Tampereen yliopisto

Anton Ihonen

NVIDIAN GPGPU-TEKNIIKKA

Tekninen katsaus CUDA-ohjelmointialustaan

Kandidaatintyö
Informaatioteknologian ja viestinnän tiedekunta
Tarkastaja: Matti Haavisto
Maaliskuu 2022

TIIVISTELMÄ

Anton Ihonen: Nvidian GPGPU-tekniikka
Kandidaatintyö
Tampereen yliopisto
Tieto- ja sähkötekniikan TkK-tutkinto-ohjelma
Maaliskuu 2022

Nykyiset grafiikkaprosessorit ovat ohjelmoitavia, massiivisen rinnakkaisia prosessoreja, joiden laskentatehoa hyödyntämällä voidaan parantaa rinnakkaistuvan sovelluksen suorituskykyä. Tässä opinnäytetyössä perehdyttiin Nvidian CUDA-ohjelmointialustaan, sen toteutustekniikkaan ja sen mahdollistamiin suorituskykyhyötyihin.

Heterogeenisessä laskennassa ohjelman rinnakkainen osa suoritetaan tietokoneen keskusprosessorin sijaan toisella laitteella. GPGPU-laskenta on heterogeenistä laskentaa, jossa ideaalisesti keskusprosessori suorittaa ohjelman sekventiaalisen ja grafiikkaprosessori rinnakkaisen osan. Grafiikkaprosessorit on suunniteltu juuri tähän: kontrollilogiikan ja välimuistien osuus piirin pinta-alasta on minimoitu ja laskentaelementtien puolestaan maksimoitu. Tämä yhdessä muistiväylän suuren kaistanleveyden ja laitteistotason vuoronnuksen kanssa mahdollistaa tehokkaan ja joustavan rinnakkaisen laskennan.

GPGPU-sovelluksia tarkasteltaessa esiin nousee joukko erityispiirteitä. Näitä ovat runsas datatason rinnakkaisuus, ratkaistavan ongelman suuri koko, korkea aritmeettinen intensiteetti ja korkeat suorituskykyvaatimukset. Grafiikkaprosessorit soveltuvat arkkitehtuurinsa vuoksi tällaisiin sovelluksiin erityisen hyvin.

GPGPU-ohjelmointi eroaa oleellisesti yleiskäyttöisen prosessorin ohjelmoinnista. Nvidian grafiikkaprosessoreja ohjelmoidaan CUDA-ohjelmointialustan ja -rajapinnan avulla. CUDA noudattaa SPMD-mallia, jossa sama ohjelma suoritetaan lukemattomilla rinnakkaisilla säikeillä. Se myös mahdollistaa grafiikkaprosessorin verrattain helposti lähestyttävän ohjelmoinnin GPGPU-ohjelmoinnin tarpeisiin mukautetulla C++-ohjelmointikielen murteella. CUDA:n tekninen toteutus nojautuukin pitkälti CUDA-ohjelmien kääntämiseen tarkoitettuun NVCC-kääntäjään.

Erityispiirteidensä ansiosta grafiikkaprosessorit ovat rinnakkaisessa laskennassa sekä laskentatehon että energiatehokkuuden suhteen huomattavasti parempia kuin yleiskäyttöiset prosessorit. Työssä havaittiin, että monissa käytännön sovelluksissa tavanomaisen grafiikkaprosessorin hyödyntäminen yleiskäyttöisen prosessorin ohella voi parantaa suorituskykyä jopa kertaluokalla. Lisäksi hyvin rinnakkaistuvan sovelluksen energiankulutus voidaan grafiikkaprosessorin avulla pudottaa jopa sadasosaan alkuperäisestä.

Avainsanat: Nvidia, grafiikkaprosessori, CUDA, GPGPU, SPMD, SIMT

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

SISÄLLYSLUETTELO

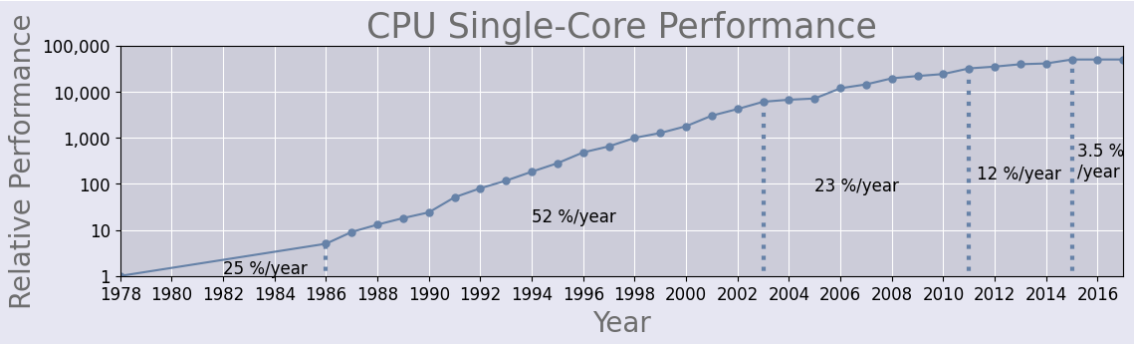
LYHENTEET JA MERKINNÄT

API	application programming interface
C++ AMP	C++ Accelerated Massive Parallelism
CPU	central processing unit
CUDA	Compute Unified Device Architecture
FLOPS	floating point operations per second
FP16	16-bit floating point
FP32	32-bit floating point
FP64	64-bit floating point
FPGA	field-programmable gate array
GDDR6X	Graphics Double Data Rate 6
GPC	graphics processing cluster
GPGPU	general-purpose computing on graphics processing units
GPU	graphics processing unit
HIP	Heterogeneous Computing Interface for Portability
INT32	32-bit integer
ISA	instruction set architecture
JIT	just-in-time
MIMD	multiple instruction streams, multiple data streams
MISD	multiple instruction streams, single data stream
NVCC	Nvidia CUDA Compiler
NVRTC	Nvidia Runtime Compilation
OpenACC	Open Accelerators
OpenCL	Open Computing Language
OpenGL	Open Graphics Library
OpenMP	Open Multi-Processing
PAM4	4-level pulse-amplitude modulation
PCI-e	Peripheral Component Interconnect Express

PTX	Parallel Thread Execution
ROP	raster operation processor
SDRAM	synchronous dynamic random-access memory
SFU	special function unit
SIMD	single instruction stream, multiple data streams
SIMT	single instruction stream, multiple threads
SISD	single instruction stream, single data stream
SM	streaming multiprocessor
SPMD	single program stream, multiple data streams
STL	Standard Template Library
TPC	texture processing cluster

1. JOHDANTO

Tietokoneiden yleiskäyttöisten keskusprosessorien (central processing unit, CPU) ydinkohtaisen suorituskyvyn kehittyminen on 2000-luvun kuluessa hidastunut ja lähestulkoon pysähtynyt[1, s. 55] (kuva 1.1). Toisaalta fyysinen koko ja lämmöntuotto asettavat rajat ytimien lukumäärälle, mikä puolestaan rajoittaa yleiskäyttöisten prosessorien suorituskykyä rinnakkaisessa laskennassa. Samaan aikaan tietotekniset sovellukset edellyttävät laitteistolta yhä suurempaa laskentatehoa. Riittävän suorituskyvyn takaamiseksi on siis yhä useammin etsittävä uudenlaisia ratkaisuja.



Kuva 1.1. *Perinteisten yleiskäyttöisten prosessorien ydinkohtaisen suorituskyvyn kehittyminen on 2000-luvun aikana hidastunut huomattavasti. Perustuu lähteeseen [1, s. 55].*

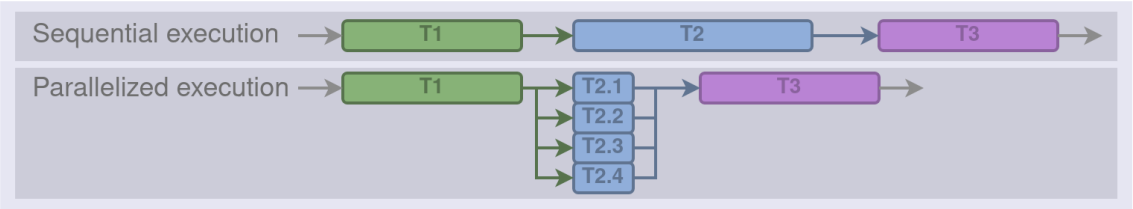
Yksi ratkaisu suorituskykyongelmaan on GPGPU-laskenta — rinnakkaisessa laskennassa loistavien grafiikkaprosessorien hyödyntäminen muussa kuin tietokonegrafiikkaan liittyvässä laskennassa (general-purpose computing on graphics processing units). GPGPU-laskenta ei ole tekniikkana uusi, mutta edellä mainituista syistä se on nyt relevantimpi kuin koskaan.

Nvidian vuonna 2007 julkaisema CUDA-ohjelmointialusta ja -rajapinta (Compute Unified Device Architecture) on edelleen GPGPU-tekniikan edelläkävijä. CUDAn tavoite on tehdä grafiikkaprosessorien rinnakkaisesta ohjelmoinnista mahdollisimman helposti lähestyttävää. Tämä työ on tekninen katsaus CUDAan ja sen toteutustekniikkaan. Samalla se toimii johdatuksena CUDA-ohjelmoinnin perusteisiin. Työssä tarkastellaan lyhyesti myös GPGPU-laskennan suorituskykyä ja energiatehokkuutta.

Työn rakenne on seuraava: luku 2 toimii johdantona rinnakkaiseen laskentaan. Luvussa 3 esitellään Nvidian uusimpien Ampere-grafiikkaprosessorien rakennetta ja toimintaa. Luvussa 4 perehdytään CUDA-ohjelmointimallin perusteisiin. Luvussa 6 tarkastellaan GPGPU-laskennan tehokkuutta. Luku 7 on yhteenveto työstä.

2. RINNAKKAISEN LASKENNAN TEORIAA

Tietokoneiden suorittamat tehtävät voidaan jakaa sekventiaalisiin (sequential) eli peräkkäisiin ja rinnakkaisiin (parallel) tehtäviin. Sekventiaalisen tehtävän osat suoritetaan peräkkäin määrätyssä järjestyksessä, kun taas rinnakkaisen tehtävän osat suoritetaan keskenään samanaikaisesti. Jos sekventiaalinen tehtävä voidaan muuttaa rinnakkaiseksi, se on rinnakkaistuva (parallelizable). Rinnakkaistamalla voidaan nopeuttaa rinnakkaistuvaa sovellusta, mikäli rinnakkaisen laskennan edellyttämät laskentaresurssit ovat käytettävissä. [2, s. 9, s. 39] Tämä on havainnollistettu kuvassa 2.1.



Kuva 2.1. *Rinnakkaistuvaa sovellusta voidaan nopeuttaa rinnakkaistamalla.*

Rinnakkaisuudesta puhuttaessa voidaan kontekstista riippuen tarkoittaa joko todellista rinnakkaisuutta (parallelism) tai näennäistä rinnakkaisuutta eli yhtäaikaisuutta (concurrency). Tässä työssä rinnakkaisuudella tarkoitetaan pääasiassa todellista rinnakkaisuutta, jossa laskentaa suorittaa samanaikaisesti monta eri fyysistä laskentaelementtiä eli prosessoria.

2.1 Laskennan nopeutuminen

Nopeutuminen (speedup) on suure, joka kertoo kuinka monta kertaa nopeampi ohjelma on laskennan rinnakkaistamisen jälkeen [2, s. 10]. Jos esimerkiksi ohjelman suoritusaika puolittuu, ohjelma nopeutuu kaksinkertaisesti eli nopeutuminen on siis 2.

Ideaalisesti ohjelman rinnakkaistettava osa nopeutuu samassa suhteessa kuin sitä suorittavien prosessorien lukumäärä kasvaa. Tosiasiassa tilanne ei kuitenkaan yleensä ole aivan näin yksinkertainen, sillä prosessorit joutuvat usein tekemään yhteistyötä, jolloin osa suoritusajasta kuluu prosessorien väliseen kommunikointiin.

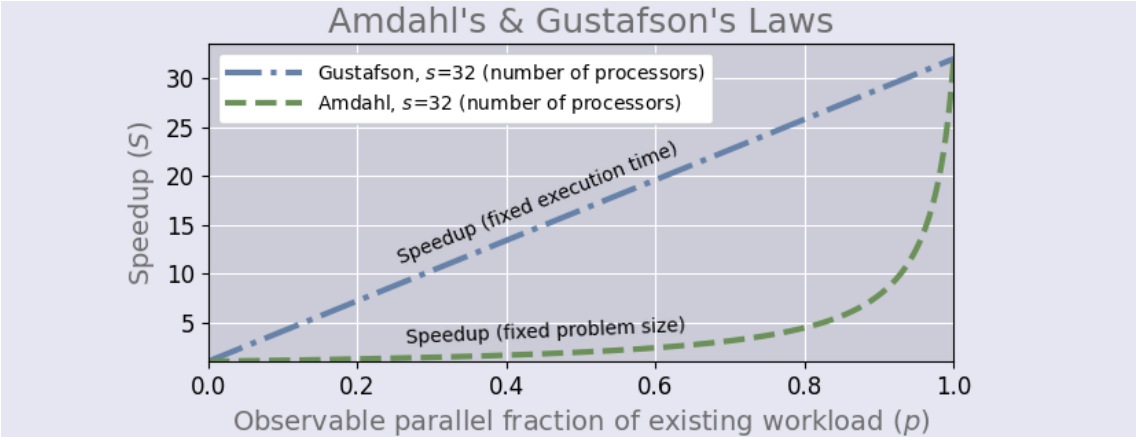
Tarkasti nopeutuminen saadaan selville mittaamalla ohjelman suoritusaika ennen rinnakkaistamista ja sen jälkeen. Nopeutumista voidaan kuitenkin arvioida myös etukäteen hyö-

dyntämällä Gene M. Amdahlin v. 1967 esittämää Amdahlin lakia[3][2, s. 37]

$$S = \frac{1}{1 - p + \frac{p}{s}} , \tag{2.1}$$

jossa

- S on ohjelman teoreettinen nopeutumiskerroin,
- p on ohjelman rinnakkaistettavan osan osuus ohjelman kokonaissuoritusajasta ja
- s on ohjelman rinnakkaistettavan osan nopeutumiskerroin.



