiikkasuoritinta. GeForce GT640 toimii 1,05 GHz:n kellotaajuudella ja voi suorittaa ohjelmaa parhaimmillaan 384 CUDA-ytimellä. CUDA-ytimet on jaettu kahteen tietovirtasuorittimeen, yksi tietovirtasuoritin voi suorittaa yhtäaikaisesti neljää säielohkoa. Globaalia DRAM-muistia GT640 sisältää 2 Gt. Molemmilla alustoilla (CPU ja GPU) ajettiin jokaiselle matriisikoolle kymmenen suorituskertaa, joista mitattiin suoritusaika, suoritusaikojen

keskiarvoa käytettiin kaavioita muodostettaessa. Kuva 24 esittää CPU ja GPU ohjelmien suoritusajat, taulukko arvoista on esitetty liitteessä E.



Kuva 24. CPU ja GPU koodin suoritusaika eri matriisikooilla.

Kaaviosta voidaan havaita, että CPU:lla suoritusaika kasvaa voimakkaasti suhteessa matriisin kokoon nähden. CPU:n suoritusajan kuvaaja on myös voimakkaasti sahalaitainen. Tämä johtuu siitä, että suorittimella on paljon muitakin tehtäviä samanaikaisesti mikä aiheuttaa sen, että suoritusaika saattaa olla nopeampi tai hitaampi kuin oletettu suoritusaika. Kaaviota tarkastellessa voidaan todeta, että grafiikkasuoritin suoriutuu jo optimoimattakin nopeammin isoista matriiseista kuin CPU. Tarkempi kuvaus muutamista pienten matriisien suoritusajoista on esitetty taulukossa 2.

Taulukko 2. CPU:n ja GPU:n suoritusaikoja.

Matriisin koko	CPU aika (ms)	GPU aika (ms)
32	0,1	0,9
64	0,6	1,0

96	1,6	1,1
128	3,8	1,4
160	7,6	2,3

Mittausten mukaan CPU:lla suoritettava koodi suoriutuu nopeammin, kun matriisin sivun pituus on 32, 64 tai 96. Mittaustulokset vaihtelevat hieman suorituskerrasta riippuen, mutta raja jossa GPU alkaa suoriutua nopeammin kuin CPU liikkuu 64:n ja 96:n välimaastossa. Tämä johtuu viiveestä joka aiheutuu siirrettäessä tietoa CPU:n ja GPU:n muistien välillä. Vasta kun GPU:n suorittama nopeutus ylittää viiveen on eduksi käyttää GPU:ta laskentaan. Viive on riippuvainen GPU:n ominaisuuksista ja käytettävän matriisin koosta.

Ajettaessa GPU-ohjelmaa NVIDIAn NSight-profiloijan kanssa saadaan tietoa ohjelman suorituksesta CUDA-ytimissä. Profiloijan antamien tietojen mukaan suoritettava ydin käyttää kahdeksan rekisteriä säiettä kohti ja 0 kt paikallista tai jaettua muistia. Koska lohkokoko on jaollinen 32:lla tietovirtasuorittimien käyttöaste nousee likimain 100%. Kuitenkaan tietovirtasuorittimien täysi työllisyysaste se ei tarkoita sitä, ettei koodia voisi optimoida. Vaan että kaikilla tietovirtasuorittimilla riittää jatkuvasti säikeitä suoritettavaksi, vaikka osa säikeistä/lohkoista odottaakin tietoa globaalimuistista. Jos käytettäisiin esim. 18x18 säiettä lohkoa kohden tippuu tietovirtasuorittimien käyttöaste alle 90%, koska tällöin säikeiden määrä ei ole jaollinen tietovirtasuorittimella yhtäaikaisesti suoritettavien säikeiden maksimäärän kanssa. Tässä tapauksessa tietovirtasuoritin ei voi ottaa maksimikokoista säienippua suoritukseen ja optimoida sen suoritusta, jolloin osa tietovirtasuorittimen ytimistä on osan aikaa toimettomia.

Globaalien muistihakujen aiheuttamista viiveistä voidaan päästä eroon käyttämällä jaettua muistia. Jaettua muistia käytetään jakamalla matriisit lohkoihin, jokainen lohko vastaa CUDA:n säielohkoa. Laskettaessa tulosmatriisia C, vastaa jokainen säielohko lohkon C_{sub} alkioiden laskemisesta. Säielohkon sisällä jokainen säie laskee yhden C_{sub} matriisin alkion tuloksen. Säielohkon kokoa voidaan varioida, mutta sen tulee olla jaollinen matriisin *C* koon kanssa. Säielohkon säikeet hakevat globaalimuistista säielohkon tarvitseman alimatriisin tiedot jaettuun muistiin. Kun kaikki säielohkon säikeet ovat suorittaneet omat hakunsa, ovat matriisien *A* ja *B* alimatriisien arvot käytettävissä nopeasti säielohkon jaetusta

muistista. Haun jälkeen säielohkon säikeet suorittavat oman alkionsa indeksin mukaisen matriisitulon laskennan alimatriisille C_{sub}. Alkion tulos tallennetaan muuttujaan kasautuvasti. Kuin kaikki alimatriisit on käyty läpi, voidaan alkion tulos tallentaa globaaliin muistiin. Toiminta on muuten kuten ensimmäisessä esimerkissä, kuva 23, mutta tällä kertaa harmaan alueen arvot haetaan jaettuun muistiin aluksi ja sen jälkeen suoritetaan kertolaskenta.

Etuna ensimmäiseen esimerkkiin on globaalimuistin käytön väheneminen, ensimmäinen esimerkki tekee globaaliin muistiin *2m* hakupyyntöä, missä *m* on matriisin sivun pituus. Lohkojaoteltu esimerkki sen sijaan hakee globaalista muistista alkion arvon *m*/*b*, kertaa, missä *m* on matriisin sivun pituus ja *b* on lohkon sivun pituus. Ytimien, joka käyttää jaettua muistia matriisikertolaskuun, koodi on esitetty liitteessä D. Lisäksi liite B esittää pääohjelman koodin, pääohjelmassa alustetaan ja aloitetaan rinnakkainen laskenta.

GPU:lla suoritettavaa laskentaa voidaan nopeuttaa myös käyttämällä jaetun muistin sijaan vakiomuistia. Kuitenkin esimerkkiohjelman tapauksessa vakiomuistin käyttäminen ei ole mahdollista, koska vakiomuistin koko on vain 64kt. Jos vakiomuistin koko olisi isompi, voitaisiin sinne kopioida lähdematriisit, jolloin olisi mahdollista nopeuttaa ohjelman suoritusta lisää. Verrattuna globaaliin muistiin vakiomuistia oikein käyttävä ohjelma on aina vähintään yhtä nopeasti suoriutuva. [67]

4.4 Kustannusparametrien liittäminen GPU-laskentaan

Matriisien kertolaskusta voidaan tunnistaa muutamia parametreja, joiden avulla grafiikkasuorittimella suoritettavien algoritmien tehokuutta voidaan vertailla. Parametreilla voidaan arvioida säikeiden suorittamien operaatioiden vaikutusta suoritusaikaan. Kuten missä tahansa CUDA-ohjelmassa myös esimerkkiohjelmissa käytetään seuraavia CUDA-operaatioita: globaalimuistin käsittely, jaetun muistin käsittely, säikeiden synkronointi sekä paikalliset operaatiot. Globaalin muistin operaatioihin kuuluvat haut grafiikkasuorittimen muistista ja tallennukset globaaliin muistiin. Jaetun muistin operaatioihin kuuluvat vastaavasti jaetun muistin haku- ja tallennusoperaatiot. Paikallisia operaatioita ovat laskentaope-

raatiot. Näitä operaatioita voidaan käyttää hyväksi määriteltäessä kustannusparametreja, jotka vaikuttavat ohjelman suoritusaikaan.

Liitettäessä kustannusparametreja ongelmaan joudutaan pohtimaan mitä mallia käytetään. Luvussa 2 esitettiin muutamia yleisiä rinnakkaisen laskennan malleja. Esimerkkiongelmaa suoritetaan tässä työssä jaetun muistin ympäristössä, joten myös mallin tulisi olla sopiva tähän ympäristöön. Jaetun muistin käsitellyistä malleista, P-PRAM ja QSM, molemmat voisivat sopia käytettäväksi malliksi. QSM-malli kuitenkin laskee kustannusta muodostamalla jonon muistialuetta kohden. Varsinkin pelkkää globaalia muistia käyttävässä ohjelmassa on viitteitä QSM-mallin suuntaan. Koska ei ole kuitenkaan tarkkaa tietoa siitä, miten CUDA-ympäristössä muistipyynnöt tapahtuvat eikä mallilla voida kuvata toimintaa yhdistettyjen muistihakujen toimintaa, valitaan käytettäväksi muokattu P-PRAM malli. P-PRAM malli on vanha malli jolla pyritään mallintamaan teoreettisen PRAM-laitteiston suorituskykyä, kuitenkin siinä on yhtymäkohtia CUDA-ympäristöön. P-PRAM mallin parametrit vastaavat osittain CUDA:n parametreja, mutta osittain mallia joudutaan muokkaamaan.

