Tampereen yliopisto

Juha Virta

GRAFIIKKAPROSESSORIN HYÖDYNTÄMINEN TIETEELLISESSÄ LASKENNASSA

Tekniikan ja luonnontieteiden tiedekunta
Kandidaatintyö
Kesäkuu 2019

TIIVISTELMÄ

Juha Virta: Grafiikkaprosessorin hyödyntäminen tieteellisessä laskennassa
Kandidaatintyö
Tampereen yliopisto
Teknisten tieteiden kandidaatin tutkinto-ohjelma
Kesäkuu 2019

Tässä työssä tarkastellaan grafiikkaprosessoria ja sen hyödyntämistä tieteellisessä laskennassa. Työn tarkoituksena on tarkastella, miksi grafiikkaprosessori on sopiva työkalu tieteen eri sovelluksiin, miten sitä on mahdollista soveltaa sekä mihin eri tieteen alojen sovelluksiin siitä saadaan hyötyjä.

Työssä käydään ensin läpi grafiikkaprosessorin teknisiä ja toiminnallisia ominaisuuksia sekä annetaan kuva grafiikkaprosessorin ja keskusprosessorin eroista ja yhteistyömahdollisuuksista. Tämän jälkeen käydään läpi rinnakkaisohjelmointi, jolla grafiikkaprosessorin laskentatehoa hyödynnetään. Sen jälkeen käydään läpi grafiikkaprosessorin hyödyntäminen tieteellisissä sovelluksissa.

Avainsanat: grafiikkaprosessori, keskusprosessori, rinnakkaisuus, ohjelmointi

SISÄLLYSLUETTELO

KUVALUETTELO

LYHENTEET JA MERKINNÄT

GPU	engl. Graphics Processing Unit, grafiikkaprosessori
CPU	engl. Central Processing Unit, keskusprosessori
GPGPU	engl. General-Purpose Computing on Graphics Processing Units, grafiikkaprosessorilla suoritettava yleinen laskenta
FLOPS	engl. Floating Point Operations Per Second, liukulukulaskennan operaatioita sekunnissa
SIMD	engl. Single Instruction, Multiple Data, saman operation suorittamista useammalle datalle
thread	pienin prosessoitava yksikkö
kerneli	grafiikkaprosessorilla suoritettava funktio
shader	alunperin grafiikan varjostukseen suunniteltu ohjelma

1. JOHDANTO

Tietokoneella suoritettavalla laskennalla on pystytty saavuttamaan merkittäviä asioita, joita nykytekniikka hyödyntää. Itseajavat autot, sään simulointi, tekoäly ja syväoppiminen neuroverkoilla ovat hyviä esimerkkejä siitä, mitä laskennalla voi saada aikaiseksi. Ne ovat myös esimerkkejä ongelmista, jotka pelkällä keskusprosessorilla suoritettuna vievät todella paljon aikaa. Grafiikkaprosessorin avulla pystytään nykyisin luomaan korkealaatuista grafiikkaa, mutta sitä pystytään käyttämään hyödyksi myös laskennallisesti tieteen saralla, sillä sen avulla voidaan nopeuttaa laskentaa huomattavasti. Grafiikkaprosessorien kehitys on myös suuntautunut siten, että nykyään ne ovat kätevästi ohjelmoitavia komponentteja sen sijaan, että niiden algoritmit olisivat tiukasti sidottu enää vain grafiikan tuottamiseen.

Monissa tieteen sovelluksissa vaadittava laskenta on usein erittäin laajaa, mutta tietynlaisissa ongelmissa laskenta pystytään jakamaan pienempiin osiin, jotka voidaan laskea samanaikaisesti. Tällaisissa tapauksissa laskennassa pystytään hyödyntämään grafiikkaprosessorin useita ytimiä ja suorittamaan laskenta rinnakkaislaskentana. Oikein sovellettuna grafiikkaprosessorilla tehtävällä laskennalla pystytään suorittamaan vaadittu tehtävä jopa kymmeniä kertoja nopeammin kuin keskusprosessorilla suoritettuna.