Kuva 2.2. *Amdahlin laki kuvaa ohjelman nopeutumista laskennan määrän ollessa vakio. Gustafsonin laki kuvaa ohjelman nopeutumista laskentaan käytettävän ajan ollessa vakio. Perustuu lähteeseen [4].*

Amdahlin laki rajoittuu tilanteisiin, joissa laskennan määrä pysyy vakiona[2, s. 40]. Koska laskennan nopeutuminen kuitenkin mahdollistaa suuremman laskentamäärän suorittamisen samassa ajassa, on usein käytännöllisempää tarkastella tilannetta, jossa laskennan *määrän* sijaan laskentaan käytetty *aika* on vakio. Tämän ottaa huomioon John L. Gustafsonin v. 1988 esittämä Gustafsonin laki[5], joka yhtälön 2.1 merkintöjä käyttäen voidaan muotoilla

$$S = 1 - p + sp . \tag{2.2}$$

Amdahlin ja Gustafsonin laeilla ei voida eksaktisti ennustaa ohjelman nopeutumista, koska ne sisältävät oletuksen edellä mainitusta ideaalitilanteesta, jossa ohjelman rinnakkaistettava osa nopeutuu samassa suhteessa kuin laskentaa suorittavien prosessorien lukumäärä kasvaa. Niiden avulla voidaan kuitenkin arvioida, kuinka paljon sovellus *enintään* nopeutuu rinnakkaistamisen seurauksena[6]. Kuvassa 2.2 on visualisoitu Amdahlin ja Gustafsonin lakien mukainen nopeutuminen ohjelman rinnakkaistettavan osuuden funktiona, kun rinnakkaistettava osuus suoritetaan 1 prosessorin sijaan 32 prosessorilla.

2.2 Heterogeeninen laskenta

Erilaiset prosessorit soveltuvat erilaisiin laskentasovelluksiin, joten tietokoneen suorituskykyä voidaan parantaa erilaisia prosessoreja sopivalla tavalla yhdistelemällä. Tällöin osa laskennasta tyypillisesti suoritetaan tietokoneen yleiskäyttöisellä keskusprosessorilla ja osa siihen yhteydessä olevalla toisella laitteella. Tällaista järjestelyä nimitetään *heterogeeniseksi* (heterogeneous). [7, s. 52–53]

Heterogeenisessä laskennassa (heterogeneous computing) voidaan yleiskäyttöisen prosessorin lisäksi hyödyntää muunlaista ohjelmoitavaa prosessoria (esim. grafiikkaprosessoria) tai muuta laitetta (esim. FPGA-piiriä (field-programmable gate array)). Edellisessä tapauksessa voidaan puhua *apuprosessorista* (coprocessor); geneerisempi termi *kiihdytin* (accelerator) kattaa molemmat tapaukset. Rinnakkaisen laskennan suorittamista toisella, nopeammalla laitteella kutsutaan laitekiihdytykseksi (hardware acceleration). [7, s. 53]

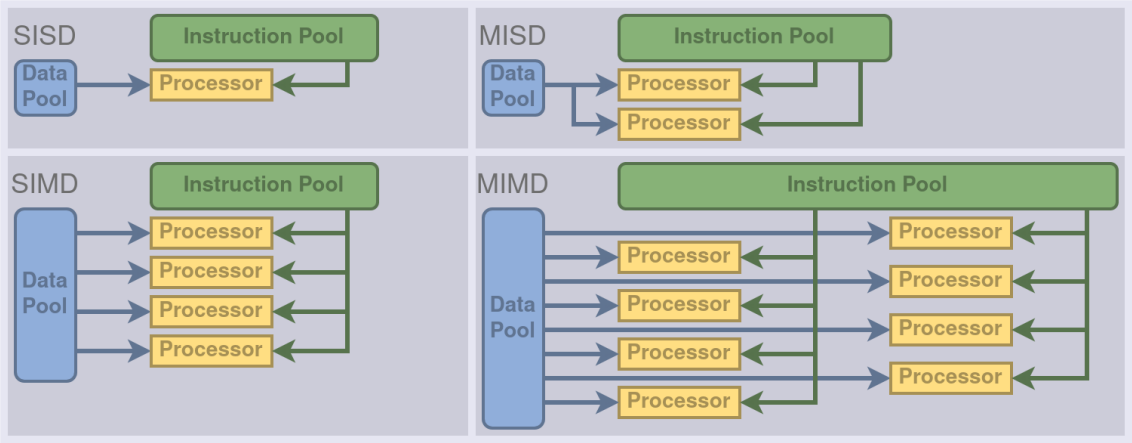
GPGPU-laskenta on yksi heterogeenisen laskennan sovelluksista.

2.3 Rinnakkaiset tietokonearkkitehtuurit

Rinnakkaisuus voidaan jakaa karkeasti kahteen luokkaan: datatason rinnakkaisuuteen (data-level parallelism) ja tehtävätason rinnakkaisuuteen (task-level parallelism). Datatason rinnakkaisuudella tarkoitetaan tilannetta, jossa montaa data-alkiota voidaan käsitellä samanaikaisesti. Vastaavasti tehtävätason rinnakkaisuudella tarkoitetaan tilannetta, jossa montaa tehtävää voidaan suorittaa toisistaan riippumatta. [8, s. 9]

Tietokoneet voivat hyödyntää data-ja tehtävätason rinnakkaisuutta useilla eri tavoilla[8, s. 9]. Perinteisesti tietokonearkkitehtuurit on jaettu rinnakkaisuutensa perusteella Michael J. Flynnin[9] v. 1966 esittelemiin neljään kategoriaan. David A. Patterson ja John L. Hennessy[1, s. 527–528][8, s. 10] luonnehtivat Flynnin esittämiä kategorioita seuraavasti:

- **SISD** (single instruction stream, single data stream). SISD-prosessorit ovat perinteisiä, yksiytimisiä prosessoreja, joissa yksittäinen käsky kohdistetaan yksittäiseen data-alkioon.
- **SIMD** (single instruction stream, multiple data streams). SIMD-prosessorit ovat prosessoreja, joissa yksittäinen käsky voidaan kohdistaa useaan data-alkioon kerrallaan (esim. x86-64-prosessorien vektorikäskyt).
- **MISD** (multiple instruction streams, single data stream). MISD-prosessorit ovat lähestulkoon sukupuuttoon kuolleita prosessoreja, joissa usea käsky kohdistetaan samanaikaisesti samaan data-alkioon.
- **MIMD** (multiple instruction streams, multiple data streams). MIMD-prosessorit ovat *moniprosessoreja* (multiprocessor), joissa useat itsenäiset prosessorit tyypillisesti



Kuva 2.3. *Havainnollistus Flynnin luokittelun mukaisista tietokonearkkitehtuureista. Perustuu lähteeseen [10].*

jakavat saman muistiavaruuden[1, s. 475] ja suorittavat eri data-alkioihin kohdistuvia erillisiä käskyjä samanaikaisesti. Sekä grafiikkaprosessorit että nykyiset yleiskäyttöiset moniydinprosessorit ovat MIMD-prosessoreja.

On huomattava, että monet nykyiset prosessorit ovat Flynnin luokittelun näkökulmasta hybridejä, jotka eivät kuulu puhtaasti vain yhteen kategoriaan[1, s. 10]. Esimerkiksi grafiikkaprosessoreissa yhdistyvät sekä SIMD että MIMD. Flynnin luokittelun kategorioiden erot on havainnollistettu kuvassa 2.3.

Grafiikkaprosessorien monisuoritinarkkitehtuuriin viitataan usein *SIMT-arkkitehtuurina* (single instruction stream, multiple threads). Tällaisessa arkkitehtuurissa monta samaan prosessoriin kuuluvaa säiettä suorittaa saman käskyn samaan aikaan[11, s. 27]. SIMT-arkkitehtuuria esitellään tarkemmin luvussa 3.

Grafiikkaprosessorien ohjelmoinnin yhteydessä puhutaan usein *SPMD-ohjelmointimallista* (single program stream, multiple data streams). Kyseisessä mallissa yksi ja sama ohjelma suoritetaan monella eri prosessorilla[1, s. 527]. CUDA-ohjelmointimallia esitellään luvussa 4.

2.4 Rinnakkainen laskenta grafiikkaprosessorilla

Grafiikkaprosessorit kehitettiin alunperin tietokonegrafiikan vaatimaa raskasta rinnakkaista laskentaa kiihdyttämään. Ensimmäiset grafiikkaprosessorit olivat kiinteätoimintoisia, joten ne eivät soveltuneet muihin tarkoituksiin. Ohjelmoitavuuden myötä niistä on kuitenkin tullut erittäin käyttökelpoisia kaikenlaisessa rinnakkaisessa laskennassa. [12]

Kuten yleiskäyttöisilläkin prosessoreilla, grafiikkaprosessoreilla on omat vahvuutensa ja heikkoutensa. Luonnollisesti grafiikkaprosessoreja pyritään hyödyntämään sovelluksissa, joiden tarpeet kohtaavat hyvin grafiikkaprosessorien vahvuuksien kanssa. Niinpä GPG-

PU-sovelluksilla on yleisesti tiettyjä yhteisiä piirteitä. Näitä ovat mm. seuraavat[11, s. 24]:

- **Runsas datatason rinnakkaisuus**. Grafiikkaprosessorit on varta vasten suunniteltu hyödyntämään datatason rinnakkaisuutta, kun taas tehtävätason rinnakkaisuuteen ne eivät sovellu kovinkaan hyvin.
- **Ratkaistavan ongelman suuri koko**. Keskusprosessorin ja grafiikkaprosessorin välinen kommunikointi vie aikaa. Ongelman on oltava riittävän suuri, jotta sen siirtäminen grafiikkaprosessorin ratkaistavaksi on kannattavaa.
- **Korkea aritmeettinen intensiteetti**. Aritmeettisella intensiteetillä tarkoitetaan laskennan (erityisesti liukulukulaskennan) määrän suhdetta muistioperaatioiden määrään[1, s. 564]. Grafiikkaprosessorien arkkitehtuuri suosii laskentatehoa muistin latenssin kustannuksella.
- **Korkeat suorituskykyvaatimukset**. Grafiikkaprosessorilla suoritettavat ohjelmat edellyttävät poikkeuksetta korkeaa suorituskykyä — muussa tapauksessa ohjelma suoritettaisiin huomattavasti helpommin ohjelmoitavalla keskusprosessorilla.

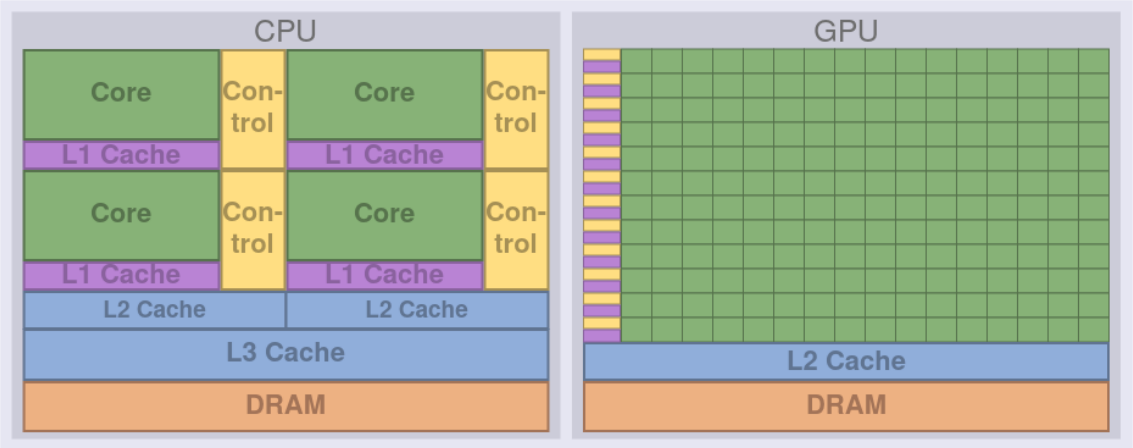
Edellä mainitut tarpeet ovat myös ohjanneet grafiikkaprosessorien arkkitehtuurin kehitystä vuosien saatossa[11, s. 23]. Luvussa 3 tutustutaan tarkemmin Nvidian Ampere-arkkitehtuuriin.

3. CUDA-LAITEARKKITEHTUURI

Nvidian GPGPU-tekniikan keskiössä ovat luonnollisesti grafiikkaprosessorit. Tässä luvussa tutustutaan CUDA-laitearkkitehtuuriin siltä osin kuin tämän työn kannalta on tarpeellista. Erityisesti tarkastellaan Nvidian uusinta grafiikkaprosessorisukupolvea, *Amperea*. Vaikka alla keskitytäänkin yksinomaan Nvidian grafiikkaprosessoreihin, monet siinä esitetyistä periaatteista pätevät laajemminkin.

3.1 Grafiikkaprosessorien erityispiirteitä

Koska grafiikkaprosessorit on suunniteltu eri tarkoitukseen kuin yleiskäyttöiset prosessorit, niillä on joukko ominaispiirteitä, joiden osalta ne eroavat yleiskäyttöisistä prosessoreista merkittävästi.