Muokatussa P-PRAM mallissa ei oteta huomioon paikallisen työn osuutta, koska oletetaan sen osuuden olevan pieni verrattuna muistihakujen aiheuttamiin viiveisiin. Normaalin P-PRAM-mallin kustannusparametrit ovat: paikallinen operaatio, EREW-muistikäsittely, CRCW-muistikäsittely, multiprefix-operaatio ja globaalisynkronointi. Koska kumpikaan esimerkkiohjelma ei tarvitse eksplisiittistä muistin käyttöoikeutta ja paikallinen operaatio sekä multiprefix-operaatio tiputetaan pois muokatusta mallista. Näin ollen ovat muokatun mallin parametrit: CRCW-muistikäsittelyn kustannusarvio ja lohkosynkronoinnin kustannusarvio. Lisäksi muokattuun malliin otetaan mukaan tieto siitä, kuinka monta säiettä ja lohkoa ajetaan sekä tietovirtasuorittimien määrä.

4.5 Mallin liittäminen ongelmaan

Liittämällä malli ongelmaan voidaan teoreettisesti arvioida miten GPU-laskenta käyttäytyy ja kuinka paljon siitä voidaan saada hyötyä. Mallin avulla voidaan myös arvioida eri lait-

teistoparametrien ja ongelman koon vaikutusta siihen, millaista algoritmia kannattaa käyttää.

Ensimmäinen esimerkkiohjelma käyttää raskaasti globaalimuistia hyväkseen, koska yksi säie tekee globaaliin muistiin pyyntöjä, *m x m* kokoisessa matriisissa, $2m+1$ kappaletta. Tästä seuraa, kun jokaiselle alkiolle luodaan oma säie, että koko ohjelma tekee muistipyyntöjä *m x m x* $(2m+1)$ kappaletta, tällöin suoritusaika on luokkaa $\theta(m^3)$. Jaettua muistia käyttävä ohjelma sen sijaan tekee enemmän laskentatyötä, mutta käyttää globaalia muistihakua vähemmän. Jaetun muistin ohjelmassa säie ajaa silmukkaa, joka hakee lähdematriiseista arvot jaettuun muistiin. Silmukassa tapahtuu (*m/b*) *x* 2 globaalimuistihakua, missä *m* on matriisin sivun pituus ja *b* lohkon sivun pituus. Lisäksi silmukka tallentaa haettujen tietojen arvot jaettuun muistiin. Säikeen sisällä tehdään myös alimatriisin kumulatiivinen kertolasku, mikä aiheuttaa operandien haun jaetusta muistista. Lisäksi säie tekee (*m/b*) *x* 2 synkronointia sekä yhden tallennuksen globaaliin muistiin.

Käyttäessä P-PRAM-mallia ohjelmien suoritusaikojen laskemiseen päästään seuraaviin kaavoihin, joiden avulla voidaan vertailla ohjelmien teoreettisia eroja suoritusajoissa. Laskettaessa kustannusarviota oletetaan, että kaikki tietovirtasuorittimet ovat täysin työllistettyjä ja, että lohko voidaan suorittaa yhdellä tietovirtasuorittimella. Pelkästään globaalimuistia käyttävän säikeen suoritusaika voidaan laskea kaavalla

$$t_{säie} = (2*\mathrm{m}*\beta + \beta), \tag{1}$$

missä *m* on matriisin sivun pituus ja β on globaalin muistihaun kustannus. Kaavassa (1) lasketaan suoritusaika, kun säie hakee matriisista *A* kaikki oman indeksinsä mukaisen rivin arvot ja matriisista *B* kaikki oman indeksinsä mukaisen sarakkeen alkioiden arvot. Tästä syntyy suoritusaika 2*m*β. Lopuksi säie tallentaa alkioiden kertolaskujen summan tulosmatriisiin suoritusajalla β.

Vastaavasti jaettua muistia käyttävän säikeen suoritusaika lasketaan kaavalla

$$t_{säie} = \frac{m}{b} * (2\beta + 2\varepsilon + 2b\varepsilon + 2b^2\delta) + \beta, \tag{2}$$

u v k l

missä *m* on matriisin sivun pituus, *b* lohkon sivun pituus, β globaalin muistihaun kustannus, ε jaetun muistihaun kustannus ja δ lohkosynkronoinnin kustannus. Jaetun muistin käyttöä on havainnollistettu kuvassa 23. Jaettua muistia käyttävä ohjelma, kaava (2), suorittaa globaalista muistista hakuja (u) kaksi kappaletta hakemalla oman indeksinsä osoittamat alkioiden arvot matriiseista *A* ja *B*. Säie tallentaa arvot lohkon jaettuun muistiin (v) sekä laskee jaetussa muistissa sijaitsevien matriisien alkioista kumulatiivista kertolaskua (k). Säie suorittaa myös kaksi lohkosynkronointia (l), lohkosynkronoinnin suoritusajan oletetaan kasvavan lohkokoon mukaan. Jokainen säie suorittaa edellä mainitut toimenpiteet silmukassa, mistä seuraa kerroin $\frac{m}{b}$. Lopuksi säie tallentaa tuloksen globaaliin muistiin.

Molemmissa tapauksissa koko ohjelman suoritusaika voidaan laskea kaavalla

$$t_{ohjelma} = \frac{b^2 * \max(1, \frac{\max(1, \frac{b^2}{w})}{s})}{k} * t_{säie}, \quad (3)$$

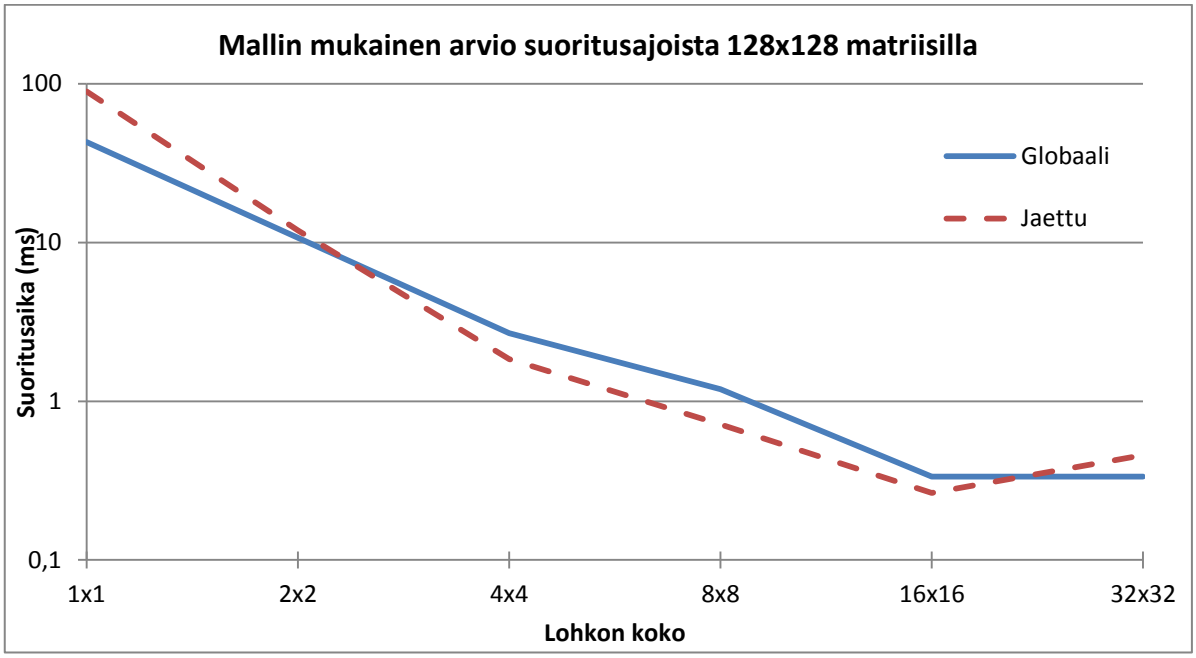
missä *m* on matriisin sivun pituus, *b* lohkon sivun pituus, *w* säienipun koko, *s* yhtäaikaisesti suoritettavien säienippujen määrä ja *k* on tietovirtasuorittimien määrä. Koko ohjelman suoritusaikaa laskettaessa pitää aluksi selvittää kuinka monta säienippua pitää lohkoa kohden suorittaa ja kuinka monta säienippua voidaan suorittaa tietovirtasuoritinta kohden. Kertomalla saatu arvo yhden säikeen suoritusajalla saadaan arvio koko ohjelman suoritusajasta.