Tässä työssä tutkitaan, kuinka grafiikkaprosessoria on mahdollista hyödyntää tietokoneella suoritettavassa laskennassa, ja millaisia hyötyjä siitä on mahdollista saada. Toisessa luvussa tarkastellaan grafiikkaprosessoria yleisesti, sekä sen toiminnallisia eroja keskusprosessorin kanssa ja miten eri prosessorityyppejä hyödynnetään yhdessä. Yleiseen tarkasteluun kuuluu sen historia, toiminta ja arkkitehtuuri. Kolmannessa luvussa perehdytään grafiikkaprosessorilla suoritettavaan rinnakkaisohjelmointiin, eli miten ohjelmia pilkotaan pienempiin osiin, miten grafiikkaprosessorilla suoritetaan ohjelmia ja miten grafiikkaprosessori ja keskusprosessori jakavat tehtäviä, sekä tarkastellaan käytettävissä olevia tärkeimpiä ohjelmointirajapintoja. Myös rinnakkaisohjelmoinnin haasteet käydään läpi. Neljännessä luvussa käsitellään tieteellisiä sovelluskohteita, joissa grafiikkaprosessoria hyödynnetään, sekä miten sitä sovelletaan käytännön laskennassa. Erityisesti tarkastellaan syväoppimista neuroverkoilla, joka on grafiikkaprosessorien potentiaalisesti tärkein sovelluskohde. Viidennessä luvussa on asioiden yhteenveto.

2. GRAFIIKKAPROSESSORIN OMINAISUUDET

Ensimmäinen varsinainen grafiikkaprosessori, Nvidian GeForce 256, kehitettiin vuonna 1999 [1]. Aluksi grafiikkaprosessorin tarkoitus oli vain kolmiulotteisen grafiikan tuottaminen [2]. Sen toiminta oli sidottua, eikä se ollut suuresti muokattavissa. Ensimmäiset ohjelmoitavat *shaderit* tulivat vuonna 2001, jolloin grafiikkaprosessoreista tuli hieman muokattavampia [1]. Kolmiulotteisen grafiikan parantamiseksi grafiikkaprosessorin ytimistä tuli yleisemmin ohjelmoitavia vuonna 2006, jolloin yhdellä ytimellä voitiin suorittaa erilaisia algoritmeja [2]. Tämän jälkeen grafiikkaprosessorilla suoritettava yleinen laskenta (GPGPU, General-purpose computing on graphics processing units) helpottui merkittävästi ja alkoi yleistyä [3]. Nykyään monet maailman tehokkaimmista supertietokoneista on grafiikkaprosessorikiihdytettyjä, mukaan lukien kaikista tehokkain eli Oak Ridge National Laboratoryn *Summit* [4].

Grafiikkaprosessori on näytönohjaimen keskeinen osa, joten se on myös nykyaikaisten tietokoneiden tärkeä peruskomponentti. Suurimmat grafiikkaprosessorien valmistajat ovat AMD ja Nvidia. [5]

2.1. Toiminta

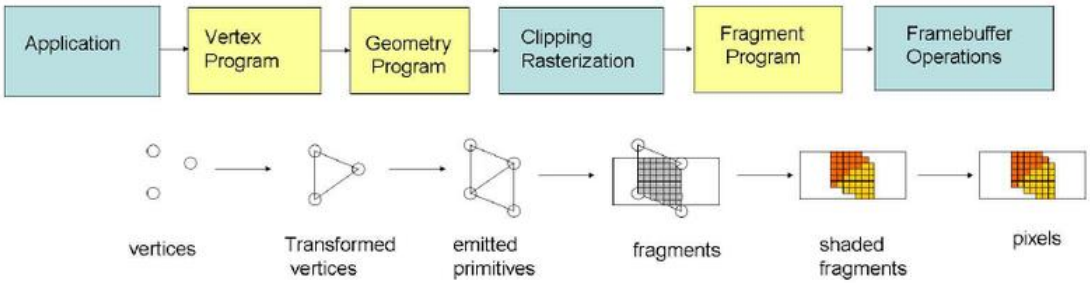
Grafiikkaprosessori on erittäin hyvä suorittamaan tietyntyyppistä laskentaa. Grafiikkaprosessorilla suoritettavalle laskennalle tyypilliset ominaisuudet ovat suuri määrä laskettavaa ja merkittävä rinnakkaisuus. Lisäksi suoritustehon (throughput) on oltava tärkeämpää kuin viive (latency) [6]. Grafiikan reaaliaikaisessa renderoinnissä vaaditaan miljardeja pikselioperaatioita sekunnissa, joten grafiikkaprosessorin täytyy pystyä suorittamaan valtavia määriä laskentaa. Grafiikan tuottaminen on kuitenkin hyvin rinnakkaistettavissa, kun kärkipiste- ja pikselioperaatiot voidaan laskea samanaikaisesti. Grafiikan tuottamisessa ruudulle suoritustehoa priorisoidaan viiveen yli, koska ihmisen silmä pystyy havaitsemaan millisekunteja (10^{-3}), kun taas prosessori suorittaa operaatioita nanosekunneissa (10^{-9}). [6]