Kuva 3.1. *Grafiikkaprosessorissa laskentaelementteihin on varattu huomattavasti suurempi osa piirin transistoreista kuin yleiskäyttöisessä prosessorissa. Siinä missä yleiskäyttöisissä prosessoreissa ytimiä on korkeintaan kymmeniä, nykyisissä grafiikkaprosessoreissa niitä on tuhansittain. Perustuu lähteeseen [13].*

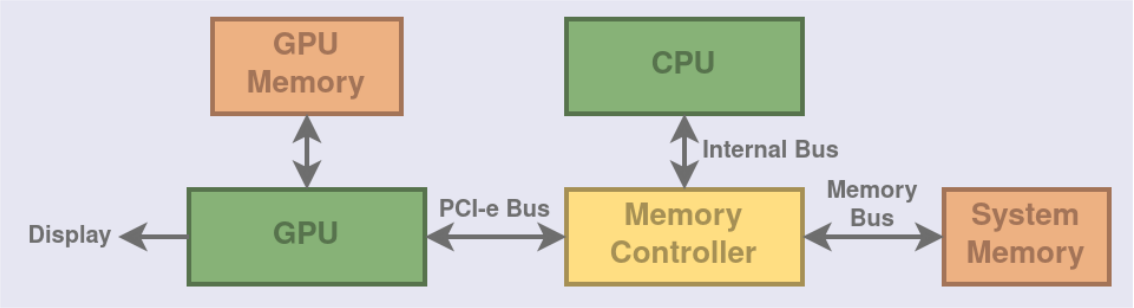
Yleiskäyttöiset prosessorit on suunniteltu suorittamaan sekventiaalisia tehtäviä mahdollisimman nopeasti, ja siksi niissä on tyypillisesti vain kourallinen säikeitä. Sekventiaalisen laskennan nopeuttamiseksi yleiskäyttöisissä prosessoreissa pyritään minimoimaan muistioperaatioiden viive, minkä takia suuri osa prosessorin pinta-alasta on omistettu eri tasoisille välimuisteille ja kontrollilogiikalle. Varsinaisten laskentaelementtien osuus prosessorin pinta-alasta on suhteellisen vaatimaton. [13]

Grafiikkaprosessorit puolestaan on suunniteltu rinnakkaiseen laskentaan: tavallisesti säi-

keitä on tuhansia[11, s. 4]. Koska massiivisella rinnakkaisuudella ja suurella muistiväylän kaistanleveydellä pystytään amortisoimaan muistioperaatioista aiheutuvat viiveet, välimuistien ja kontrollilogiikan osuus grafiikkaprosessorin pinta-alasta on verrattain pieni; valtaosa piirin pinta-alasta koostuu laskentaelementeistä. Grafiikkaprosessorin kokonaislaskentateho on siis huomattavasti korkeampi kuin vastaavan yleiskäyttöisen prosessorin. Ero prosessorien rakenteissa on havainnollistettu kuvassa 3.1. [13]

3.2 Kommunikaatiorajapinta

Grafiikkaprosessori ei yleensä ole tietokoneen ainoa prosessori, vaan se usein täydentää yleiskäyttöistä keskusprosessoria. Grafiikkaprosessori voi olla joko yleiskäyttöisestä prosessorista erillinen tai siihen integroitu. Tehokkaat grafiikkaprosessorit ovat useimmiten erillisiä. Tällöin prosessorit kommunikoivat keskenään tyypillisesti nopean PCI-e-väylän (Peripheral Component Interconnect Express) välityksellä. Keskusprosessorin ja grafiikkaprosessorin välinen kommunikointiväylä on havainnollistettu kuvassa 3.2. [11, s. 8]



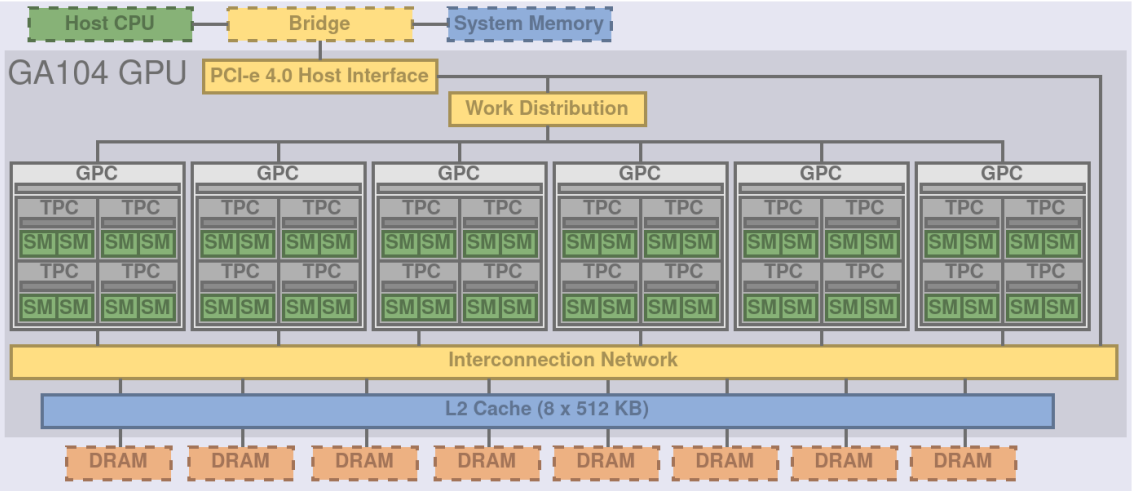
Kuva 3.2. *Keskusprosessori ja erillinen grafiikkaprosessori kommunikoivat tyypillisesti PCI-e-väylän ja keskusprosessorin muistiohjaimen välityksellä. Perustuu lähteeseen [11, s. 8].*

Halvemmissa järjestelmissä grafiikkaprosessori saattaa olla integroitu yleiskäyttöiseen prosessoriin. Tällöin grafiikkaprosessorilla ei välttämättä ole omaa muistia, vaan tietokoneen keskusmuisti toimii samalla grafiikkaprosessorin muistina. Tällaisen järjestelmän suorituskyky on verraten heikko, koska erillinen muisti on yleensä keskusmuistia nopeampaa sekä kaistanleveyden että latenssin kannalta. [11, s. 9]

Koska grafiikkaprosessori on käyttöjärjestelmän näkökulmasta ulkoinen laite, käyttäjäsovellukset kommunikoivat sen kanssa käyttöjärjestelmän ytimeen kuuluvan laiteajurin välityksellä[11, s. 9]. Ajuri muun muassa hallitsee grafiikkaprosessorin muistia ylläpitämällä sen sivutauluja[11, s. 9] ja tarjoaa toimintoja, joita hyödyntämällä käyttäjäsovellukset voivat esimerkiksi varata, lukea ja kirjoittaa grafiikkaprosessorin muistia[14]. Tavallisesti käyttäjäsovellus ei kuitenkaan kommunikoi ajurin kanssa suoraan, vaan korkeamman tason ohjelmointirajapinnan kautta[13].

3.3 Prosessorimatriisi

Tarkastellaan esimerkin vuoksi kahta Nvidian Ampere-sukupolveen kuuluvaa kuluttajatason grafiikkaprosessoria: GA102:a ja GA104:ää. Koska GA104 on näistä kahdesta hiukan yksinkertaisempi, kuvassa 3.3 on esitetty sen perusrakenne GPGPU-sovelluksen näkökulmasta. GA104:ssa on 48 moniprosessoria (streaming multiprocessor, SM). Kukin moniprosessori sisältää 128 *CUDA-ydintä*. Yhteensä CUDA-ytimiä on siis 6144. [15, s. 45–48]



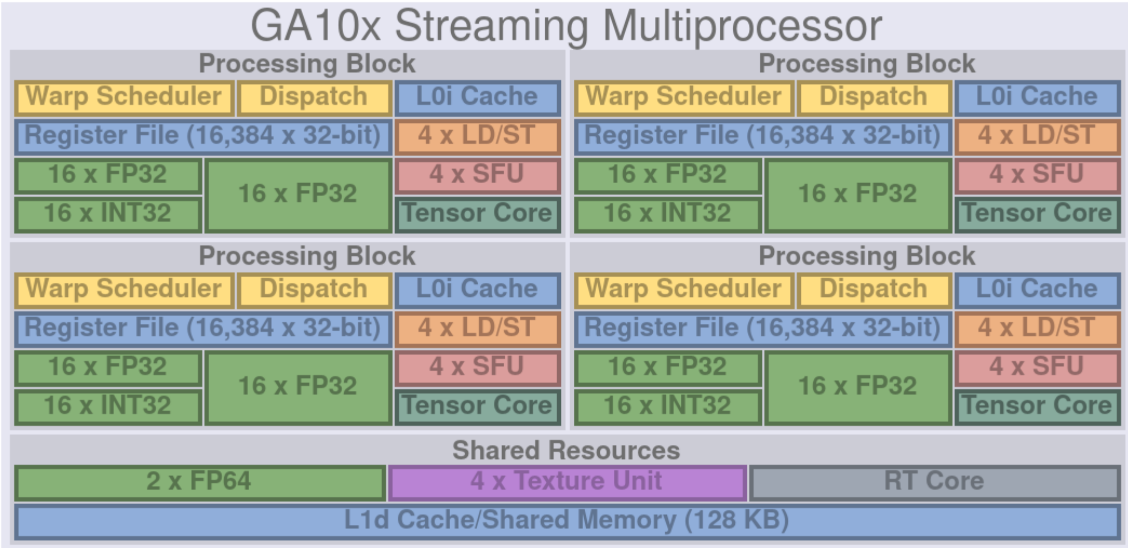
Kuva 3.3. *GA104-grafiikkaprosessorin yksinkertaistettu rakenne. Grafiikkaprosessori on useista pienemmistä moniprosessoreista rakentuva massiivisen rinnakkainen moniprosessori. Perustuu lähteisiin [11, s. 11][15, s. 46].*

Resurssien tehokkaan jakamisen vuoksi moniprosessorit on organisoitu *klustereiksi* (cluster). Tekstuuriprosessointiklusteri (texture processing cluster, TPC) muodostuu kahdesta moniprosessorista ja niiden jakamasta *PolyMorph Enginestä*. Neljä saman *rasterimoottorin* (Raster Engine) jakavaa TPC:tä puolestaan muodostaa grafiikkaprosessointiklusterin (graphics processing cluster, GPC). Klusterit yhdistyvät L2-välimuistiin ja grafiikkaprosessorin ulkoiseen muistiin koko prosessorin laajuisen yhteysverkon (interconnection network) kautta[15, s. 46][11, s. 11].

Edellä kuvatulla tavalla järjestetyt moniprosessorit yhdessä muodostavat *prosessorimatriisin* (processor array). Tällainen arkkitehtuuri on helposti skaalattavissa erilaisiin suorituskyky-ja hintaluokkiin muuttamalla prosessorien ja muistien määrää. [11, s. 11] Esimerkiksi GA104:ssa on 48 moniprosessoria ja 8 muistiosiota[15, s. 46], kun taas kalliimmassa GA102:ssä niitä on peräti 84 ja 12[15, s. 10]. GA102:ssa myös klustereiden koostumus on erilainen, sillä sen GPC:issä on 6 TPC:tä kussakin[15, s. 10].

3.4 Moniprosessoriarkkitehtuuri

Kuten edellä todettiin, grafiikkaprosessorit ovat moniprosessoreista koostuvia moniprosessoreita. Moniprosessoriarkkitehtuurin tarkoitus on tehdä resurssien jakamisesta tehokasta. Kun skalaariprosessorit sijaitsevat samassa moniprosessorissa, ne voivat jakaa keskenään muun muassa käskyvälimuistin, kontrolliyksikön ja moniprosessorin sisäisen jaetun muistin. Tämä on edullista piirin pinta-alan ja tehonkulutuksen kannalta. [11, s. 25]



Kuva 3.4. *Moniprosessorit koostuvat lukuisista laskentaytimistä, jotka jakavat keskenään joukon resursseja. Ampere-arkkitehtuurin moniprosessoreissa on lisäksi useita erikoislaskentayksiköitä. Perustuu lähteeseen [15, s. 12].*

Kuvassa 3.4 on esitetty GA10x-grafiikkaprosessorien moniprosessorin rakenne. Ampere-arkkitehtuurissa moniprosessori sisältää 4 *prosessointilohkoa* (processing block); 2 64-bittistä liukulukulaskentayksikköä (FP64); 4 tekstuuriyksikköä; 1 RT-eli säteenseurantaytimen (RT Core); sekä jaetun muistin, joka toimii samalla L1-välimuistina. [15, s. 12–13]

Kukin prosessointilohko koostuu 32-bittisistä liukuluku-(FP32) ja kokonaislukulaskentaa (INT32) suorittavista CUDA-ytimistä; jaetuista rekistereistä (register file); kontrolliyksiköstä (warp scheduler); L0-käskyvälimuistista (L0i); erikoislaskentayksiköistä, joilla voidaan laskea esimerkiksi tietokonegrafiikassa yleisiä trigonometrisiä funktioita[11, s. 35]; load/store-yksiköistä; sekä tekoälysovelluksiin tarkoitetusta *tensor*-ytimestä (Tensor Core). [15, s. 12–13]

CUDA-ytimet on jaettu kahteen rinnakkaiseen datapolkuun, joissa kussakin on 16 FP32-ydintä ja niistä toisessa lisäksi 16 INT32-ydintä. Yksittäinen prosessointilohko pystyy siis suorittamaan joko 32 FP32-operaatiota tai 16 FP32-ja 16 INT32-operaatiota kellojaksoa kohden. Koska Ampere-moniprosessori koostuu neljästä lohkosta, se pystyy suorittamaan 128–256 FP32-operaatiota per kellojakso. FP64-operaatioita se voi sen sijaan suorittaa ainoastaan 2 per kellojakso, koska FP64-ytimiä on koko moniprosessorissa vain

2. [15, s. s. 12–13]

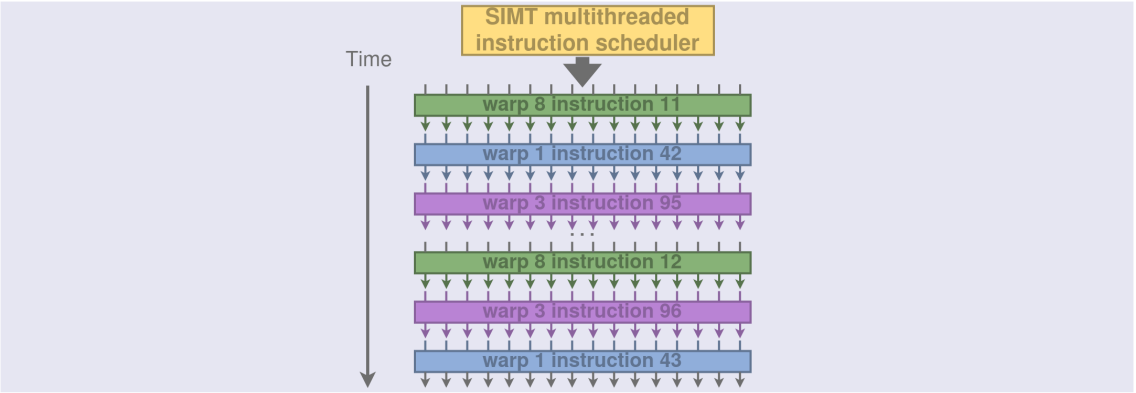
Moniprosessoriarkkitehtuurin massiivinen rinnakkaisuus palvelee grafiikkaprosessoreissa useaa tarkoitusta[11, s. 25]:

- **Muistioperaatioiden latenssin amortisointi**. Grafiikkaprosessorien välimuistit ovat verrattain pieniä, joten muistioperaatiot kestävät usein satoja kellojaksoja. Säikeiden suuri määrä mahdollistaa yhden säikeen suorittamisen toisen odottaessa muistioperaation valmistumista.
- **Fyysisten prosessorien lukumäärän virtualisointi**. Ohjelmien skaalaaminen helpottuu, kun ohjelmoijan ei tarvitse välittää fyysisten prosessorien lukumäärästä.
- **Hienojakoisen rinnakkaisen ohjelmoinnin tukeminen**. Kun jokaisella säikeellä on omat rekisterinsä ja suorituskontekstinsa, ne ovat toisistaan riippumattomia.
- **Ohjelmoinnin yksinkertaistaminen**. Rinnakkaisen ohjelman sijaan ohjelmoija voi kirjoittaa yhden *sekventiaalisen* ohjelman, jonka yksittäinen säie suorittaa toisista riippumatta (SPMD-ohjelmointimalli).