Edellä mainituilla kaavoilla voidaan arvioida grafiikkasuorittimella suoritettavan ohjelman suoritusaikaa erilaisissa laitteistoissa. Suoritusaikaa voidaan arvioida varioimalla globaalin muistin ja jaetun muistin nopeusarvoja, käytettävää lohkokokoa sekä käytettävissä olevien tietovirtasuorittimien määrä. Seuraavassa luvussa esitetään muutamia arvioita mahdollisesta suoritusajasta sekä verrataan niitä käytännössä mitattuihin arvoihin.

5 Tulokset

Tässä osiossa esitetään tuloksia käytettäessä teoreettista mallia sekä oikeassa suoritusympäristössä ajettuja ohjelmia. Teoreettisella mallilla arvioidaan 128x128 kokoisten matriisien kertolaskun suoritusaikaa, jaetun muistin ja globaalin muistin ohjelmilla. Oikeassa suoritusympäristössä matriisien kertolaskut suoritetaan 128x128, 256x256, 512x512 ja 1024x1204 kokoisille matriiseille. Pienemmällä matriisikoolla pyritään vertailemaan teorian ja käytännön eroa, isommilla taas pyritään samaan esimerkkiohjelmien mahdollinen suorituskykyero paremmin esille. Käytännössä mitattuja tuloksia muodostettaessa ohjelmia on ajettu kymmen kertaa GT640 grafiikkasuorittimella, tulokseksi on valittu mitattujen suoritusaikojen keskiarvo.

Käyttämällä edellisessä luvussa kuvattuja malleja (kaavat 1,2,3) saadaan arvioitua lohkokoon vaikutusta laskennan suorittamiseen. Teoreettisessa arvioinnissa matriisitulon suoritusaika lasketaan 1x1 … 32x32 säiettä sisältävissä lohkoissa, jolloin matriisia kohden suoritetaan vastaavasti 128x128 … 4x4 lohkoa. Kuva 25 esittää teoreettisen arvio suoritusajasta eri lohkokooilla. Y-akseli kuvaa arvioitua suoritusaikaa ja vastaavasti X-akseli kuvaa käytettyjen säikeiden määrää lohkoa kohden. Sinisen käyrän (globaalimuisti) arvot on laskettu käyttämällä kaavoja 1 ja 3 ja punaisen käyrän (jaettu muisti) käyttäen kaavoja 2 ja 3. Parametreina kaavoissa on käytetty $\beta = 0{,}04$ µs, $\varepsilon = 0{,}004$ µs, $\delta = 0{,}005$ µs, $w = 32$, $s = 4$ ja $k = 2$. Arvot perustuvat NVIDIAn GeForce GT640 grafiikkasuorittimen ominaisuuksiin sekä arvioihin operaatioiden kustannuksista. GeForce GT640 toimii 1,05 GHz:n kellotaajuudella ja voi suorittaa ohjelmaa parhaimmillaan 384 CUDA-ytimellä. CUDA-ytimet on jaettu kahteen tietovirtasuorittimeen, yksi tietovirtasuoritin voi suorittaa yhtäaikaisesti neljää säielohkoa. Globaalia DRAM-muistia GT640 sisältää 2 Gt. [78, 79]

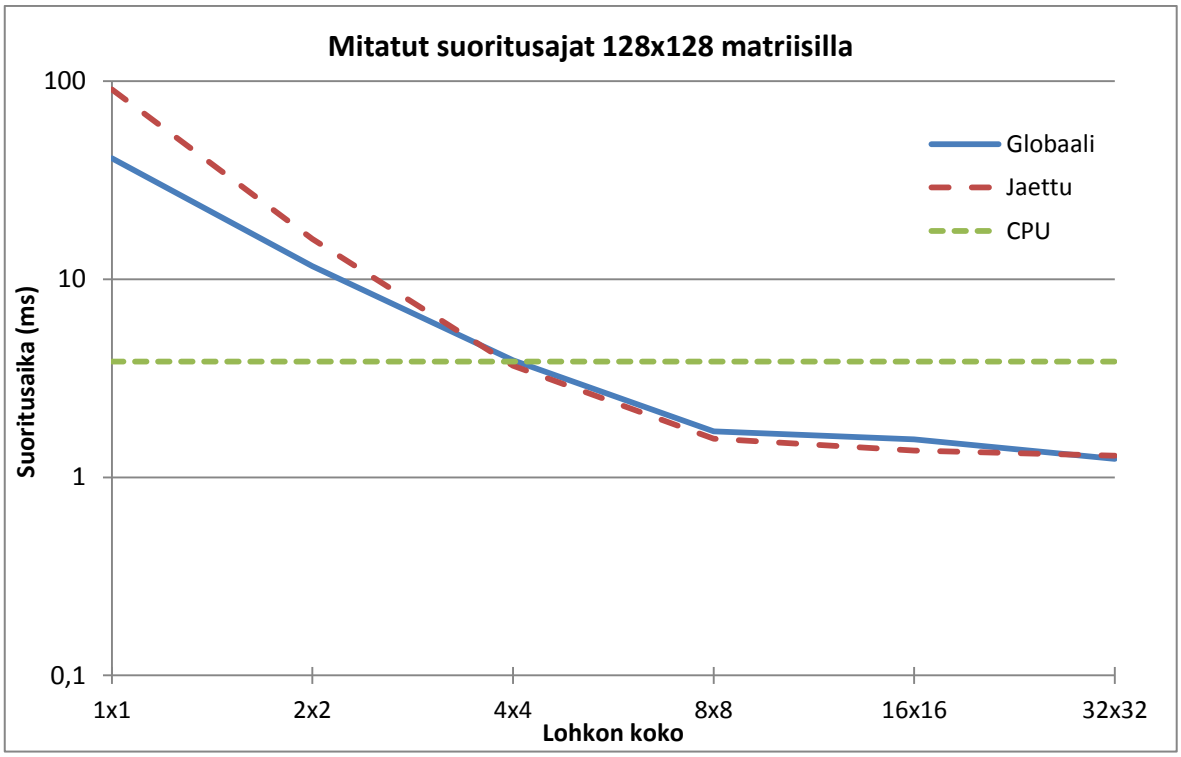


Kuva 25. Arvio teoreettisista suoritusajoista 128x128 matriisilla.

Teoreettisen arvion mukaan molemmat ohjelmat suoriutuvat nopeammin lohkokoon kasvaessa. Nopeutuminen kuitenkin tasaantuu isommilla lohkokooilla ja jaetun muistin ohjelma vaikuttaa jopa hidastuvan. Hidastuminen johtuu lohkosynkronoinnin kustannusparametrin arvon kasvamisesta. Säikeiden määrän kasvattaminen käyttäytyy vastaavasti kuin Gustafsonin laissa on määritelty, eli lohkokoon kasvattaminen ei lisää vastaavassa suhteessa tehokkuutta. Globaalia muistia käyttävä ohjelma suoriutuisi teoreettisen arvion mukaan paremmin kuin jaetun muistin ohjelma pienillä lohkokooilla sekä suurimmalla lohkokoolla. Pienellä lohkokoolla tämä on oletettavaa koska tällöin jaetusta muistista ei ole hyötyä, vaan sen käyttö aiheuttaa lisätyötä säikeille. Molemmat ohjelmat vaikuttavat saavuttavan tehokkaimman hyödyn, lohkokoon kasvatuksesta, 16x16 lohkokokoa käytettäessä. Tämä arvio perustuu siihen, että pienemmillä lohkokooilla tietovirtasuorittimien työllisyysaste ei nouse tarpeeksi suureksi ja isoilla vastaavasti osa säielohkoista joutuu odottamaan joko vapaata tietovirtasuoritinta tai tietoa globaalista muistista.

Teoreettisia tuloksia tarkasteltaessa tulee ottaa huomioon, että teoria ei ota kantaa siihen, miten tietovirtasuoritin optimoi säikeiden suorittamista. Teoriassa ei voida myöskään ennustaa tarkasti globaalien muistihakujen käyttäytymistä, muistihakujen yhdistämisen ja välimuistin käytön suhteen.

Teoreettisia arvioita vastaavat käytännön testit ajettiin todellisessa suoritusympäristössä GT640 grafiikkasuorittimella. Ajettavat ohjelmat on laadittu Microsoft Visual Studio –ohjelmankehitysohjelmistossa johon on asennettu NVIDIAn CUDA-kehitysympäristö. Testiajojen tulokset on esitetty liitteessä F. Kuva 26 esittää globaalin muistin ohjelman mitatut suoritusajat sekä jaetun muistin mitatut suoritusajat 128x128 matriiseilla. Y-akseli kuvaa suoritusaikaa ja vastaavasti X-akseli kuvaa käytettyjen säikeiden määrää lohkoa kohden. Sinisen käyrän (globaalimuisti) arvot on ajettu ohjelmalla, joka on esitetty liitteessä C ja punaisen käyrän ohjelmalla, joka on esitetty liitteessä D. Lohkon säikeiden määrä suorituksissa on sama kuin teoreettisessa mallinnuksessa 1x1 … 32x32. Suoritus on tehty lohkokooilla, jotka ovat jaollisia matriisin koon kanssa. Jos lohko koko ei olisi jaollinen matriisin koon kanssa, tulisi se ottaa huomioon koodia laadittaessa. Tällöin koodiin tulisi `if-else` rakenteita, jotka tarkastaisivat ovatko alkiot matriisin sisäpuolella, `if-else` rakenne kuitenkin hidastaisi koodin suoriutumista.