Graphics pipeline on tapahtumasarja, jolla grafiikkaprosessori luo reaaliaikaista grafiikkaa tietokoneen ruudulle. Tapahtumasarja koostuu käytännössä kahdesta osasta, geometriaoperaatioista ja rasteroinnista [7].

Geometriaoperaatioissa käytetään yleensä kolmioita kolmiulotteisessa koordinaatistossa. Kolmiot koostuvat kärkipisteistä (vertex), jotka siirretään koordinaatistoon ja varjostetaan. Varjostus tapahtuu yleensä laskemalla niiden

vuorovaikutuksen valon kanssa kyseisessä näkymässä. Kärkipisteet voidaan laskea toisistaan riippumatta. Sen jälkeen kärkipisteistä kootaan kolmioita. [6]

Rasteroinnissa määritetään, mitkä kolmiot vaikuttavat mihinkin pikseleihin. Jokainen kolmio luo fragmentin jokaiseen pikseliin, jonka alueelle se kuuluu. Koska kolmioita voi olla useampi päällekkäin, pikselin väriarvo voidaan joutua laskemaan useasta fragmentista. Fragmenttien lopullinen väri saadaan varjostamalla ne kärkipisteiden värien ja tekstuurien avulla. Myös fragmentit voidaan varjostaa riippumatta toisistaan. Fragmenteista muodostetaan lopullinen kuva ruudulle, jossa on yksi väri yhdessä pikselissä. [6] Kuvassa 1 näkyy tapahtumasarjan vaiheet.



Kuva 1: *Grafiikan luomisen tapahtumasarja [8]*

Aiemmin edellä mainitut operaatiot eivät olleet ohjelmoitavissa. Kiinteätoiminen tapahtumasarja ei pysty tehokkaasti tekemään monimutkaisempia varjostus- ja valotusoperaatioita, joita vaaditaan monimutkaisempiin efekteihin. Keskeinen askel eteenpäin oli kiinteiden kärkipiste- ja fragmenttioperaatioiden korvaaminen käyttäjän määrittelemillä ohjelmilla, jotka suorittavat kyseiset operaatiot. Nykyiset grafiikkaprosessorit eivät enää suorita varjostusoperaatioita erikseen, vaan nykyisin on käytössä yhtenäinen varjostusmalli (Unified Shader model). Kun varjostusmalli on kehittynyt ja grafiikkaprosessorien sovellukset ovat kasvattaneet ohjelmien monimutkaisuutta, myös grafiikkaprosessorien arkkitehtuuri on keskittynyt ohjelmoitavien osien kehittämiseen. Nykyiset grafiikkaprosessorit ovatkin ohjelmoitavia koneita, joilla on tukenaan kiinteätoimisia yksikköjä. [6]