Edellä mainitut seikat liittyvät vahvasti grafiikkaprosessorien ohjelmointiin, ja ne tulevatkin uudelleen vastaan CUDA-ohjelmoinnin tarkastelun yhteydessä luvussa 4.

3.5 SIMT-arkkitehtuuri

Grafiikkaprosessorien moniprosessorit hallitsevat ja suorittavat satoja säikeitä. Tämän mahdollistamiseksi niissä hyödynnetään SIMT-arkkitehtuuria. SIMT muistuttaa SIMD-arkkitehtuuria, mutta siinä monta säiettä suorittaa saman käskyn samanaikaisesti, kun taas SIMD:ssä yksi säie kohdistaa yhden käskyn moneen data-alkioon samanaikaisesti. Tällainen arkkitehtuuri on hyvin tehokas, koska kaikki säikeet suorittavat saman käskyn ja voivat siten jakaa saman kontrollilohkon keskenään. [11, s. 28–29]



Kuva 3.5. *Moniprosessorin SIMT-vuorontaja valitsee yhden 32:sta rinnakkaisesta säikeestä koostuvan warpin kerrallaan suoritukseen. Warpit ovat toisistaan riippumattomia, joten vuorontaja voi koska tahansa vaihtaa suoritusvuorossa olevaa warpia. Perustuu lähteeseen [11, s. 28].*

Ampere-arkkitehtuurissa jokainen prosessointilohko sisältää *SIMT-vuorontajan*, joka luo, hallitsee, vuorontaa ja suorittaa säikeitä 32:sta rinnakkaisesta säikeestä koostuvina *warpeina*[13]. Samaan warpiin kuuluvat säikeet suorittavat samaa ohjelmaa ja aloittavat sen suorittamisen samasta osoitteesta. Joka kerta kun uuden käskyn suoritus on alkamassa, vuorontaja valitsee jonkin warpin suoritukseen ja määrittää, minkä käskyn warpin säikeet seuraavaksi suorittavat. Vain yksi warp on kerrallaan suorituksessa. [11, s. 28–29] Warpien vuoronnus on havainnollistettu kuvassa 3.5.

Vaikka samaan warpiin kuuluvat säikeet aloittavatkin ohjelman suorittamisen samasta osoitteesta, ne voivat haarautua erilleen esimerkiksi if–else-haarassa. Tällaisessa tilanteessa säikeiden on voitava suorittaa eri käskyjä. SIMT-vuorontaja seuraa säikeiden haarautumisen tilaa ja määrää, mitkä säikeet ovat kullakin käskyllä aktiivisia — ts. mitkä säikeet suorittavat kyseisen käskyn. Epäaktiiviset säikeet eivät tällä aikaa tee mitään. Paras mahdollinen tehokkuus SIMT-prosessorilla saavutetaan kaikkien säikeiden suorittaessa samaa käskyä samanaikaisesti. [11, s. 28–29]

3.6 Muistiarkkitehtuuri

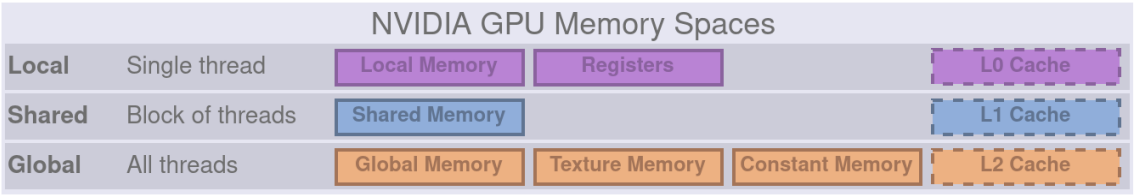
Grafiikkaprosessoreilla suoritettava laskenta on usein hyvin muisti-intensiivistä: käsiteltävän datan määrä on suuri ja sitä on voitava siirrellä nopeasti. Massiivisen rinnakkainen prosessointi edellyttää massiivisen rinnakkaista muistia, ja muistin ominaisuuksilla onkin suuri merkitys prosessorin suorituskyvyn kannalta. Tästä syystä grafiikkaprosessorien muistijärjestelmillä on yleisesti seuraavat ominaisuudet[11, s. 36–37]:

- **Suuri muistiväylän leveys**. Kuten edelläkin todettiin, grafiikkaprosessorien muisti on jaettu useisiin osioihin joita hallitsevat eri muistiohjaimet. Esimerkiksi GA104-grafiikkaprosessorissa 8 muistia, kukin 32-bittinen, muodostavat yhdessä väylän, jonka leveys on 256 bittiä.
- **Suuri signalointinopeus**. Parhaan mahdollisen suorituskyvyn saavuttamiseksi grafiikkaprosessoreissa hyödynnetään aggressiivisia signalointitekniikoita, joiden avulla data saadaan liikkumaan johdoissa mahdollisimman nopeasti. Esimerkiksi GA102-grafiikkaprosessorin GDDR6X SDRAM -muistijärjestelmässä käytetään PAM4-signalointia, jossa amplitudimodulaation avulla kutakin johtoa pitkin lähetetään 4 bittiä per kellojakso — 2 bittiä kellosignaalin laskevalla ja 2 nousevalla reunalla[15, s. 30].
- **Läpisyötön priorisointi**. Jotta suurten datamäärien liikuttelu olisi mahdollisimman tehokasta, grafiikkaprosessorien muistit keskittyvät maksimoimaan läpisyötön (throughput), eivät niinkään minimoimaan latenssia.
- **Datan kompressointi**. Muistijärjestelmissä hyödynnetään sekä häviöllistä että häviötöntä kompressointia. Ohjelmoijan tulee tiedostaa, missä tilanteessa kompressointi on häviöllistä, ja huomioida sen vaikutus ohjelman toimintaan.

- **Muistiliikenteen minimointi**. Välimuisteilla ja muistioperaatioiden synkronoinnilla pyritään minimoimaan prosessoripiirin ulkopuolelle suuntautuvien muistioperaatioiden määrä.

Grafiikkaprosessorien muistijärjestelmä eroaa yleiskäyttöisen prosessorin muistijärjestelmästä hyvin oleellisesti siten, että muistiresursseja on varattu selkeästi eri tarkoituksiin. Nvidian grafiikkaprosessorien muistijärjestelmä voidaan jakaa seuraaviin osiin[11, s. 38–50][13]:

- **Lokaali muisti** (local memory). Jokaisella säikeellä on pieni määrä omaa muistia, johon muilla säikeillä ei ole pääsyä.
- **Jaettu muisti** (shared memory). Samaan moniprosessoriin kuuluvat säikeet jakavat moniprosessorin muistin, jonka avulla säikeet voivat kommunikoida keskenään.
- **Globaali muisti** (global memory). Kaikilla säikeillä on pääsy globaaliin muistiin, johon ne voivat kohdistaa sekä luku-että kirjoitusoperaatiota.
- **Vakiomuisti** (constant memory). Muuttumatonta dataa voidaan tallentaa vakiomuistiin, johon kaikilla säikeillä on pääsy. Tämä on hyödyllistä, koska vakiodatasta ei ole tarpeen tehdä lokaaleja kopioita ja kaikki vakiot voidaan tarvittaessa tallentaa yhteen ja samaan muistiin. Säikeet eivät voi kirjoittaa vakiomuistiin[13].
- **Tekstuurimuisti** (texture memory). Tekstuureilla on erinäisiä ominaisuuksia, joiden takia ne tallennetaan erilliseen tekstuurimuistiin. Kaikilla säikeillä on pääsy tähän muistiin. Säikeet eivät voi kirjoittaa tekstuurimuistiin[13].



Kuva 3.6. *Grafiikkaprosessorin muistijärjestelmä on hierarkkinen. Muistiavaruuksia on kolme: lokaali, jaettu ja globaali. Perustuu lähteisiin [11, s. 36–37][13].*

Edellä mainitut muistit on jaettu kolmeen muistiavaruuteen. Ensimmäinen on lokaali muistiavaruus, jonka muodostaa lokaali muisti. Jaettu muisti muodostaa niinikään oman muistiavaruutensa. Globaali muistiavaruus sen sijaan sisältää sekä globaalin muistin että vakio-ja tekstuurimuistin. [13] Muistiavaruuksien hierarkia ja eri muistien sijoittuminen niihin on tiivistetty kuvaan 3.6.

Nykyaikaisten grafiikkaprosessorien muistijärjestelmät ovat virtuaalisia. Sivutauluja ylläpitää laiteajuri. [11, s. 9]

4. CUDA-OHJELMOINTIMALLI

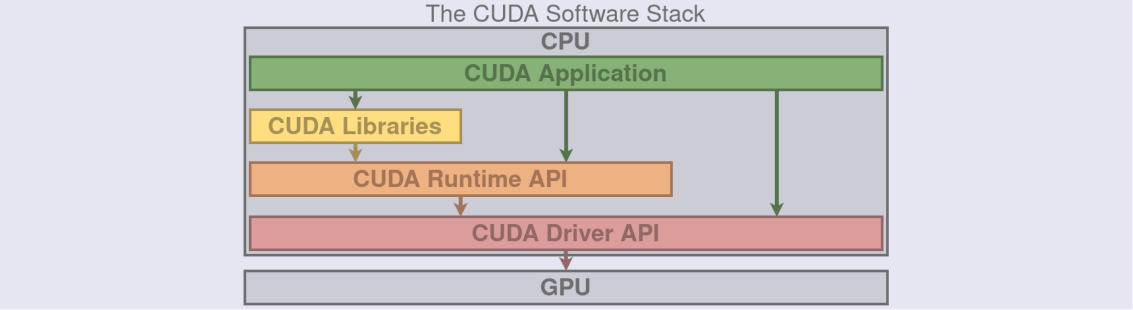
CUDA on Nvidian kehittämä suljetun lähdekoodin GPGPU-ohjelmointimalli (programming model) ja -rajapinta (application programming interface, API). Sen tarkoitus on tehdä rinnakkaisesta ohjelmoinnista helposti lähestyttävää. CUDA-laitekoodi kirjoitetaan GPGPU-laskennan tarpeisiin mukautetulla C++-ohjelmointikielen versiolla (CUDA C++), joka ei suuresti eroa standardin mukaisesta C++-kielestä. CUDA-rajapintaa voi hyödyntää myös muista ohjelmointikielistä, kuten esimerkiksi Fortranista, käsin. [13]

CUDA-ohjelmointimalli perustuu kolmeen avainabstraktioon: säiejoukkojen hierarkiaan, jaettuun muistiin ja barrier-synkronointiin. Nämä abstraktiot on toteutettu CUDA C++:ssa minimaalisilla standardinmukaiseen C++:aan tehdyillä lisäyksillä. [13] Tässä luvussa esitetään CUDA-ohjelmointimalli työn kannalta tarpeellisella tarkkuudella erityisesti mainittuihin abstraktioihin keskittyen. Kohdassa 4.7 sivutaan lyhyesti myös muita GPGPU-rajapintoja.

4.1 Rajapinnat ja kirjastot

CUDA tarjoaa ohjelmoijalle kaksi ohjelmointirajapintaa: CUDA Runtime APIn ja CUDA Driver APIn. [13] Niiden oleellinen ero on abstraktiotaso: CUDA Driver API on selvästi matalamman tason rajapinta kuin CUDA Runtime API. Ohjelmoija voi valita rajapinnan tarpeidensa mukaan: Runtime API on huomattavasti helppokäyttöisempi, mutta Driver API tarjoaa matalan tason ominaisuuksia, jotka Runtime API puolestaan piilottaa ohjelmoijalta. [16] Esimerkkinä Driver APIn tarjoamista mahdollisuuksista voidaan mainita matalan tason virtuaalimuistinhallintarajapinta, jonka avulla ohjelman suorituskykyä voi optimoida huomattavasti pidemmälle kuin Runtime APIa käyttämällä on mahdollista[17]. Ohjelmoija voi myös halutessaan käyttää kumpaakin rajapintaa samanaikaisesti, mutta pelkkä Runtime API riittää useimpien sovellusten tarpeisiin[13].

CUDA Runtime APIn päälle on lisäksi rakennettu joukko kirjastoja, jotka tarjoavat sovellusten usein tarvitsemia toimintoja. Näitä ovat esimerkiksi lineaarialgebrakirjasto cuBLAS, Fourier-muunnoskirjasto cuFFT sekä hiukan C++:n STL-kirjastoa muistuttava rinnakkainen algoritmikirjasto Thrust[18]. Kirjastojen ja rajapintojen suhde käyttäjäsovellukseen ja toisiinsa on esitetty kuvassa 4.1.