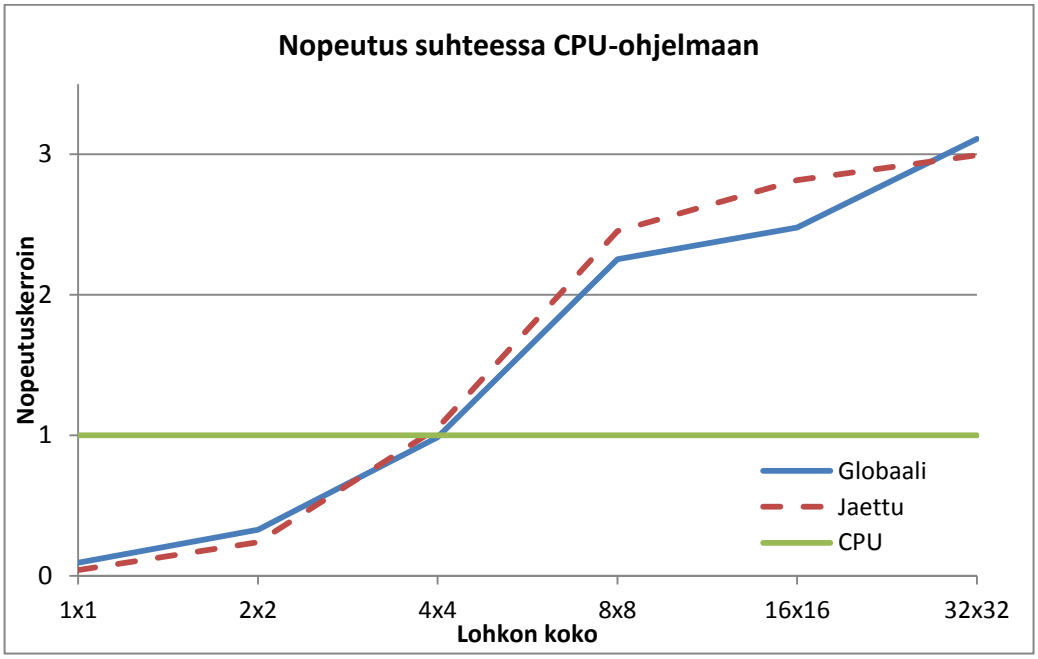
Kuva 26. Mitatut suoritusajat esimerkkiohjelmien suorituksista.

Kuvasta voidaan havaita, että mitatut suoritusajat ovat linjassa teoreettisiin arvioihin, mutta suoritusajat ovat hieman hitaampia kuin arvioidut. Tämä johtuu parametrien arvoista, jotka

perustuivat arvioihin. Kuten teoreettinen arvio ennusti, globaalinmuistin ohjelma toimii alkuun nopeammin kuin jaetun muistin ohjelma. Tämä johtuu edellä mainitusta ylimääräisestä työstä jota jaetun muistin käyttö aiheuttaa. Jaetun muistin ohjelma toimi käytännössä myös nopeammin vasta isommalla lohkokoolla mitä teoriassa ennustettiin, eikä nopeusero ole niin suuri kuin teoriassa ennustettiin. Mitatuissa arvoissa ohjelmat eivät kuitenkaan hidastu, vaikka teoria niin arvioikin. Sen sijaan suoritusnopeus saavuttaa tehokkuuden lisäyksen suhteen ylärajan lohkokoolla 8x8. Minkä jälkeen lohkokoon kasvattamisella on vain lievä nopeuttava vaikutus. Kuvassa on esitetty myös CPU-ohjelman suoritusaika vihreällä viivalla. Käytettäessä ”huonoja” lohkokokoja 1x1, 2x2 ja 4x4 grafiikkasuorittimesta ei saada hyötyä suoritusnopeuteen. Valitulla matriisikoolla globaalin muistin ohjelman ja jaetun muistin ohjelman välinen suorituskykyero on pieni, vaikka edellä oletettiinkin jaetun muistin käytön mahdollistavan nopeamman suorituksen. Tähän on syynä matriisin suhteellisen pieni koko, jolloin jaetusta muistista ei ehditä saada täyttä hyötyä, jäljempänä tarkastellaan eroa isommilla matriisikooilla.

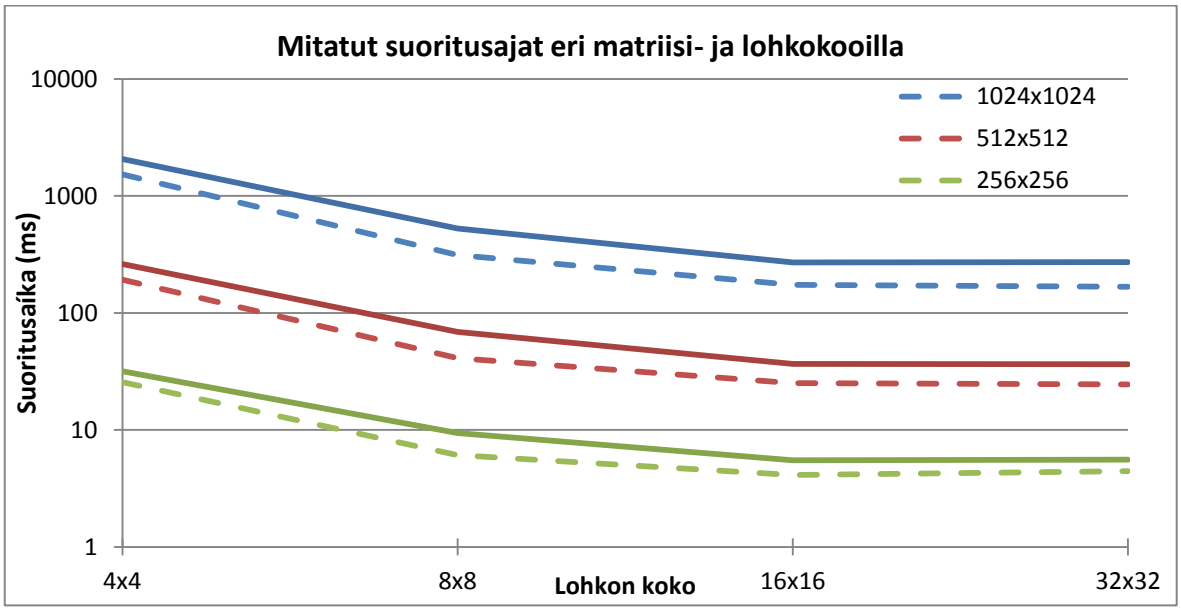
Teoreettisia ja mitattuja arvoja voidaan selittää myös tietovirtasuorittimien käyttöasteella, käytettäessä lohkokokoja 1x1, 2x2 ja 4x4 käyttöaste on vain 25%. 8x8 lohkokoko saavuttaa 50% käyttöasteen ja 16x16 sekä 32x32 nostavat käyttöasteen 100%. Täydellä 100% käyttöasteella saavutetaan tilanne, jossa säikeet suorittavat jatkuvasti ongelman ratkaisua edistävää työtä. Kuitenkin vaikka käyttöaste olisi 100%, voi suorituskyky laskea kuten edellä suoritetuista mittauksista havaittiin.

Kuva 27 esittää saavutetun nopeutuksen käyttäessä eri lohkokokoja, verrattuna CPU-ohjelmaan. Parhaimmillaan jaetun muistin ohjelma nopeuttaa laskentaa noin kolminkertaisesti ja globaalin muistin ohjelma yli kolminkertaisesti. Kuten teoreettisessa arvioinnissa todettiin ohjelmien nopeutuminen vaikuttaa seuraavan Gustafsonin lain mukaista kaavaa, koska grafiikkasuorittimella ei voida suorittaa yli 32x32 kokoisia lohkoja, ei voida arvioida miten nopeutuminen tapahtuisi isommilla lohkokooilla.



Kuva 27. Esimerkkiohjelmien saavuttama nopeutus 128x128 matriisilla.

Koska pienellä matriisilla jaetun muistin ja globaalin muistin ohjelmien suorituskykyero jäi pieneksi, ajettiin samat testit vielä 256x256, 512x512 ja 1024x1024 matriiseille. Tällä kertaa käytettiin, tietovirtasuorittimen rajoituksien vuoksi, lohkokokoja 4x4, 8x8, 16x16 ja 32x32.



Kuva 28. Mitatut suoritusajat eri matriisi- ja lohkokooilla.