2.2. Arkkitehtuuri

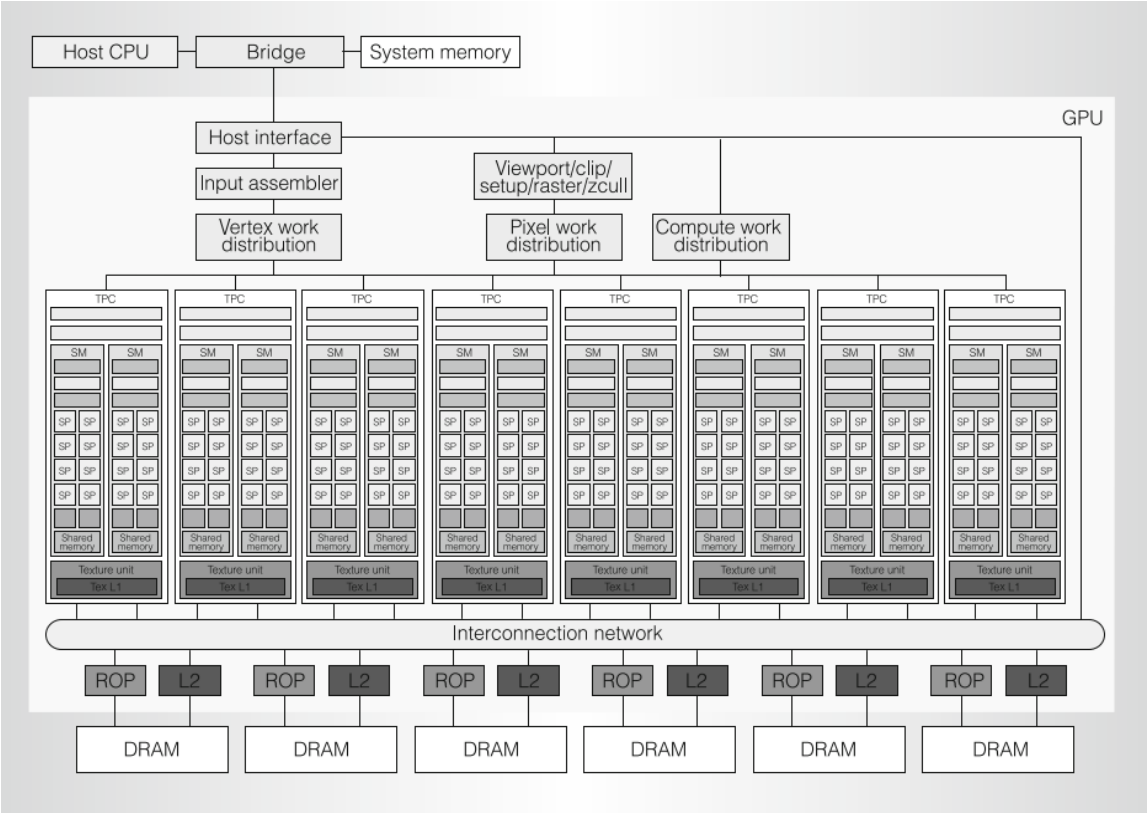
Grafiikkaprosessorien mikroarkkitehtuuri kehittyy koko ajan paremmaksi, mikä lisää samalla grafiikkaprosessorien suorituskykyä. Arkkitehtuurin merkitys kasvaa koko ajan

ydinten korkean lukumäärän vuoksi [9]. Nvidian ensimmäinen ohjelmoitava mikroarkkitehtuuri oli Tesla [3], jonka jälkeen uusia arkkitehtuurimalleja on kehitetty tasaisin väliajoin aina nykyisiin Voltaan ja Turingiin asti [10,11]. Grafiikkaprosessorien arkkitehtuureissa on kuitenkin tietynlaisia samankaltaisuuksia, joten tarkastelu yleisemmällä tasolla on järkevää [12], minkä jälkeen yhden arkkitehtuurin yksityiskohtaisempi tarkastelu on selkeämpää.