Kuva 4.1. *CUDA-sovellus voi kommunikoida grafiikkaprosessorin kanssa CUDA-kirjastojen, CUDA Runtime APIn ja CUDA Driver APIn avulla. Perustuu lähteisiin [13][18].*

4.2 Heterogeenisyys

CUDA-ohjelmat ovat heterogeenisiä: osan koodista suorittaa *isäntä* (host) ja osan *laite* (device). Käytännössä isäntä on tietokoneen keskusprosessori ja laite CUDA-yhteensopiva Nvidian grafiikkaprosessori. Isännällä ja laitteella on omat erilliset muistiavaruutensa. Kommunikointi muistiavaruudesta toiseen tapahtuu CUDA-rajapinnan funktioiden, kuten esimerkiksi `cudaMemcpy`:n, avulla. [13]

CUDA määrittelee joukon erityisiä *suoritusavaruusmääreitä* (execution space specifier), joiden avulla kääntäjälle kerrotaan, mikä laite kutakin funktiota voi kutsua. Avainsana `__device__` merkitsee funktion, jonka laite suorittaa ja jota vain laite voi kutsua; `__host__` funktion, jonka isäntä suorittaa ja jota vain isäntä voi kutsua; ja `__global__` funktion, jonka laite suorittaa ja jota sekä isäntä että laite voivat kutsua. [13]

4.3 Kernelit

CUDA-laitekoodi koostuu pääasiassa laskentaytimistä eli *kerneleistä*. Kernel on CUDA C++ -funktio, jota isäntä kutsuu mutta jonka laite suorittaa. Toisin kuin tavalliset C++-funktiot, kernelit suoritetaan aina rinnakkaisesti; isäntä määrittelee kerneliä kutsuessaan, kuinka monessa säikeessä se suoritetaan. Kaikki rinnakkaiset kernel-instanssit vastaanottavat samat, isännän määrittelemät funktiokutsuparametrit. [13]

```
1  // A CUDA device kernel that adds two vectors.
2  __global__ void vectorAdd(float* A, float* B, float* C)
3  {
4      // The thread index dictactes which components of the
5      // input vectors are added by this thread.
6      int i = threadIdx.x;
7      C[i] = A[i] + B[i];
8  }
9
10 int main()
11 {
12     ...
13     // Invoke the kernel with a 1-dimensional block of N threads.
14     vectorAdd<<<1, N>>>(A, B, C);
```

```
15     ...
16 }
```

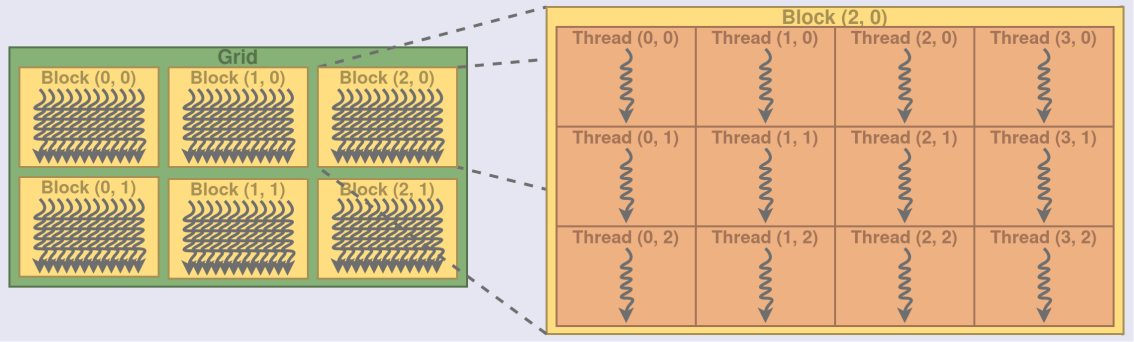
Ohjelma 4.1. *Esimerkki CUDA-kernelin* `vectorAdd` *kutsumisesta isäntäkoodista. Muokattu lähteestä [13].*

Ohjelmassa 4.1 on esimerkki CUDA-kernelistä `vectorAdd`, joka laskee yhteen N-ulotteiset vektorit `A` ja `B` ja tallettaa tuloksen summavektoriin `C`. Kukin kernel-instanssi eli säie laskee yhden summavektorin komponentin. Kun tämän kernelin suorittaa N säiettä — yksi kutakin alkiota kohden — lopputulokseksi saadaan kokonainen summavektori.

4.4 Säiehierarkia

Koska kaikki säikeet vastaanottavat kernel-kutsun yhteydessä samat parametrit, parametreilla ei voida suoraan ohjata yksittäisen säikeen toimintaa. Säikeellä on kuitenkin oltava keino päätellä roolinsa kernelin toteuttamassa laskutoimituksessa. Tätä varten jokaiselle säikeelle annetaan uniikki *tunniste* (thread ID), jonka se saa tietoonsa CUDA C++:n sisäänrakennettujen muuttujien avulla. Ohjelmoija määrittelee kunkin säikeen toiminnan koodissa säietunnisteen perusteella. [13]

Tunnisteen lisäksi kullakin säikeellä on indeksi. Siinä missä tunniste on lineaarinen, indeksi on 3-ulotteinen vektori (x, y, z). Tämä on käytännöllistä, koska erilaiset laskutoimitukset voivat tapahtua eri ulottuvuuksissa ja usein on luontevaa organisoida säikeet samaan ulottuvuuteen kuin ratkottava ongelmakin. Esimerkiksi matriisien yhteenlasku on 2-ulotteinen operaatio, mutta tilavuuteen liittyvät laskut ovat 3-ulotteisia. Indeksin ulottuvuuden määrää ohjelmoija kernel-kutsun yhteydessä, jossa säikeiden lukumäärän lisäksi määritellään 1-, 2-tai 3-ulotteiset *lohkot* (thread block), joihin säikeet jaetaan. [13] Kuvassa 4.2 on havainnollistettu 2-ulotteisen säielohkon rakenne.



Kuva 4.2. *Tässä esimerkissä 2-ulotteinen lohkoruudukko on jaettu 2-ulotteisiin säielohkoihin. Kukin lohko koostuu joukosta yhteen niputettuja säikeitä. Perustuu lähteeseen [13].*

Tunnisteen ja indeksin välinen yhteys on suoraviivainen[13]:

- Jos lohko on 1-ulotteinen ja säikeen indeksi on x, säikeen tunniste on myös x.

- Jos lohko on 2-ulotteinen ja sen dimensiot ovat (D_x, D_y) ja säikeen indeksi on (x, y), säikeen tunniste on $x + yD_x$.
- Jos lohko on 3-ulotteinen ja sen dimensiot ovat (D_x, D_y, D_z) ja säikeen indeksi on (x, y, z), säikeen tunniste on $x + yD_x + zD_xD_y$.

Tunnistetta voi siis ajatella säikeen linearisoituna indeksinä. Ohjelman 4.1 esimerkissä indeksi on 1-ulotteinen, joten säikeen tunniste on sama kuin sen indeksi eikä sitä tarvitse varsinaisesti laskea, vaan se voidaan lukea `threadIdx`-vektorin `x`-komponentista. Luettuaan tunnisteensa säie käyttää sitä indeksinä vektoreihin `A`, `B` ja `C`.

Kaikki saman lohkoon kuuluvat säikeet sijaitsevat samassa fyysisessä prosessoriytimessä. Tämän takia säikeiden maksimilukumäärä lohkoa kohden on rajattu; sen määrää grafiikkaprosessorin *compute capability* (ks. kohta 4.6). Kaikissa tähänastisissa grafiikkaprosessoreissa lohkon maksimikoko on kuitenkin sama, 1024 säiettä. [13]

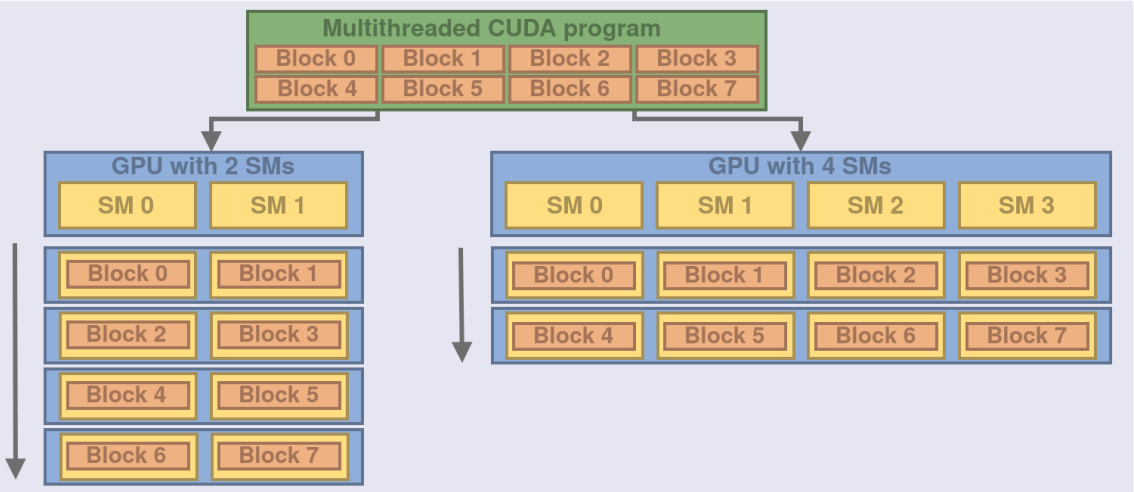
Lohkojen rajallisesta koosta johtuen kerneleitä on usein tarpeellista suorittaa rinnakkain useilla, keskenään samankokoisilla lohkoilla. Samaan tapaan kuin säikeet, lohkot on jaettu 1-, 2-tai 3-ulotteiseen *ruudukkoon* (grid). Ruudukon ulottuvuuden ei tarvitse olla sama kuin sen sisältämien lohkojen. Ruudukon koon määrää ratkaistavan ongelman koko, joka on usein suurempi kuin yksittäisen lohkon maksimikoko. [13] Kuvassa 4.2 on havainnollistettu 2-ulotteisen ruudukon rakenne.

Samoin kuin säikeillä, myös lohkoilla on 1-, 2-tai 3-ulotteinen indeksi, joka kuitenkin säikeen indeksistä poiketen on uniikki. Säie voi lukea lohkon indeksin `blockIdx`-muuttujasta ja koon `blockDim`-muuttujasta. Ohjelmassa 4.2 on esimerkki, jossa säikeet on hiukan yksinkertaistetusti jaettu useisiin eri lohkoihin. Tyypillinen lohkon koko on tässäkin esimerkissä käytetty 16 x 16. [13]

```
1  // A CUDA device kernel that adds two matrices.
2  __global__ void matrixAdd(float A[N][N], float B[N][N], float C[N][N])
3  {
4      int i = blockIdx.x * blockDim.x + threadIdx.x;
5      int j = blockIdx.y * blockDim.y + threadIdx.y;
6      if (i < N && j < N)
7          C[i][j] = A[i][j] + B[i][j];
8  }
9
10 int main()
11 {
12     ...
13     // Invoke the kernel by organizing the threads in
14     // 2-dimensional blocks of 16 x 16 threads.
15     dim3 threadsPerBlock(16, 16);
16     dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
17     matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
18     ...
19 }
```

Ohjelma 4.2. *Tässä esimerkissä `matrixAdd`-kerneliä kutsuvat säikeet on jaettu useisiin lohkoihin, joissa on 16 x 16 säiettä kussakin. Muokattu lähteestä [13].*

CUDA-ohjelmointimalli edellyttää, että lohkot voidaan suorittaa toisistaan riippumattomasti. Tämä antaa vuorontajalle (ks. kohta 3.5) mahdollisuuden vuorontaa lohkot missä tahansa järjestyksessä mille tahansa fyysiselle prosessorille. CUDA-ohjelma skaalautuu siis näiltä osin automaattisesti: ohjelmoija voi lisätä lohkojen määrää muuttamatta koodia muilta osin lainkaan. [13]



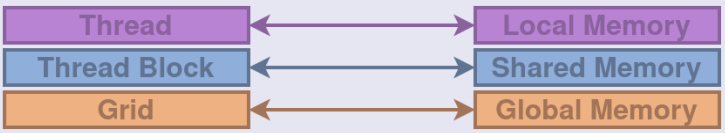
Kuva 4.3. *CUDA-ajoympäristö huolehtii laskennan jakamisesta fyysisten laskentaresurssien kesken. Näin CUDA-ohjelmat skaalautuvat joustavasti fyysisestä laitteistosta riippumatta. Perustuu lähteeseen [13].*

Samaan lohkoon kuuluvat säikeet voivat tarvittaessa tehdä yhteistyötä kahdella tavalla: ne voivat jakaa dataa jaetun muistin kautta (ks. kohta 4.5), minkä lisäksi ne voivat synkronoitua keskenään CUDAn sisäänrakennetun barrier-synkronoinnin avulla. Yksi synkronoinnin eduista on sen tarjoama mahdollisuus koordinoida muistioperaatioita siten, että usea säie lukee tai kirjoittaa muistiin samanaikaisesti, mikä voi parantaa ohjelman suorituskykyä huomattavasti. Synkronointi tapahtuu kernelissä `__syncthreads()`-funktion avulla: se muodostaa esteen (barrier), jolle saavuttuaan säikeet voivat jatkaa suoritustaan vasta kaikkien saman lohkon säikeiden saavutettua sen. [13]

Jotta säikeiden yhteistyö olisi tehokasta, sekä datan jakamisen että synkronoinnin on niinikään oltava tehokasta. Tästä syystä CUDA-ohjelmointimalli edellyttää jaetun muistin olevan pienilatenssinen muisti lähellä prosessoriydintä ja `__syncthreads()`-funktion olevan suorituskyvyn näkökulmasta hyvin kevyt. [13]

4.5 Muistihierarkia

CUDA-ohjelmointimallin muistihierarkia vastaa täysin kohdassa 3.6 esitettyä Nvidian grafiikkaprosessorin muistiarkkitehtuuria. Muisti on siis jaettu lokaaliin, jaettuun, globaaliin, vakio-ja tekstuurimuistiin kohdassa 3.6 esitetyllä tavalla. Lokaali muisti näkyy ainoastaan sen omistavalle säikeelle; jaettu muisti samaan lohkoon kuuluville säikeille; ja globaali, vakio-ja tekstuurimuisti kaikkien lohkojen kaikille säikeille. [13]