Kuva 28 esittää esimerkkiohjelmien suoritusaikoja eri matriisikooilla, kuvassa Y-akseli kuvaa suoritusaikaa millisekunneissa ja vastaavasti X-akseli kuvaa käytettyjen säikeiden määrää lohkoa kohden. Jaetun muistin ohjelman suoritusaika on esitetty katkoviivalla. Suoritusaikojen nopeutuminen seuraa samaa kaavaa kuin 128x128 matriisilla. Maksiminopeutus saavutetaan 16x16 lohkokoolla. Lohkokoko 32x32 nopeuttaa jaetun muistin ohjelmaa lievästi ja globaalin muistin ohjelman suoritus taas saattaa hidastuakin, riippuen matriisien koosta. Nyt jaetun muistin ohjelma toimii 1,2 … 1,6 kertaa nopeammin kuin globaalin muistin ohjelma. Tämä johtuu siitä, että globaalia muistia käytetään isolla matriisikoolla enemmän ja jaetun muistin käytön hyöty korostuu. Trendi on myös kasvava jaetun muistin hyväksi, joten voidaan olettaa että vielä suuremmilla matriiseilla nopeusero jaetun muistin hyväksi kasvaa. Mittausten mukaan jaetun muistin ohjelma suoriutuu 75 kertaa nopeammin 32x32 lohkokoolla, käytettäessä 1024x1024 matriisia, kuin vastaava matriisien kertolasku suoritettuna CPU:lla.

Vertaamalla teoreettisen ja oikean suoritusympäristön tuloksia keskenään voidaan todeta, että teoreettinen malli lohkokoon vaikutuksesta suoritusnopeuteen antaa hyvän arvion. Kuitenkin teoreettista arviota muodostettaessa parametrien arvot tulee selvittää tarkasti, jotta malli vastaa todellisuutta. Todellisuudessa nopeutukseen vaikuttavat myös muut grafiikkasuorittimen ominaisuudet joita ei yksinkertaisella mallilla voida esittää. Näitä asioita ovat mm. tietovirtasuorittimen ylikuormittuminen tai mahdollisesti vapaaksi jäävät ytimet.

6 Yhteenveto

Tässä työssä käsiteltiin rinnakkaisen laskennan malleja, grafiikkasuorittimella tehtävää laskentaa sekä grafiikkasuorittimella suoritettavan ohjelman optimointia. Rinnakkaisen laskennan mallit ovat laaja tutkimusalue, joten alueesta käsiteltiin vain niitä osioita, jotka koettiin tärkeiksi tutkimusalueen yleisen kehityksen tai tämän työn kannalta. Malleista itsestään voisi laatia useita gradu-töitä niiden syvällisyyden vuoksi, joten tässä työssä tyydyttiin esittämään malleista perusteet. Rinnakkaisen laskennan malleista tarkasteltiin kahden eri suuntauksen, viestinvälitysmallien ja yhteisen muistin mallien malleja. Molemmat mallit ovat suunnattuja eri laitteistoarkkitehtuureihin ja molemmista malleista löytyy omia hyviä ja huonoja puolia. Rinnakkaisen laskennan mallin valitseminen riippuu suuresti käyttötarkoituksesta, suoritusympäristöstä ja tavoitteesta mihin mallilla pyritään. Mikään malli ei suoraan ole oikea, vaan mallin valinta tulee tehdä tapauskohtaisesti.

GPU-laskenta on voimakkaasti kehittyvä alue, jossa valmistajat pyrkivät tarjoamaan laskentatehoa niin peliteollisuuden tarpeisiin kuin tutkimuskäyttöönkin. Nopea kehitystyö tekee GPU-laskennan teknisestä esittämisestä haastavaa mm. tietovirtasuorittimen rakenne saattaa vaihdella eri grafiikkasuorittimien välillä paljonkin. Kuitenkin osa perusteista, kuten grafiikkasuorittimen yleinen rakenne, kehittyy pienissä askeleissa. Tällä hetkellä vallalla on laskentaan keskittyvien suoritinytimien määrän kasvattaminen, niiden nopeuden sijaan. Tähän asettaa tulevaisuudessa haasteita grafiikkasuorittimien fyysinen koko, koska yhdelle tietovirtasuorittimelle voidaan sijoittaa vain tietty määrä ytimiä. Mielenkiintoa GPU-laskentaa kohden lisää se, että grafiikkasuorittimet tekevät tuloaan myös mobiililaitteisiin. Mobiililaitepuolella haasteena ovat mm. grafiikkasuorittimen virrankulutus sekä sen fyysinen koko. Mobiililaitepuolen grafiikkasuorittimet kuitenkin tulevat tulevaisuudessa avaamaan uusia mahdollisuuksia mobiililaitekannan hyödyntämisessä rinnakkaislaskentaan. Grafiikkasuorittimen teknistä esittämistä jouduttiin tässä työssä osittain käsittelemään pintapuolisesti, sillä laitteiston toteutus kuuluu liikesalaisuuksien piiriin. Tällä hetkellä NVIDIAn laitteistototeutukset Fermi ja Kepler sekä niihin liitettävä CUDA-ohjelmointiympäristö ovat laajalti käytettyjä. Tulevaisuudessa myös muut, erityisesti korkeamman tason ohjelmointikielet saattavat kuitenkin horjuttaa CUDA:n asemaa. Tämä tapahtuu kuitenkin grafiikkasuoritinvalmistajien ehdoilla, koska he ovat vastuussa grafiik-

kasuorittimien rautatason rajapintojen hallinnasta. Grafiikkasuorittimen ohjelmointiominaisuudet ovat monimuotoiset, kuitenkin suurimmat suuntalinjat liittyvät erilaisten muistien käsittelyyn, tiedonsiirtoon muistien välillä ja säiejoukon hallintaan. Suuresti vaikuttaa myös itse suoritettavan ohjelman rakenne, ohjelman tulisi pyrkiä saavuttamaan tilanne, jossa grafiikkasuorittimien ytimet suorittavat ongelman ratkaisua edistävää työtä jatkuvasti.

GPU-laskennan optimoinnissa voidaan parhaillaan saavuttaa merkittäviä nopeutuksia. Tässä työssä saavutettiin 384:llä ytimellä parhaimmillaan 75-kertainen nopeutus verrattuna peräkkäisohjelman suoritusaikaan. GPU-laskennan käyttämiseen ei tule kuitenkaan rynnätä suoraan. Jotta GPU-laskennasta voidaan saada täysi hyöty, on ongelman ratkaisun sopivuus grafiikkasuorittimella suoritettavaksi arvioitava tarkasti. Ensimmäinen arvio ohjelman suoriutumisesta grafiikkasuorittimella voidaan tehdä teoreettisella mallilla. Teoreettisella mallilla voidaan arvioida suunnitellun rinnakkaisen algoritmin suoritusaikaa kohdenäytönohjaimella. Jos teoreettinen malli esittää ohjelman nopeutuvan GPU-laskentaa avuksi käyttämällä, kannattaa aluksi toteuttaa ohjelmasta rinnakkaisen laskennan versio pienellä työmäärällä. Usein verrattain pienellä työmäärällä pystytään testaamaan GPU-laskennan tehokkuutta. Jossain tapauksissa jo algoritmin rinnakkaistaminen riittää, koska tällöin saadaan hyöty sadoista GPU:n suoritinytimistä. Jos taas algoritmi on monimutkainen, voidaan se joutua kirjoittamaan kokonaan uudestaan rinnakkaista laskentaa varten. Tällaisessa tapauksessa tulee harkita onko työmäärä suhteessa saavutettavaan nopeutukseen. Kun ensimmäinen rinnakkaisen laskennan versio ohjelmasta on tehty, voidaan verrata rinnakkaisen laskennan ohjelman suorituskykyä peräkkäissuorittimella suoritettavaan ohjelmaan. Tämän jälkeen voidaan suorittaa GPU-laskennan optimointia jos rinnakkaisen laskennan ohjelma vaikuttaa lupaavalta ja lisää suorituskykyä tarvitaan. Tässä työssä saavutettiin parhaimmillaan 1,7-kertainen nopeutus optimoidun ja optimoimattoman ohjelman välillä, kuitenkin suorittamalla vielä laajempaa optimointia voitaisiin mahdollisesti saavuttaa suurempi nopeutus. Kuitenkin jokapäiväisessä ongelman ratkaisussa jo perusversio tarjoaa riittävää nopeutusta.

Arvioitaessa teoreettisesti saavutettavaa parannusta käytettiin muokattua rinnakkaisen laskennan mallia, joka johdettiin P-PRAM mallista. Teoreettisen mallin käyttäytyminen oli

rinnastettavissa käytännön mittauksiin, käytetyillä parametreilla. Kuitenkin jatkotutkimuksen aihetta nousi esiin mm. globaalien muistihakujen suoritusaika-arvioissa, joita voitaisiin tarkentaa esim. käyttämällä hybridimalleja. Teoreettisen arvion ja todellisen suoritusajan eroihin vaikutti myös se, että teoreettiseen malliin ei pystytty yhdistämään täysin oikeita grafiikkasuorittimen suorituskykyarvoja. Nämä arvot ovat osittain grafiikkasuoritin ja ongelmakohtaisia, joten niiden tarkka selvittäminen tai menetelmä niiden selvittämiseksi jätettiin jatkopohdinnan aiheeksi.