Grafiikkaprosessorit ovat moniytimisiä laitteita, jotka toimivat Flynnin luokittelun [13] SIMD (single instruction multiple data) -periaatteella. Se tarkoittaa laitteita, joilla on useita prosessointielementtejä, jotka suorittavat samaa operaatiota usealle datalle samaan aikaan ja käyttävät hyväkseen datatasoista rinnakkaisuutta (data level parallelism). [14]

Grafiikkaprosessorit koostuvat laskentayksiköistä. Laskentayksikkö sisältää rekisterejä, paikallista muistia, vakiomuistia (constant memory) sekä prosessointielementtejä, ja se voi suorittaa ainakin yhden työryhmän (work group) rinnakkain. [12]

Grafiikkaprosessoreilla on myös yleinen hierarkkinen muistimalli, joka koostuu neljästä päämuistista. Yleinen muisti (global memory) on sen päämuisti, joka on käytettävissä kaikille laskentayksiköille. Kuvamuisti on erityinen tila yleisen muistin käytössä. Paikallinen muisti on nopea muistivarasto jokaiselle laskentayksikölle, ja se on sekä luettavissa että kirjoitettavissa työryhmän kaikille yksiköille. Vakiomuisti on muistia, joka on vain luettavissa ja jolla on lyhyt viive. [12] Vakiomuistia pidetään varattuna datalle, joka pysyy vakiona suoritettaessa grafiikkaprosessorilla ajettavia funktioita eli kerneleitä [15]. Kaikilla grafiikkaprosessoreilla on myös yhteneväisyyksiä komponenteissa, joista arkkitehtuuri koostuu. Kuvassa 2 näkyy Nvidian Tesla -mikroarkkitehtuuri, jossa on yhtenäinen varjostusmalli.



Kuva 2: *Tesla -mikroarkkitehtuuri [16]*

Tesla -mikroarkkitehtuurissa on 8 laskentayksikköä eli TPC:tä (texture/processor cluster). Kyseinen grafiikkaprosessori sisältää 128 ydintä eli *streaming prosessoria* (SP), jotka on jaettu 16:een *streaming multiprosessoriin* (SM). Jokaisessa laskentayksikössä on siis 2 streaming multiprosessoria ja 16 streaming prosessoria. Streaming prosessorien ryhmä on se osa, joka suorittaa ohjelmoitavan laskennan. [16]

Muistisysteemi koostuu ulkoisesta dynaamisen muistin (DRAM, Dynamic Random Access Memory) hallinnasta. Sidotun toiminnan rasterointiprosessorit (ROP, Raster Operating Processor) suorittavat väri- ja syvyysoperaatioita suoraan muistiin. Arkkitehtuuri sisältää myös L1 ja L2 *cacheja* eli välimuistia sekä tekstuuriyksikköjä (Texture unit). L1 cache on pienempi ja nopeampikäyttöinen kuin L2 cache. Kuvan yläreunassa olevat lohkot toimittavat ytimille tulevan sisääntulon.

2.3. Erot keskusprosessoriin

Keskusprosessori ja grafiikkaprosessori ovat valmistettu hyvin erilaisiin filosofioihin perustuen. Keskusprosessori on suunniteltu suorittamaan monia erilaisia sovelluksia

sekä nopeaan toimintaan yhdelle tehtävälle. Sen arkkitehtuurillinen kehitys on mahdollistanut suuren kehityksen suorituskyvyn parantamisessa, mutta johtanut pinta-alan monimutkaisuuden sekä tehon kulutuksen kasvuun. Tämän takia keskusprosessorissa voi olla vain pieni määrä prosessointiytimiä, jotta se pysyisi tehon kulutuksen ja lämmöntuoton kannalta tiettyjen rajojen sisäpuolella. [17]

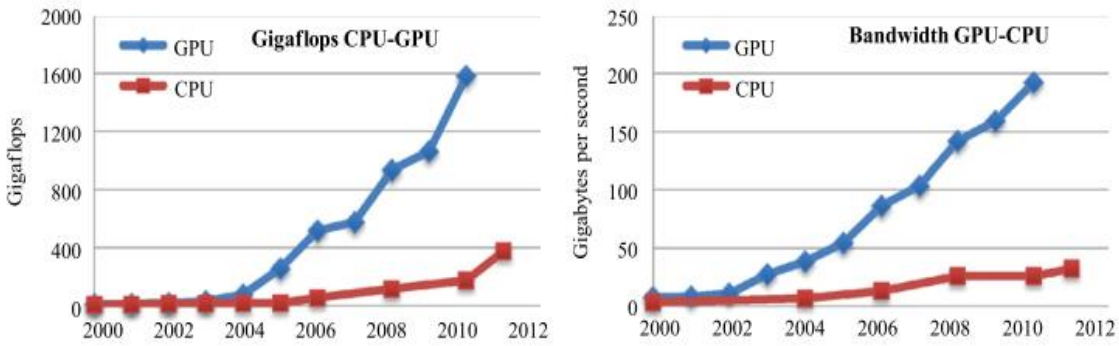
Tietokoneen laskentatehoa mitataan liukulukulaskennan operaatioilla sekunnissa (FLOPS, floating point operations per second). Grafiikkaprosessorilla on huomattavasti suurempi huippulaskentateho kuin keskusprosessorilla [9]. Kuitenkin keskusprosessori antaa parhaan suorituskyvyn yhdelle *threadille* [17].