Kuva 4.4. *CUDA-ohjelmointimallin määrittelemä muistihierarkia on yhteneväinen kohdassa 3.6 esitetyn laitteistotason muistiarkkitehtuurin kanssa. Perustuu lähteeseen [13].*

Muuttujia voi CUDA-koodissa sijoittaa eri muistiavaruuksiin hiukan samaan tapaan kuin funktioita suoritusavaruuksiin. Tämä tapahtuu muistiavaruusmääreiden `__device__`, `__managed__`, `__constant__`, `__shared__` ja `__restrict__` avulla. Muuttujien sijoittelu muistiavaruuksiin on kuitenkin huomattavasti kontekstisidonnaisempaa kuin funktioiden sijoittelu suoritusavaruuksiin. Tästä syystä globaalille, lokaalille ja tekstuurimuistille ei ole omia eksplisiittisiä määreitä, vaan sijoittelu riippuu edellä mainittujen määreiden lisäksi kontekstista — esimerkiksi kernelin paikalliset muuttujat sijoitetaan automaattisesti säikeen lokaaliin muistiin. Lisäksi tekstuurimuistin käsittely on mahdollista ainoastaan erityisen ohjelmointirajapinnan kautta. [13]

Lokaali ja jaettu muisti ovat säikeiden näkökulmasta haihtuvia eikä niiden sisällön siis voi olettaa säilyvän samana kernel-kutsusta toiseen. Globaalit muistit — globaali, vakio-ja tekstuurimuisti — sen sijaan säilyttävät tilansa kernel-kutsujen välillä. [13]

4.6 Compute capability -versiointi

Jokaisella CUDA-yhteensopivalla laitteella on kaksiosainen compute capability -versionumero — esimerkiksi 7.5 (Turing) tai 8.6 (Ampere). Compute capability kertoo, mitä ominaisuuksia laite tukee. Laitteet, joiden compute capabilityjen ensimmäiset numerot (major version) ovat samat, kuuluvat samaan laitesukupolveen. Toinen numero (minor version) sen sijaan vaihtelee myös laitesukupolven sisällä. Saman sukupolven laitteiden väliset erot ovat verrattain pieniä, kun taas eri sukupolvien välillä voi olla hyvin huomattaviakin eroja. [13]

CUDA-ohjelmointirajapinta mahdollistaa laitteen compute capabilityn selvittämisen ajonaikaisesti. Näin CUDA-sovellus voi säätää toimintaansa laitteen tukemien ominaisuuksien perusteella. Ohjelmoijan on tärkeää huomioida eri laitteiden ominaisuudet, sillä compute capability vaikuttaa laitteen suorituskykyyn ja toimintaan monin tavoin. Esimerkiksi 16-bittisiä liukulukuoperaatioita tuetaan vasta compute capabilitystä 5.3 (Maxwell) alkaen. [13]

Compute capabilityn roolia CUDA-ohjelman käännösprosessissa käsitellään tarkemmin kohdassa 5.2.

4.7 Vaihtoehtoisia rajapintoja

CUDAn lisäksi on olemassa joukko muita GPGPU-ohjelmointirajapintoja. Eräitä näistä on listattu taulukossa 4.1. Mainituista teknologioista vain CUDA on puhtaasti GPGPU-rajapinta; C++ AMP, OpenACC, OpenCL ja OpenMP ovat yleisluontoisempia rinnakkaisen ohjelmoinnin rajapintoja; DirectX-rajapintakokoelmaan lukeutuva DirectCompute sekä OpenGL ja Vulkan ovat grafiikkarajapintoja, jotka mahdollistavat GPGPU-laskennan laskentavarjostimien (compute shader) avulla.

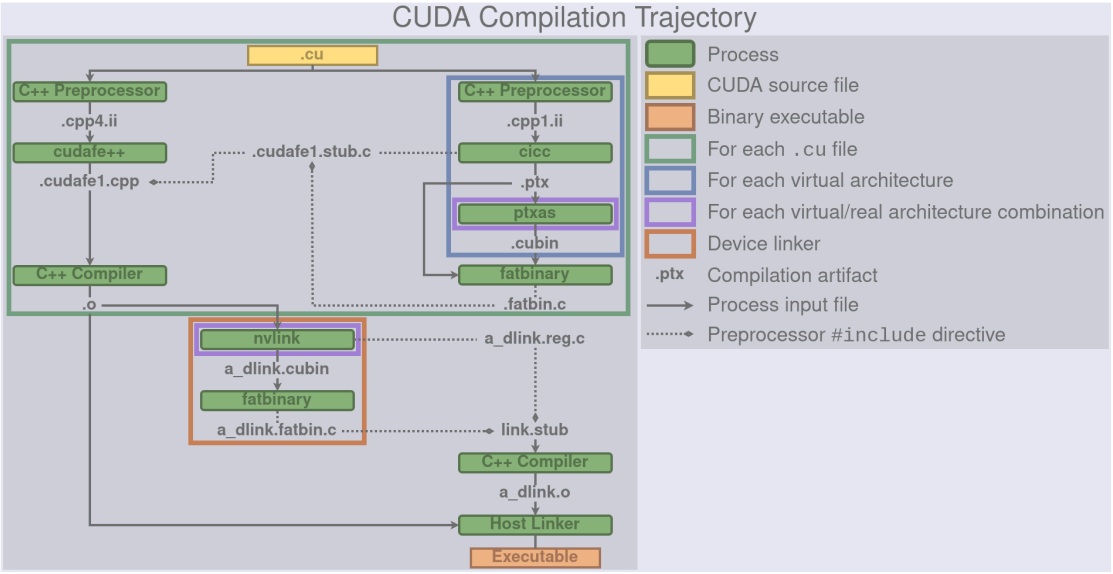
Rajapinta	Kehittäjä	Alusta
C++ AMP[19]	Microsoft	Windows
CUDA[13]	Nvidia	Alustariippumaton
DirectCompute[20]	Microsoft	Windows
OpenACC[21]	Cray, CAPS, Nvidia, PGI	Alustariippumaton
OpenCL[22]	Khronos Group, Inc.	Alustariippumaton
OpenGL[23]	Khronos Group, Inc.	Alustariippumaton
OpenMP[24]	OpenMP Architecture Review Board	Alustariippumaton
Vulkan[25]	Khronos Group, Inc.	Alustariippumaton

Taulukko 4.1. *GPGPU-ohjelmointirajapintoja.*

Lisäksi on olemassa joukko muita GPGPU-laskennan hyödyntämisen mahdollistavia rajapintoja ja työkaluja. Esimerkkeinä voidaan mainita MATLAB-ohjelmiston CUDA-ohjelmointirajapinta[26]; AMD:n HIP-kääntäjä (Heterogeneous Computing Interface for Portability) ja hipify-työkalu, jolla CUDA-koodia voidaan kääntää alustariippumattomaksi HIP C++ -koodiksi[27]; ja AMD:n kehittämä GPUFORT-työkalu, joka kääntää Fortran-pohjaista CUDA-koodia Fortran-pohjaiseksi OpenMP-tai HIP-koodiksi[28].

5. CUDA-OHJELMAN KÄÄNNÖSPROSESSI

Heterogeenisen CUDA-ohjelman kääntäminen on huomattavasti monimutkaisempaa kuin tavallisen, homogeenisen C++-ohjelman. Koska CUDAn tekninen toteutus tukeutuu vahvasti kääntäjäteknisiin ratkaisuihin, tässä luvussa luodaan yleiskatsaus CUDA-ohjelmien kääntämiseen.



Kuva 5.1. *CUDA-ohjelman käännösprosessi on monimutkainen ja sisältää monta vaihetta. NVCC-kääntäjäajurin tehtävä on piilottaa prosessin kompleksisuus ohjelmoijalta. Perustuu lähteeseen [29].*

CUDA-ohjelmat käännetään Nvidian omalla LLVM-pohjaisella[30] NVCC-kääntäjällä (Nvidia CUDA Compiler). NVCC on kääntäjäajuri (compiler driver), joka ajaa joukkoa muita käännöstyökaluja; käännösprosessin monimutkaisuus piilotetaan näin ohjelmoijalta[29]. Käännösprosessin yksityiskohtaisia vaiheita ei käsitellä tässä työssä tarkemmin, mutta havainnollistuksen vuoksi ne on tiivistetty kuvaan 5.1.

5.1 Yhteensopivuushaasteet

Nvidian grafiikkaprosessorit julkaistaan sukupolvittain. Eri sukupolvien grafiikkaprosessorien toiminnallisuudessa, arkkitehtuurissa ja suorituskyvyssä on huomattavia eroja; saman sukupolven prosessorien väliset erot ovat selvästi pienempiä. Erojen seurauksena binääriyhteensopivuus sukupolvesta toiseen ei ole lainkaan taattua, joskin mahdollista: esimerkiksi Maxwell-arkkitehtuurin grafiikkaprosessorit julkaistiin kolmessa sukupolves-

sa, jotka ovat keskenään taaksepäin binääriyhteensopivia — ts. 1. sukupolven Maxwell-prosessorille käännetty CUDA-koodi on yhteensopiva 2. ja 3. sukupolven Maxwell-prosessorien kanssa. [29]

Binääriyhteensopivuus aiheuttaa kääntäjätekniikan kannalta kaksi merkittävää ongelmaa [29].

1. Kuinka varmistetaan, että CUDA-ohjelma toimii edelleen käyttäjän vaihdettua grafiikkaprosessorinsa uudempaan?
2. Kuinka varmistetaan, että käyttäjä saa CUDA-ohjelmaa käyttäessään täyden hyödyn uudesta grafiikkaprosessoristaan?

Sekä käyttäjä-että kehittäjäkokemuksen kannalta olisi suotavaa, ettei käyttäjän tarvitsisi kääntää tai asentaa ohjelmaa uudelleen eikä osata valita ohjelmabinäärien joukosta juuri oman laitteensa kanssa yhteensopivaa binääriä.

5.2 Virtuaaliarkkitehtuurit

NVCC määrittelee joukon virtuaalisia kohdearkkitehtuureja, joista kukin vastaa tiettyä compute capability -versiota (ks. kohta 4.6). Esimerkiksi `compute_50`-virtuaaliarkkitehtuuri vastaa compute capability -versiota 5.0. Todelliset arkkitehtuurit ovat yhden tai useamman virtuaaliarkkitehtuurin *toteutuksia*. Käytännöllisesti katsoen virtuaaliarkkitehtuuri määrittelee, millaisia ominaisuuksia sen toteuttavan todellisen arkkitehtuurin on tuettava. Todellisten ja virtuaalisten arkkitehtuurien yhteys on esitetty taulukossa 5.1.

Arkkitehtuuri	**Virtuaaliarkkitehtuuri**	**Ominaisuudet**
sm_35, sm_37	compute_35, compute_37	Perusominaisuudet + Kepler-tuki + *Unified memory* + *Dynamic parallelism*
sm_50, sm_52, sm_53	compute_50, compute_52, compute_53	+ Maxwell-tuki
sm_60, sm_61, sm_62	compute_60, compute_61, compute_62	+ Pascal-tuki
sm_70, sm_72	compute_70, compute_72	+ Volta-tuki
sm_75	compute_75	+ Turing-tuki
sm_80, sm_86, sm_87	compute_80, compute_86, compute_87	+ Ampere-tuki

Taulukko 5.1. *Jokaista compute capability -versiota vastaa yksi virtuaalinen ja yksi todellinen arkkitehtuuri. Todellinen arkkitehtuuri voi kuitenkin olla yhden tai useamman virtuaaliarkkitehtuurin toteutus. Lähteestä [29].*

CUDA-ohjelmaa käännettäessä valitaan joukko virtuaaliarkkitehtuureja ja todellisia arkkitehtuureja, joille ohjelma tulee kääntää. Valitsemalla mahdollisimman *pieni* virtuaaliarkkitehtuuri ohjelmasta saadaan yhteensopiva mahdollisimman monen todellisen arkkiteh-

tuurin kanssa. Toisaalta valitsemalla mahdollisimman uusi todellinen arkkitehtuuri ohjelman suorituskyvystä saadaan paras mahdollinen, koska uudet arkkitehtuurit tuovat mukanaan uusia suorituskykyä parantavia ominaisuuksia. [29]

5.3 PTX-virtuaalikone

PTX (Parallel Thread Execution) kehittämä massiivisen rinnakkainen virtuaalikone (virtual machine) ja -käskykanta (instruction set architecture, ISA)[31], jonka voi ajatella edustavan geneeristä Nvidian grafiikkaprosessoria. Koska todellisten grafiikkaprosessorien arkkitehtuuri muuttuu verraten nopeasti, PTX:n tarkoitus on tarjota kääntäjille stabiili kohdearkkitehtuuri[11, s. 31].

```
1         .reg     .b32  r1, r2;         // Declare 32-bit registers r1 and r2
2         .global  .f32  array[N];       // Declare float array in global memory
3
4 start:  mov.b32        r1, %tid.x;     // Load thread ID into r1
5         shl.b32        r1, r1, 2;      // Left shift r1 by 2 bits
6         ld.global.b32  r2, array[r1];  // Load array element into r2
7         add.f32        r2, r2, 0.5;    // Add 0.5 to r2
```

Ohjelma 5.1. *Esimerkki PTX-assemblykoodista. Koodin suorittava säie lataa globaalissa muistissa sijaitsevan taulukon yksittäisen alkion rekisteriin ja lisää siihen luvun 0.5. Lähteestä [31].*

PTX-assemblykoodi on tekstimuotoista ja muistuttaa kovasti x86-assemblykieltä, kuten ohjelman 5.1 esimerkistä nähdään. NVCC käyttää PTX-assemblykieltä käännösprosessin välikielenä[29] (intermediate language) hiukan samaan tapaan kuin esimerkiksi Java JVM-tavukoodia.

5.4 Kaksivaiheinen kääntäminen

Koska CUDA-ohjelmat käännetään sekä virtuaalisille että todellisille arkkitehtuureille, käännösprosessi on kaksivaiheinen[29].