Kuten esimerkkiongelmaa ratkaistessa todettiin, rinnakkainen laskenta tarjoaa mahdollisuudet ratkaista ongelmia tehokkaammin kuin mitä peräkkäisohjelma voi tehdä. Vaatimuksena on kuitenkin, että ongelma on sopiva ratkaistavaksi rinnakkaisessa suoritusympäristössä. Lisäksi optimoimalla rinnakkaisen laskennan ohjelmaa voidaan saavuttaa vielä nopeammin suoriutuva ohjelma. Koska GPU-laskenta tarjoaa helposti lähestyttävän ratkaisun nopeutukseen, tulevaisuudessa tullaan varmasti hyödyntämään rinnakkaista laskentaa vielä tehokkaammin kuin tällä hetkellä. Tästä esimerkkeinä ovat useiden erilaisten laskentaintensiivisten ohjelmien jo käyttämät rinnakkaisen laskennan menetelmät. Useammat ohjelmat tulevat jatkossa myös käyttämään laajemmin rinnakkaisen laskennan ominaisuuksia, kun ne tulevat saataville entistä korkeamman tason ohjelmointikielille. Mielenkiintoinen haara rinnakkaisessa laskennan ovat myös mobiililaitteet ja niiden muodostamat mobiililaitteiden verkot. Näitä verkkoja voidaan tulevaisuudessa hyödyntää rinnakkaiseen laskentaan.

Lähteet

[1] L. Natvig, "Computational models for parallel computing and BSPLab", Tekninen raportti, Norwegian University of Science and Technology, Trondheim, 1998.

[2] G.-L. Chen, G.-Z. Sun, Y.-Q. Zhang ja Z.-Y. Mo, *Study on Parallel Computing*, "Proceedings of the 3rd International Workshop on Frontiers in Algorithmics", Springer, Hefei, 2009.

[3] C. Kessler ja J. Keller, *Models for Parallel Computing: Review and Prespectives*, "Proceedings of PARS", vol. 24, PARS-Mitteilungen, Bonn, 2007.

[4] C. S. Ravela, "Comparison of Shared memory based parallel programming models", Pro gradu -tutkielma, Blekinge Institute of Technology, Ronneby, 2010.

[5] J. Haataja ja K. Mustikkamäki, "Rinnakkaisohjelmointi MPI:llä", CSC - Tieteellinen laskenta Oy, Espoo, 2001.

[6] Message Passing Interface Forum, "Standard, MPI: A Message-Passing Interface" [Online]. WWW-sivu: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf. [Viitattu: 5.7.2013].

[7] J. J. Dongarra, S. W. Otto, M. Snir ja D. Walker, *A Message Passing Standard for MPP and Workstations*, Communications of the ACM, vol. 39 no. 7, ACM, New York, 1996.

[8] J. T. Larsson, "Aspects of the efficient Implementation of the Message Passing Interface (MPI)", Väitöskirja, University of Copenhagen (DIKU), Kööpenhamina, 2009.

[9] L. A. Smith, *Mixed mode MPI/OpenMP programming*, "UK High-End Computing Technology Report", Edinburgh Parallel Computing Center, Edinburgh, 2000.

[10] A. Geist, J. Dongarra ja A. Beguelin, "PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing", The MIT Press, Cambridge, 1994.

[11] A. Prakash, "Parallel Virtual Machine", Seminaariraportti, Cochin University of Science & Technology, Kochi, 2010.

[12] L. Honghui, "Message Passing Versus Distributed Shared Memory on Networks of Workstations", Pro gradu -tutkielma, Rice University, Houston, 1996.

[13] A. Barak, A. Braverman, I. Gilderman ja O. Laden, *Performance of PVM with the MOSIX Preemptive Process Migration Scheme*, "Seventh Israeli Conference on Computer Systems and Software Engineering", IEEE, Jerusalem, 1996.

[14] G. L. Valiant, *A Bridging Model for Parallel Computation*, Communications of the ACM, vol. 33 no. 8, ACM, New York, 1990.

[15] B. Gianfranco, H. T. Kieran, A. Pietracaprina, G. Pucci ja P. *Spirakis, BSP vs LogP*, "Proceedings of the eight annual ACM symposium on Parallel algorithms and architectures", ACM, New York, 1996.

[16] M. Goudreau, K. Lang, S. Rao, T. Suel ja T. Tsantilas, *Towards efficiency and portability: Programming with the BSP model*, "Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures", ACM, New York, 1996.

[17] J. E. Savage, "Models of Computation, Exploring the Power of Computing", Addison-Wesley, Boston, 1998.

[18] M. Chady, "Evaluation of the BSP Model", Pro gradu -tutkielma, The University of Birmingham, Birmingham, 1997.

[19] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian ja T. von Eicken, *LogP: Towards a Realistic Model of Parallel Computation* "Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of

parallel programming", ACM, New York, 1993.

[20] R. M. Karp, A. Sahay, E. Santos, ja K. E. Schauser., "Optimal Broadcast and Summation in the LogP Model", Tekninen raportti, University of California, Berkeley, 1992.

[21] Y. S. Lee, "BSLogGP: A New Cost Model for Irregular Data Parallelsim on Clusters", Pro gradu -tutkielma, University of New South Wales, Kensington, 2005.

[22] M. Quinn, "Parallel Programming in C with MPI and OpenMP", McGraw-Hill Professional, New York, 2003.

[23] P. Gibbons, *What good are shared-memory models?*, "Proceedings of the 1996 ICPP Workshop on Challenges for Parallel Processing", Bell Labs, New Jersey, 1996.

[24] S. Fortune ja J. Wyllie, *Parallelism in random access machines*, "Proceedings of the tenth annual ACM symposium on Theory of computing", ACM, New York, 1978.

[25] P. B. Gibbons, "The Asynchronous PRAM: A Semi-Synchronous Model for Shared Memory MIMD Machines", University of California, Berkeley, 1989.

[26] M. Forsell, *On the performance and cost of some PRAM models on CMP hardware*, International Journal of Foundations of Computer Science, vol. 21 no. 3, World Scientific, Singapore, 2010.

[27] M. Penttonen, "Johdatus algoritmien sunnitteluun ja analysointiin", Otatieto, Helsinki, 1997.

[28] P. G. Gibbons, *A more practical PRAM model*, "Proceedings of the first annual ACM symposium on Parallel algorithms and architectures", ACM, New York, 1989.

[29] G. Gan, "Programming model and exceution model for OpenMP on the Cyclops-64 manycore processor", Väitöskirjan osa, University of Delaware, Delaware, 2010.

[30] J.-M. Mäkelä, "Moniydinsuorittimien rinnakkaisohjelmointi", Pro gradu -tutkielma,

Turun yliopisto, Turku, 2011.

[31] J. P. Hoeflinger ja B. R. de Supinski, *The OpenMP Memory Model*, "Proceedings of the 2005 and 2006 international conference on OpenMP shared memory parallel programming", Springer-Verlag, Berlin, 2008.

[32] T. B. Gendreau, *Using OpenMP in a parallel computing course*, The Midwest Instruction and Computing Symposium, The University of Wisconsin-La Crosse, La Crosse, 2010.

[33] OpenMP Architecture Review Board, "OpenMP Application Programming Interface", [Online]. WWW-sivu: http://www.openmp.org/mp-documents/spec30.pdf. [Viitattu: 5.7.2013].

[34] M. Suess, "User-Centric Perspective on Parallel Programming with Focus on OpenMP", Väitöskirja, University of Kassel, Kassel, 2007.

[35] J. Gabriele, J. Haoqiang, A. M. Dieter ja H. F. Ferhat, *Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP*, "Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing", IEEE Computer Society, Weimar, 2009.

[36] B. Grayson, M. Dahlin ja V. Ramachandran, *Experimental Evaluation of QSM, a Simple Shared Memory Model*, "13th International and 10th Symposium on Parallel and Distributed Processing", IEEE, San Juan, 1999.

[37] V. Ramachandran, *A General Purpose Shared-Memory Model For Parallel Computation*, "IMA Workshop on Parallel Algorithms", Springer-Verlag, Minneapolis, 1997.

[38] J. T. Harris ja I. M. Cole, *The Parametrized PRAM*, "Proceedings of the Workshop on Parallel and Distributed Processing", Elsevier Press, Amsterdam, 1993.

[39] G. A. Geist, J. A. Kohl ja P. M. Papadopulos, *PVM and MPI: A comparison of*

features, Calculateurs Paralleles, vol. 8, Hermès science, London, 1996.

[40] E. Elts, "Comparative analysis of PVM and MPI for the development of physical applications on parallel clusters", Tekninen raportti, Saint-Petersburg State University, Saint-Petersburg, 2004.

[41] G. Bilardi, K. T. Herley, A. Pietracaprina, G. Pucci ja P. Spirakis, *BSP vs LogP*, "Proceedings of the eigth annual ACM symposium on Parallel algorithms and architectures", ACM, New York, 1996.

[42] R. Niedermeier ja P. Rossmanith, *PRAM's towards realistic parallelism: BRAM's*, "10th International Conference, FCT '95", Springer, Dresden, 1995.

[43] S. Theubl, "Applied High Performance Computing Using R", Diplomityö, University of Wien, Wien, 2007.

[44] M. J. Chorley, D. W. Walker ja M. F. Guest, *Hybrid Message-Passing And Shared-Memory Programming In A Molecular Dynamics Application On Multicore Clusters*, The International Journal of High Performance Computing Applications, vol. 23 no. 3, Sage Publications, Thousand Oaks, 2009.

[45] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale ja R. Thakur, *MPI+MPI: A New Hybrid Approach to Parallel Programming with MPI Plus Shared Memory*, Computing, vol. 95 no. 12, Springer-Verlag, Wien, 2013.

[46] L. E. Charles , *The Cilk++ concurrency platform*, "Proceedings of the 46th Annual Design Automation Conference", ACM, New York, 2009.

[47] B. Lewis ja B. J. Daniel, "Threads Primer: A Guide to Multithreaded Programming", Prentice Hall, New Jersey, 1995.

[48] C. P. Leonardo, "Hybrid Parallel Programming - Evaluation of OpenACC", Kandidaatintutkielma, Universidade Federal do Rio Grande, Rio Grande, 2012.

[49] B.-N. Amotz ja K. Shlomo, *Designing broadcasting algorithms in the postal model for message-passing systems*, "Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures", ACM, New York, 1992.

[50] M. Yossi, V. Ramachandran ja P. Gibbons, *The Queue-Read Queue-Write Asynchronous PRAM Model*, "Proceedings of the Second International Euro-Par Conference on Parallel Processing", Springer-Verlag, Lontoo, 1996.

[51] J. Reinders, "Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism", O'Reilly Media, Sebastopol, 2007.

[52] J. E. Stone, D. Gohara ja G. Shi, *OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems*, IEEE Design & Test, vol. 12 no. 3, IEEE, 2010.

[53] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston ja P. Hanrahan, *Brook for GPUs: Stream Computing on Graphics Hardware*, "ACM SIGGRAPH 2004 Papers", ACM, New York, 2004.

[54] J. Nickolls ja D. Kirk, *Appendix A: Graphics and Computing GPUs*, "Computer Organization and Design, Revised Fourth Edition", Morgan Kaufmann Publishers, Burlington, 2008.

[55] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone ja J. Phillips, *GPU Computing*, "Proceedings of the IEEE", vol. 96 no. 5, *IEEE, New York*, 2008.

[56] J. Palacios ja J. Triska, "A Comparison of Modern GPU and CPU Architectures: And the Common Convergence of Both", Tekninen raportti, Oregon State University, Oregon, 2011.

[57] M. Johansson ja O. Winter, "General purpose computing on graphics processing units using OpenCL", Pro gradu -tutkielma, Chalmers University of Technology, Göteborgm 2011.

[58] NVIDIA, "CUDA C Programming Guide", versio 4.2, [Online]. WWW-sivu:

docs.nvidia.com/cuda/cuda-c-programming-guide/. [Viitattu: 25.10.2012].

[59] S. A. Hørup, S. A. Juul ja H. H. Larsen, "The Art of General-Purpose Computations on Graphics Processing Units", Tekninen raportti, Aalborg University, Aalborg, 2011.

[60] NVIDIA, "Whitepaper, Fermi Compute Architecture: Fermi", [Online]. WWW-sivu: www.nvidia.com/content/PDF/fermi_white_papers/ NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. [Viitattu: 30.10.2012].

[61] F. Lionetti, "GPU Accelerated Cardiac Electrophysiology", Pro gradu -tutkielma, University of California, San Diego, 2010.

[62] T. Wood, "Applications of GPUs in Computational Finance", Pro gradu -tutkielma, Universiteit van Amsterdam, Amsterdam 2010.

[63] S. Sengupta, "Efficient Primitives and Algorithms for Many-core architectures", Väitöskirjan osa, University of California, Davis, 2010.

[64] A. Resios, "GPU performance prediction using parametrized models", Pro gradu -tutkielma, Utrecht University, Utrech, 2011.

[65] B. A. Ruud, "Parallel alignment of short sequence reads on graphics processors", Pro gradu -tutkielma, University of Oslo, Oslo, 2011.

[66] G. Karch, "GPU-based acceleration of selected clustering techinques", Silesian Pro gradu -tutkielma, University of Technology, Gliwice, 2010.

[67] J. Sanders ja E. Kandrot, "CUDA by example", "Addison-Wesley, Boston, 2011.

[68] I. Buck, M. Garland ja J. Nickolls, *Scalable Parallel Programming with CUDA*, "Queue - GPU Computing", vol. 6 no. 2, ACM, New York, 2008.

[69] B. K. David ja W.-m. W. Hwu, "Programming Massively Parallel Processors, Second Edition: A Hands-on Approach", Morgan Kaufmann Publishers, Burlington, 2012.

[70] Y. Torres, A. Gonzalez-Escribano ja D. R. Llanos, *Understanding the Impact of*

CUDA Tuning Techniques for Fermi, "The 2011 International Conference on High Performance Computing & Simulation", IEEE, Istanbul, 2011.

[71] N. Whitehead ja A. Fit-Florea, "Precision and Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs", [Online]. WWW-sivu: https://developer.nvidia.com/sites/default/files/akamai/cuda/files/NVIDIA-CUDA-Floating-Point.pdf [Viitattu: 1.9.2013].

[72] M. Bauer, H. Cook ja B. Khailany, *CudaDMA: optimizing GPU memory bandwidth via warp specialization*, "Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis", ACM, New York, 2011.

[73] J. C. María, J. M. Garcìa ja M. Ujaldòn, *The GPU on the Matrix-Matrix Multiply: Performance Study and Contributions*, "Advances in Parallel Computing", vol. 19, IOS Press, Amsterdam, 2010.

[74] Y. Torres, A. Gonzalez-Escribano ja D. R. Llanos, *Understanding the Impact of CUDA Tuning Techniques for Fermi*, "The 2011 International Conference on High Performance Computing & Simulation", IEEE, Istanbul, 2011.

[75] M. Gillespie, "Amdahl's Law, Gustafson's Trend, and the Performance Limits of Parallel Applications", [Online]. WWW-sivu: http://software.intel.com/sites/default/files/m/d/4/1/d/8/Gillespie-0053-AAD_Gustafson-Amdahl_v1__2_.rh.final.pdf [Viitattu: 15.7.2013].

[76] S. Sahni ja V. Thanvantri, "Parallel Computing: Performance Metrics and Models", Tekninen raportti, University of Florida, Gainesville, 1996.

[77] S. Robinson, *Toward an Optimal Algorithm for Matrix Multiplication*, SIAM News, vol. 38 no. 9, Society for Industrial and Applied Mathematics, Philadelphia, 2005.

[78] D. L. Baggio, "GPGPU based image segmentation Livewire algorithm implementation", Pro gradu -tutkielma, Campo Montenegro, São Josè dos Campos,

2007.

[79] Intel, "Intel® Core i7-3900 Desktop Processor Extreme Edition Series", [Online]. WWW-sivu: http://download.intel.com/support/processors/corei7ee/sb/core_i7-3900_d_x.pdf. [Viitattu 30.10.2012].

[80] S. Crow, "Evolution of the Graphical Processing Unit", Pro gradu -tutkielma , University of Nevada, Reno, 2004.

[81] R. Farber, "CUDA Application Design", Morgan Kaufmann Publishers, Burlington, 2011.

[82] M. Bauer, H. Cook ja B. Khailany, *CudaDMA: Optimizing GPU Memory Bandwidth via Warp*, "Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis", ACM, New York, 2011.

[83] L. A. Smith, *Mixed Mode MPI / OpenMP Programming*, "UK High-End Computing Technology Report", The University of Edinburgh, Edinburgh, 2000.

[84] D. Blythe, *Rise of the Graphics Processor*, "Proceedings of The IEEE", vol. 96 no. 5, IEEE, New York, 2008.

[85] NVIDIA, "Whitepaper NVIDIA GeForce GTX 680", [Online]. WWW-sivu: http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf. [Viitattu 30.10.2012].