1. vaiheessa CUDA C++ -koodi käännetään valitun virtuaaliarkkitehtuurin kanssa yhteensopivaksi PTX-koodiksi. Koodi tallennetaan PTX-tiedostoon.
2. vaiheessa PTX-koodi käännetään todellisen kohdearkkitehtuurin natiivikoodiksi. Näin syntyy *cubin*-tiedosto (CUDA binary).

Vaiheet 1–2 toistetaan jokaiselle halutulle virtuaaliarkkitehtuurin ja todellisen arkkitehtuurin kombinaatiolle. NVCC pakkaa käännösprosessin ensimmäisessä vaiheessa tuotetut PTX-tiedostot ja toisessa vaiheessa tuotetut cubin-tiedostot samaan *fatbinaryyn*. Fatbinary on siis ohjelmabinääritiedosto, joka sisältää joukon eri arkkitehtuurien kanssa yhteensopivia PTX-ja cubin-tiedostoja. [29]

5.5 JIT-kääntäminen

Usein ei voida etukäteen tietää, millä laitteilla CUDA-ohjelmaa tullaan suorittamaan. On myös mahdollista, että ohjelmaa tullaan suorittamaan sellaisilla laitteilla ja arkkitehtuureilla, joita ei vielä ole edes olemassa. Tästä syystä CUDA tukee JIT-kääntämistä (just-in-time): tarvittaessa fatbinaryn sisältämä PTX-koodi käännetään kohdearkkitehtuurin natiivikoodiksi vasta ajonaikaisesti.

Kun CUDA-ohjelma suoritetaan, CUDA-ajuri tarkistaa, onko fatbinaryssä kohdearkkitehtuurin kanssa yhteensopivaa cubin-tiedostoa. Jos on, ajuri lähettää sen sellaisenaan grafiikkaprosessorille suoritettavaksi. Jos ei, ajuri valitsee PTX-tiedostoista sopivimman ja JIT-kääntää sen cubin-objektikoodiksi ajonaikaisesti, ja vasta tämän tehtyään lähettää cubin-ohjelman grafiikkaprosessorille. [29] Tämä mekanismi varmistaa CUDA-ohjelman yhteensopivuuden uudempien laitteiden kanssa. Lisäksi se mahdollistaa CUDA-ohjelman optimoimisen myös tulevaisuuden laitteita ajatellen, sillä JIT-käännösprosessin aikana ajuri voi optimoida ohjelman juuri kyseessä olevaa laitetta silmällä pitäen.

JIT-kääntämisellä on kuitenkin hintansa: se hidastaa ohjelman käynnistymistä merkittävästi. Tästä syystä ajuri tallentaa JIT-käännetyn ohjelman välimuistiinsa (compute cache), jossa se säilyy valmiiksi käännettynä ohjelman tulevilla suorituskerroilla uudelleenkäyttöä varten. Kun ajuri päivitetään, välimuisti invalidoidaan, jolloin ajuri kääntää ohjelman uudestaan. Näin kaikki CUDA-ohjelmat pääsevät hyötymään uudempien ajuriversioiden suorituskykyparannuksista. [13]

CUDA-ohjelma voi myös halutessaan JIT-käännättää sisältämänsä CUDA C++ -koodin itse NVRTC-ohjelmointirajapintaa[32] (Nvidia Runtime Compilation) hyväksi käyttäen. [13]

6. GPGPU-LASKENNAN TEHOKKUUS

Motivaationa GPGPU-laskennan hyödyntämiseen on lähes poikkeuksetta jokin suorituskykyyn liittyvä näkökohta. Niinpä tässä luvussa tarkastellaan lyhyesti Nvidian grafiikkaprosessorien suorituskykyä sekä absoluuttisesti että suhteessa yleiskäyttöisiin prosessoreihin.

6.1 Tarkastelussa hyödynnetty materiaali

Tässä luvussa merkittävässä roolissa on JetBrainsin vuosina 2007–2020 julkaistuista Nvidian ja AMD:n grafiikkaprosessoreista keräämä data[33]. Tässä datassa ovat mukana myös myös luvussa 3 esitellyt GA102- ja GA104-grafiikkaprosessorit. AMD:n grafiikkaprosessoreita koskeva data on suodatettu tarkastelusta pois.

Lisäksi tarkastelussa tukeudutaan muutamien akateemisten tutkimusten tuloksiin. Tutkimuksissa on vertailtu Nvidian grafiikkaprosessorien suorituskykyä ja energiatehokkuutta suhteessa yleiskäyttöisiin prosessoreihin ja FPGA-piireihin. Tutkimuksia ei sisällytetty enempää, jotta työn laajuus pysyisi tarkoituksenmukaisena.

Tutkimuksissa saatujen tulosten yleistämistä estävät ainakin seuraavat seikat:

- Ovatko tutkimuksessa käytetyt laitteet hinnaltaan ja ominaisuuksiltaan vastaavia?
- Miten määritellään hinnan ja ominaisuuksien vastaavuus, kun kyseessä ovat toisistaan selvästi eroavat ja eri käyttökohteisiin tarkoitetut tuotteet?
- Missä määrin vanhemmissa tutkimuksissa saadut tulokset ovat sovellettavissa nykyiseen teknologiaan?
- Kuinka paljon sovelluskohde suosii kunkin prosessorityypin ominaisuuksia?
- Mitä teknologioita sovelluksen toteuttamisessa on hyödynnetty[34]?

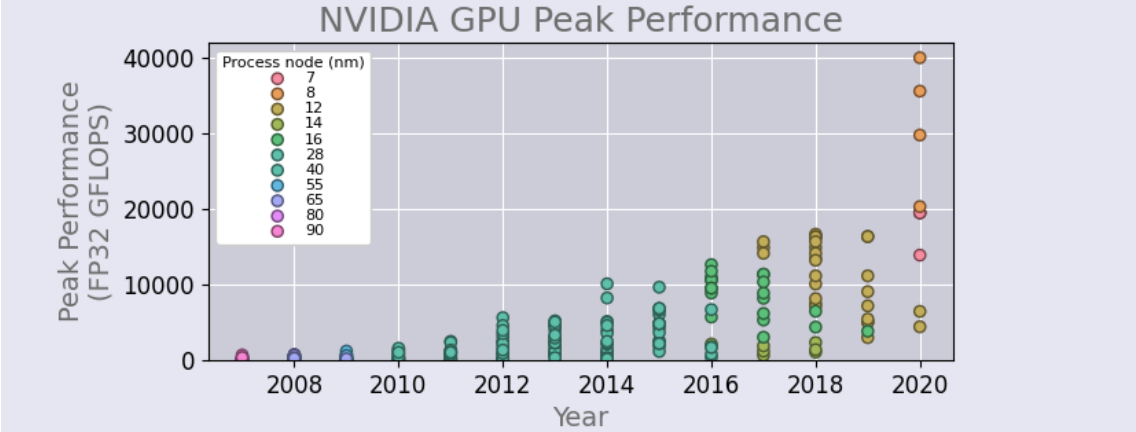
Edellä mainituista syistä tässä luvussa tyydytään, kuten yllä todettiin, tarkastelemaan grafiikkaprosessorien suorituskykyä hyvin karkealla tasolla.

6.2 Laskentateho

Tässä yhteydessä laskentateholla tarkoitetaan prosessorin suurinta teoreettista laskentatehoa. Tämän suureen perusyksikkönä on tapana käyttää *FLOPS:ia* (floating point operations per second). FLOPS-luku ilmaisee, kuinka monta liukulukulaskuoperaatiota proses-

sori kykenee suorittamaan sekunnissa. Kontekstista riippuu, tarkoitetaanko liukuluvulla tässä yhteydessä 16-, 32-vai 64-bittistä liukulukua (FP16/FP32/FP64).

Kuvassa 6.1 on esitetty Nvidian grafiikkaprosessorien teoreettisen maksimisuorituskyvyn kehitys vuosina 2007–2020. Kuten kuvasta havaitaan, suorituskyky ei ole ainoastaan noussut vaan sen kehitys on kiihtymässä. Kuten työn johdannossa todettiin, prosessorien sekventiaalisen suorituskyvyn kehittyminen on lähes pysähtynyt. Lisäksi tietokonegrafiikan merkitys erilaisissa sovelluksissa kasvaa koko ajan, mikä luonnollisesti edellyttää grafiikkaprosessoreilta yhä suurempaa suorituskykyä. Näistä syistä kuvasta havaittava rinnakkaisuuteen perustuva laskentatehon kehitys ei olekaan lainkaan yllättävää.



Kuva 6.1. *Nvidian grafiikkaprosessorien maksimilaskentatehon (FP32 GFLOPS) kehitys 2007–2020. Perustuu lähteeseen [33].*

Lähteen [33] mukaan Nvidian RTX 3070 -näytönohjaimen (jonka grafiikkaprosessori on luvussa 3 esitelty GA104) laskentateho on noin 20310 GFLOPS (FP32). Vertailun vuoksi Intelin 40-ytiminen lippulaivaprosessori Xeon Platinum 8380 pystyy suorittamaan 32 x 40 = 1280 FP32-laskutoimitusta kellojaksoa kohden[35]. 2,30 GHz:n kellotaajuudella[36] tämä tekee n. 2944 GFLOPS. RTX 3070:n teoreettinen laskentateho on siis noin 7-kertainen Xeon 8380:aan nähden. Kirjoitushetkellä RTX 3070:n arvonlisäverollinen kuluttajahinta Suomessa on noin 1000 ja Xeon 8380:n noin 10000 euroa. Xeon 8380:aan verrattuna RTX 3070 tarjoaa siis noin 70-kertaisen laskentatehon euroa kohden.

Vuosi	Sovellus	CPU	GPU	FPGA
2010	Monte Carlo -finanssisimulaatio[37]	$\times 1.0$	$\times 50$	$\times 420$
2012	Smith–Waterman-algoritmi[38]	$\times 1.0$	$\times 8.2$	$\times 35$
2014	Kolmipisteinen Viterbi-dekoodaus[39]	$\times 1.0$	$\times 7.5$	$\times 9.8$
2016	Reaaliaikainen kuvankäsittely[40]	$\times 1.0$	$\times 150$	$\times 1600$

Taulukko 6.1. *Eräitä tutkimuksia, joissa on mitattu saman algoritmin suhteellista suorituskykyä eri laitteilla. Mikäli tutkimuksessa oli useita tuloksia per kategoria, taulukossa on esitetty niiden keskiarvo.*

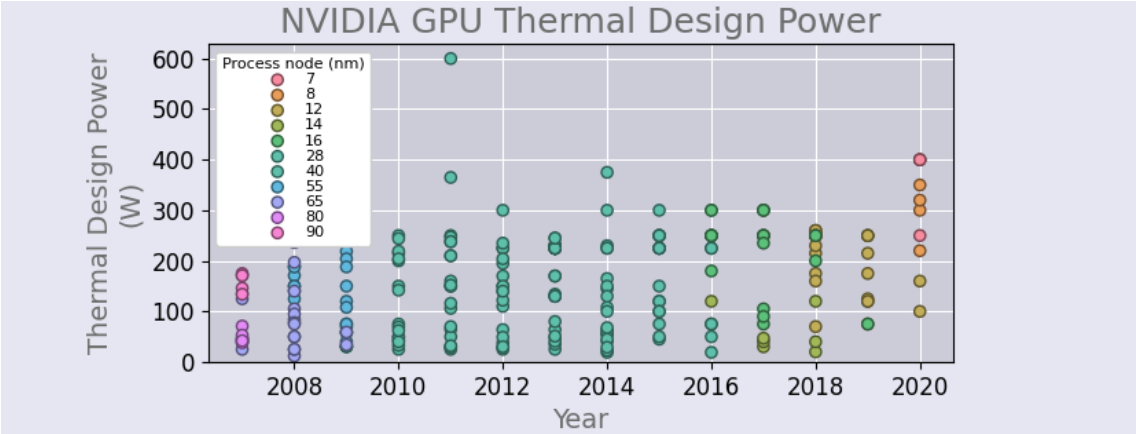
Vaikka teoreettinen GFLOPS-arvo onkin oikein hyvin suuntaa antava suorituskyvyn mittari, käytännön sovelluksissa suorituskykyyn vaikuttavat muutkin tekijät kuin raaka laskentateho. Siksi mielenkiintoisempaa onkin usein tietää, kuinka paljon yleiskäyttöisiä prosessoreja nopeammin grafiikkaprosessorit suoriutuvat rinnakkaisesta laskennasta reaalimaailman sovelluksissa.

Taulukkoon 6.1 on koottu tuloksia eräistä akateemisista tutkimuksista, joissa grafiikkaprosessorien suorituskykyä on verrattu yleiskäyttöisiin prosessoreihin ja FPGA-piireihin.

Tuloksista havaitaan, että grafiikkaprosessori voi hyvinkin nopeuttaa laskentaa jopa satakertaisesti yleiskäyttöiseen prosessoriin nähden. Edellä suoritetun GFLOPS-vertailun perusteella tutkimuksissa esitetyt lukemat vaikuttavat uskottavilta.

6.3 Energiatehokkuus

Tässä yhteydessä energiatehokkuudella tarkoitetaan prosessorin laskentatehoa kulutettua energian yksikköä kohti. Energiatehokkuus on erityisen kiinnostava metriikka aikana, jona digitaalilaitteet syövät merkittävän osan yhteiskunnan kuluttamasta energiasta, yhä useampi laite toimii akulla ja grafiikkaprosessorit yleistyvät halvoissakin laitteissa. Se on myös tietyssä mielessä reilumpi laskennan tehokkuuden mittari kuin absoluuttinen suorituskyky, koska se ottaa huomioon laskentatehon lisäksi laskentaan kulutetun energian.