Liitteet

A Matriisien kertolasku CPU:lla

```
void matrixMultiplication(int* a, int* b, int* c, int size)
{
    for(int i = 0; i < size; i ++)
    {
         for(int j = 0; j < size; j ++)
         {
             for(int x = 0; x < size; x++)
                  c[i*size+j] +=  a[i*size+x]*b[x*size+j];
         }
    }
}
```

B Matriisien kertolasku GPU:lla, pääohjelma

```
#define TILE_WIDTH 32
int TILE_SIZES[] ={2,4,8,16,32};
int tileSizes = 5;
int main()
{
     srand (time(NULL));
     int ii = 0;
     bool first = true;
     int initValue = 0;
     int tile = TILE_WIDTH;
     cudaError_t cudaStatus;
     int m = 1024;
     tile = 4;

     matrix_type *a = (matrix_type*)malloc(m*m*sizeof(matrix_type));
     matrix_type *b = (matrix_type*)malloc(m*m*sizeof(matrix_type));
     matrix_type* c = (matrix_type*)malloc(m*m*sizeof(matrix_type));
     matrix_type* ccuda = (matrix_type*)malloc(m*m*sizeof(matrix_type));

     float timeMS_GPU = 0.0;
     for(int i = 0; i < 10; i++)
     {

       fillMatrix(a,m);
       fillMatrix(b,m);
       memset(c,0,sizeof(matrix_type)*m*m);

       cudaStatus = cudaDeviceSetCacheConfig(cudaFuncCachePreferShared )
                    ;
       cudaEvent_t start, stop;
       cudaEventCreate( &start );
       cudaEventCreate( &stop );
       cudaEventRecord( start, 0 );

       cudaStatus = addWithCuda(c,ccuda, a, b, m,TILE_SIZES[tile],true);
```

```
        cudaEventRecord( stop, 0 );
        cudaEventSynchronize( stop );
        float elapsedTime;
        cudaEventElapsedTime( &elapsedTime,start, stop );

        cudaEventDestroy( start );
        cudaEventDestroy( stop );

        if (cudaStatus != cudaSuccess)
        {
        fprintf(stderr, "addWithCuda failed!");
        return 1;
        }
        timeMS_GPU += elapsedTime;
    }

    printf("%d:%d:%1.5f\n",m,TILE_SIZES[tile],timeMS_GPU/10.00);

    free(a); a= NULL;
    free(b);b= NULL;
    free(c);c= NULL;
    free(ccuda);ccuda= NULL;

    cudaStatus = cudaDeviceReset();
    if (cudaStatus != cudaSuccess)
    {
        fprintf(stderr, "cudaDeviceReset failed!");
        return 1;
    }

    return 0;
}
```

```
cudaError_t addWithCuda(matrix_type *c, matrix_type* ccuda, const mat-
                        rix_type *a, const matrix_type *b, unsigned int
                        size,  int tileSize, bool useShared)
{
    matrix_type *dev_a = 0;
    matrix_type *dev_b = 0;
    matrix_type *dev_c = 0;
    cudaError_t cudaStatus;
    cudaStatus = cudaSetDevice(0);
    cudaStatus = cudaMalloc((void**)&dev_c, size *
                                size*sizeof(matrix_type));
    if (cudaStatus != cudaSuccess)
    {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }

    cudaStatus = cudaMalloc((void**)&dev_a, size *
                                size*sizeof(matrix_type));
    if (cudaStatus != cudaSuccess)
    {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }

    cudaStatus = cudaMalloc((void**)&dev_b, size *
                                size*sizeof(matrix_type));
    if (cudaStatus != cudaSuccess)
    {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }

    cudaStatus = cudaMemcpy(dev_a, a, size * size *
                        sizeof(matrix_type), cudaMemcpyHostToDevice);
    if (cudaStatus != cudaSuccess)
    {
        fprintf(stderr, "cudaMemcpy failed!");
```

```
    goto Error;
}

 cudaStatus = cudaMemcpy(dev_b, b, size * size* sizeof(matrix_type),
                    cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess)
{
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}

 cudaStatus = cudaMemcpy(dev_c, c, size * size* sizeof(matrix_type),
                    cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess)
{
    fprintf(stderr, "cudaMemcpy failed!");
    goto Error;
}



if(!useShared)
{
     dim3 dimGrid(size/tileSize, size/tileSize);
     dim3 dimBlock(tileSize, tileSize);
     addKernel<<<dimGrid, dimBlock>>>(dev_c, dev_a, dev_b,size);
}
else
{
     dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
     dim3 dimGrid((size-1)/TILE_WIDTH+1, (size-1)/TILE_WIDTH+1, 1);
     addKernelTileShared<<<dimGrid,dimBlock>>> ( dev_a , dev_b,dev_c,
                                                              size);
}
cudaStatus = cudaGetLastError();
if (cudaStatus != cudaSuccess)
{
    fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorSt
                    ring(cudaStatus));
```

```
        goto Error;
    }

    cudaStatus = cudaDeviceSynchronize();
    if (cudaStatus != cudaSuccess)
    {
        fprintf(stderr, "cudaDeviceSynchronize returned error code %d
                        after lauching addKernel!\n", cudaStatus);
        goto Error;
    }
    cudaStatus = cudaMemcpy(ccuda, dev_c, size *size* sizeof(int), cuda-
                        MemcpyDeviceToHost);
    if (cudaStatus != cudaSuccess)
    {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }
    Error:
        cudaFree(dev_c);
        cudaFree(dev_a);
        cudaFree(dev_b);

    return cudaStatus;
}
```

C Matriisien kertolasku GPU:lla, perusversio, ydin

```
__global__ void addKernel(matrix_type *c, const matrix_type *a,
                                  const matrix_type *b, int size)
{
          matrix_type Cvalue = 0;
          int row = (blockIdx.y*blockDim.y) + threadIdx.y;
          int col = (blockIdx.x*blockDim.x) + threadIdx.x;

          for (int e = 0; e < size; ++e)
          {
                 Cvalue += a[row * size + e] * b[e * size + col];
          }

          c[row * size + col] = Cvalue;
}
```

D Matriisien kertolasku GPU:lla, jaetun muistin versio, ydin

```
__global__ void addKernelTileShared(matrix_type* A, matrix_type* B,
                                    matrix_type* C, int Width)
{
      int bx = blockIdx.x;
      int by = blockIdx.y;
      int tx = threadIdx.x;
      int ty = threadIdx.y;
      int aBegin = Width * TILE_WIDTH * by;
      int aEnd = aBegin + Width - 1;
      int aStep = TILE_WIDTH;
      int bBegin = TILE_WIDTH * bx;
      int bStep = TILE_WIDTH * Width;
      float Csub = 0;
      for (int a = aBegin, b = bBegin;      a <= aEnd; a += aStep,
                                            b += bStep)
      {
          __shared__ float As[TILE_WIDTH][TILE_WIDTH];
          __shared__ float Bs[TILE_WIDTH][TILE_WIDTH];
          As[ty][tx] = A[a + Width * ty + tx];
          Bs[ty][tx] = B[b + Width * ty + tx];

          __syncthreads();

          for (int k = 0; k < TILE_WIDTH; ++k)
                 Csub += As[ty][k] * Bs[k][tx];

          __syncthreads();
      }
      int c = Width * TILE_WIDTH * by + TILE_WIDTH * bx;
      C[c + Width * ty + tx] = Csub;
}
```

E Mittaustuloksia testiajoista CPU-ohjelma vs. GPU-ohjelma

m x m	CPU	GPU
32	0,1	0,9
64	0,6	1,0
96	1,6	1,1
128	3,8	1,4
160	7,6	2,3
192	12,8	2,6
224	19,0	4,1
256	48,1	5,7
288	41,0	7,6
320	80,3	10,2
352	79,5	13,4
384	171,2	16,5
416	146,9	20,8
448	292,1	25,3
480	229,0	30,7
512	861,1	36,6
544	376,2	43,8
576	650,8	50,8
608	614,8	59,9
640	886,2	68,7
672	943,3	80,1
704	1210,5	90,4
736	1485,6	104,3
768	3087,4	115,7
800	2833,3	132,0
832	3791,9	145,7
864	4915,4	164,4
896	6090,7	181,4
928	7430,2	202,3
960	9039,7	221,8
992	10764,3	245,9
1024	12674,4	268,4

F Mittaustuloksia globaalin muistin ja jaetun muistin ohjelmista

	Globaalin muistin ohjelma suoritusajat (ms)						
	m x m						
b x b		**1x1**	**2x2**	**4x4**	**8x8**	**16x16**	**32x32**
	1024x1024	-	-	2071,7	528,7	270,8	272,0
	512x512	-	-	261,9	68,9	36,7	36,4
	256x256	-	-	31,7	9,4	5,5	5,6
	128x128	90,9	16,0	3,7	1,6	1,4	1,3

	Jaetun muistin ohjelma suoritusajat (ms)						
	m x m						
b x b		**1x1**	**2x2**	**4x4**	**8x8**	**16x16**	**32x32**
	1024x1024	-	-	1532,9	313,2	174,1	168,0
	512x512	-	-	192,0	41,1	25,1	24,5
	256x256	-	-	25,7	6,1	4,1	4,5
	128x128	40,8	11,7	3,9	1,7	1,6	1,2

Taulukoissa m kuvaa matriisin sivun pituutta ja b lohkon sivun pituutta.