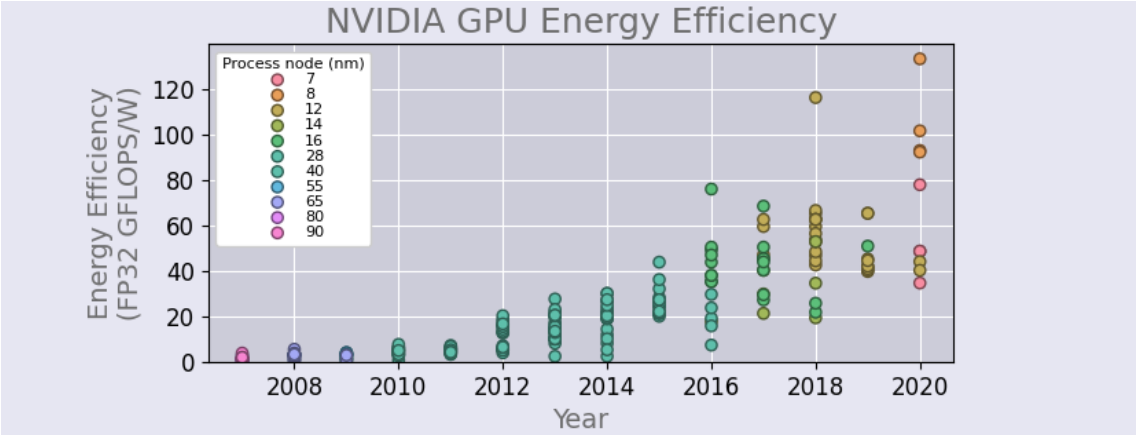


Kuva 6.2. *Nvidian grafiikkaprosessorien tuottaman hukkalämmön kehitys 2007–2020. Perustuu lähteeseen [33].*

Kuvassa 6.2 on esitetty Nvidian grafiikkaprosessorien tuottaman hukkalämmön ja kuvassa 6.3 puolestaan niiden energiatehokkuuden kehitys vuosina 2007–2020. Kuvasta 6.2 havaitaan, että absoluuttinen hukkalämpö on pysynyt vuosien saatossa lähes samana. Edellä kuitenkin havaittiin Nvidian grafiikkaprosessorien laskentatehon kasvaneen kiihtyvää vauhtia. Kuva 6.3 vahvistaa tästä vedettävän johtopäätöksen: grafiikkaprosessorien energiatehokkuus on kasvanut lähes käsi kädessä laskentatehon kanssa.

Lienee perusteltua olettaa, että grafiikkaprosessorien laskentatehon huikean kasvun on

mahdollistanut suurelta osin juuri niiden energiatehokkuuden parantuminen, sillä alhaisempi energiankulutus mahdollistaa logiikan tiheämmän sijoittelun prosessorilastun sisällä.



Kuva 6.3. *Nvidian grafiikkaprosessorien energiatehokkuuden kehitys 2007–2020. Perustuu lähteeseen [33].*

GPGPU-laskennan energiatehokkuudesta suhteessa yleiskäyttöisiin prosessoreihin sekä FPGA-piireihin on tehty jonkin verran vertailevaa tutkimusta. Taulukkoon 6.2 on koottu eräiden tutkimusten tuloksia.

Vuosi	Sovellus	CPU	GPU	FPGA
2010	Monte Carlo -finanssisimulaatio[37]	$\times 1.0$	$\times 16$	$\times 340$
2012	Smith–Waterman-algoritmi[38]	$\times 1.0$	$\times 1.6$	$\times 90$
2014	Kolmipisteinen Viterbi-dekoodaus[39]	$\times 1.0$	$\times 1.4$	$\times 6.8$
2016	Reaaliaikainen kuvankäsittely[40]	$\times 1.0$	$\times 220$	$\times 22000$

Taulukko 6.2. *Eräitä tutkimuksia, joissa on mitattu saman algoritmin suhteellista energiatehokkuutta eri laitteilla. Mikäli tutkimuksessa oli useita tuloksia per kategoria, taulukossa on esitetty niiden keskiarvo.*

Tuloksista havaitaan, että grafiikkaprosessorit voivat olla jopa kertaluokkaa energiatehokkaampia kuin yleiskäyttöiset prosessorit. Tässä vertailussa ne eivät kuitenkaan pärjää FPGA-piireille, joiden energiatehokkuus on vielä kertaluokkaa parempi.

7. YHTEENVETO

Tässä työssä perehdyttiin Nvidian CUDA-ohjelmointialustaan ja sen toteutustekniikkaan.

Luvussa 2 käytiin läpi joitakin rinnakkaiseen laskentaan liittyviä periaatteita. Grafiikkaprosessorit ovat MIMD-moniprosessoreja, joiden avulla voidaan kiihdyttää erityisesti aritmeettisesti intensiivisiä, massiivisen datarinnakkaisia algoritmeja. Amdahlin ja Gustafsonin lakien avulla voidaan etukäteen arvioida, kuinka paljon sovellus tulee rinnakkaistamisen seurauksena nopeutumaan.

Luvussa 3 tarkasteltiin Nvidian grafiikkaprosessoriarkkitehtuuria uusimpia Ampere-prosessoreja esimerkkeinä käyttäen. Nykyaikaiset grafiikkaprosessorit ovat hyvin monimutkaisia moniprosessoreja, jotka koostuvat suuresta määrästä pienempiä prosessoreja.

Luvussa 4 tutustuttiin CUDA-ohjelmoinnin perusperiaatteisiin. CUDA perustuu kolmeen avainabstraktioon: säiejoukkojen hierarkiaan, jaettuun muistiin ja barrier-synkronointiin. Näiden CUDA C++ -ohjelmointikielen abstraktioiden avulla CUDA tekee grafiikkaprosessorin rinnakkaisesta ohjelmoinnista verraten helposti lähestyttävää.

Luvussa 5 perehdyttiin heterogeenisten CUDA-ohjelmien käännösprosessiin liittyviin seikkoihin. Grafiikkaprosessorien arkkitehtuuria kehitetään huomattavasti perinteisiä yleiskäyttöisiä prosessoreja aggressiivisemmin, mistä seuraa uniikkeja kääntäjäteknisiä haasteita. Nvidian NVCC-kääntäjä hyödyntää käännösprosessissa mm. virtualisointia ja kaksivaiheista kääntämistä.

Lopuksi luvussa 6 arvioitiin lyhyesti GPGPU-laskennan tehokkuutta sekä laskentatehon että energiatehokkuuden kannalta. Tutkimusten perusteella grafiikkaprosessorin hyödyntäminen voi parantaa GPGPU-laskentaan soveltuvan sovelluksen suorituskykyä hyvinkin kymmenkertaisesti ja energiatehokkuutta jopa satakertaisesti keskusprosessoriin nähden.

LÄHTEET

[1] Patterson, D. A. ja Hennessy, J. L. *Computer Organization and Design: The Hardware/Software Interface*. eng. Cambridge, MA, 2021.

[2] Czarnul, P. *Parallel Programming for Modern High Performance Computing Systems*. eng. Boca Raton, FL, 2018.

[3] Amdahl, G. M. Computer Architecture and Amdahl's Law. eng. *Computer (Long Beach, Calif.)* 46.12 (2013), s. 38–46. ISSN: 0018-9162.

[4] Gustafson, J. L. Gustafson's Law. *Encyclopedia of Parallel Computing*. Toim. D. Padua. Boston, MA: Springer US, 2011, s. 819–825. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_78. URL: https://doi.org/10.1007/978-0-387-09766-4_78 (viitattu 10. 11. 2021).

[5] Gustafson, J. Reevaluating Amdahl's law. eng. *Communications of the ACM* 31.5 (1988), s. 532–533. ISSN: 0001-0782.

[6] NVIDIA Corporation. *CUDA C++ Best Practices Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide (viitattu 27. 01. 2022).

[7] McCool, M. *Structured parallel programming patterns for efficient computation*. eng. Amsterdam ; 2012.

[8] Hennessy, J. L. ja Patterson, D. A. *Computer Architecture: A Quantitative Approach*. eng. Waltham, MA, 2012.

[9] Flynn, M. Very high-speed computing systems. eng. *Proceedings of the IEEE* 54.12 (1966), s. 1901–1909. ISSN: 0018-9219.

[10] Ilg, M., Rogers, J. ja Costello, M. Projectile Monte-Carlo Trajectory Analysis Using a Graphics Processing Unit (elokuu 2011). DOI: 10.2514/6.2011-6266.

[11] Kirk, D. ja Nickolls, J. *Computer Organization and Design: The Hardware/Software Interface. Appendix B. Graphics and Computing GPUs*. eng. 2021.

[12] Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G. ja Dongarra, J. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. eng. *Parallel computing* 38.8 (2012), s. 391–407. ISSN: 0167-8191.

[13] NVIDIA Corporation. *CUDA C++ Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (viitattu 13. 11. 2021).

[14] NVIDIA Corporation. *CUDA Driver API*. URL: https://docs.nvidia.com/cuda/cuda-driver-api (viitattu 04. 02. 2022).

[15] NVIDIA Corporation. NVIDIA Ampere GA102 GPU Architecture. Second-Generation RTX (2021). URL: https://www.nvidia.com/content/PDF/nvidia-

ampere-ga-102-gpu-architecture-whitepaper-v2.pdf (viitattu 15. 11. 2021).

[16] NVIDIA Corporation. *CUDA Runtime API*. URL: https://docs.nvidia.com/cuda/cuda-runtime-api (viitattu 27. 01. 2022).

[17] NVIDIA Corporation. *Introducing Low-Level GPU Virtual Memory Management*. URL: https://developer.nvidia.com/blog/introducing-low-level-gpu-virtual-memory-management/ (viitattu 13. 01. 2022).

[18] NVIDIA Corporation. *NVIDIA CUDA-X. GPU-Accelerated Libraries*. URL: https://developer.nvidia.com/gpu-accelerated-libraries (viitattu 13. 01. 2022).

[19] Microsoft Corporation. *C++ AMP (C++ Accelerated Massive Parallelism)*. URL: https://docs.microsoft.com/en-us/cpp/parallel/amp/cpp-amp-cpp-accelerated-massive-parallelism?view=msvc-160 (viitattu 20. 11. 2021).

[20] Microsoft Corporation. *Compute Shader Overview*. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d11/direct3d-11-advanced-stages-compute-shader (viitattu 20. 11. 2021).

[21] openacc-standard.org. *The OpenACC® Application Programming Interface. Version 3.2*. URL: https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.2-final.pdf (viitattu 20. 11. 2021).

[22] The Khronos Group, Inc. *OpenCL Reference Pages (Version V3.0.9)*. URL: https://www.khronos.org/registry/OpenCL/sdk/3.0/docs/man/html/ (viitattu 14. 11. 2021).

[23] The Khronos Group, Inc. *Compute Shader. OpenGL Wiki*. URL: https://www.khronos.org/opengl/wiki/Compute_Shader (viitattu 14. 11. 2021).

[24] The OpenMP Architecture Review Board. *OpenMP Accelerator Support for GPUs*. URL: https://www.openmp.org/updates/openmp-accelerator-support-gpus/ (viitattu 20. 11. 2021).

[25] The Khronos Group, Inc. *Compute Shaders. Vulkan Guide*. URL: https://vkguide.dev/docs/gpudriven/compute_shaders/ (viitattu 14. 11. 2021).

[26] MathWorks. *MATLAB GPU Computing Support for NVIDIA CUDA-Enabled GPUs*. URL: https://se.mathworks.com/solutions/gpu-computing.html (viitattu 21. 11. 2021).

[27] Advanced Micro Devices, Inc. *HIP Programming Guide*. URL: https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html (viitattu 14. 11. 2021).

[28] Advanced Micro Devices, Inc. *GPUFORT*. URL: https://github.com/ROCmSoftwarePlatform/gpufort (viitattu 21. 11. 2021).

[29] NVIDIA Corporation. *NVCC*. URL: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html (viitattu 19. 12. 2021).

[30] NVIDIA Corporation. *CUDA LLVM Compiler*. URL: https://developer.nvidia.com/cuda-llvm-compiler (viitattu 19. 12. 2021).

[31] NVIDIA Corporation. *PTX ISA. Parallel Thread Execution ISA Version 7.5*. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html (viitattu 20. 12. 2021).

[32] NVIDIA Corporation. *NVRTC (Runtime Compilation)*. URL: https://docs.nvidia.com/cuda/nvrtc/index.html (viitattu 27. 01. 2022).

[33] JetBrains s.r.o. *We Analyzed 495 AMD Radeon and Nvidia GPU Specifications and Shared the Dataset with Everyone*. URL: https://blog.jetbrains.com/datalore/2020/12/07/we-analyzed-495-nvidia-and-amd-radeon-gpu-specifications/ (viitattu 20. 11. 2021).

[34] Pallipuram, V. K., Bhuiyan, M. ja Smith, M. C. A comparative study of GPU programming models and architectures using neural networks. eng. *The Journal of supercomputing* 61.3 (2011), s. 673–718. ISSN: 0920-8542.

[35] Wikipedia. *FLOPS*. URL: https://en.wikipedia.org/wiki/FLOPS (viitattu 20. 11. 2021).

[36] Intel Corporation. *Intel® Xeon® Platinum 8380 Processor*. URL: https://www.intel.com/content/www/us/en/products/sku/212287/intel-xeon-platinum-8380-processor-60m-cache-2-30-ghz/specifications.html (viitattu 20. 11. 2021).

[37] Tian, X. ja Benkrid, K. High-Performance Quasi-Monte Carlo Financial Simulation: FPGA vs. GPP vs. GPU. eng. *ACM transactions on reconfigurable technology and systems* 3.4 (2010), s. 1–22. ISSN: 1936-7406.

[38] Zou, D., Dou, Y. ja Xia, F. Optimization schemes and performance evaluation of Smith-Waterman algorithm on CPU, GPU and FPGA. eng. *Concurrency and computation* 24.14 (2012), s. 1625–1644. ISSN: 1532-0626.

[39] Li, R., Dou, Y. ja Zou, D. Efficient parallel implementation of three-point viterbi decoding algorithm on CPU, GPU, and FPGA. eng. *Concurrency and computation* 26.3 (2014), s. 821–840. ISSN: 1532-0626.

[40] Georgis, G., Lentaris, G. ja Reisis, D. Acceleration techniques and evaluation on multi-core CPU, GPU and FPGA for image processing and super-resolution. eng. *Journal of real-time image processing* 16.4 (2016), s. 1207–1234. ISSN: 1861-8200.