Yksi suurimmista tekijöistä keskusprosessorin suorituskykyyn on perinteisesti ollut sen suoritustaajuuden tasainen kasvaminen. Kaksinkertaistamalla suoritustaajuuden saatiin myös kaksinkertaistettua sen suorituskyky. Tässä ollaan kuitenkin tultu suorituskyvyn rajoille lämmöntuoton osalta ja suoritustaajuuden maksimiarvo on hieman alle 4.0 GHz. [18] Grafiikkaprosessorien suoritustaajuus on alhaisempi, noin 1.5 GHz. Taulukossa 1 on esitelty rakenteellisia sekä suorituskykyyn perustuvia eroja prosessorityyppien välillä.

	Num. PE	Frequency (GHz)	Num. Transistors	BW (GB/sec)	SP SIMD width	DP SIMD width	Peak SP Scalar FLOPS (GFLOPS)	Peak SP SIMD Flops (GFLOPS)	Peak DP SIMD Flops (GFLOPS)
Core i7-960	4	3.2	0.7B	32	4	2	25.6	102.4	51.2
GTX280	30	1.3	1.4B	141	8	1	116.6	311.1/933.1	77.8

Taulukko 1: *Prosessorityyppien vertailu [17]*

Taulukossa on vertailussa Intelin Core i7 960 keskusprosessori ja Nvidian GTX280 grafiikkaprosessori. Prosessointielementtien ja suoritustaajuuden lisäksi eroja on transistorien määrässä, joka on suurempi grafiikkaprosessorissa. Grafiikkaprosessorissa on myös suurempi muistikaistanleveys. Pieniä eroja on myös single precision (SP), eli 32 bittisten liukulukujen ja double precision (DP), eli 64 bittisten liukulukujen SIMD leveyksissä. Taulukosta nähdaan myös huippulaskentateho eri inputeilla. Eroja on lisäksi myös välimuistien suuruudessa. Keskusprosessorissa on enemmän välimuistia, jotta se toimisi mahdollisimman tehokkaasti ja viivettä saataisiin vähennettyä. Esimerkkinä Intel Core i7 960:ssa on erilliset 32 KB L1 cachet ohjeille ja datalle jokaiselle ytimelle. Niiden lisäksi jokaiselle ytimelle on 256 KB yhtenäinen L2 cache. Kaikille ytimille on käytössä 8 MB yhteinen L3 cache. GTX280:ssa on muutamia 16 KB cacheja. [17]



Kuva 3: *FLOPS:in ja kaistanleveyksien vertailu [18]*

Kuvassa 3 näkyy grafiikkaprosessorin ja keskusprosessorin laskentatehon vertailu Gigaflopseina vasemmalla ja kaistanleveyden vertailu Gigabitteinä sekunnissa oikealla.

2.4. Heterogeeninen laskenta

Heterogeeninen laskenta on grafiikkaprosessorin ja keskusprosessorin käyttö yhteistyönä maksimaalisen laskentatehon saamiseksi sekä työmäärän tasapainottamiseksi välttämällä seisonta-aikaa molemmille prosessorityypeille [19]. Heterogeeniset systeemit ovat kehittyneet, koska grafiikkaprosessorilla ja keskusprosessorilla on toisiaan täydentäviä ominaisuuksia, joka antaa sovelluksille mahdollisuuden toimia parhaiten käyttämällä molempia prosessorityyppejä. Sarjatyyppiset ohjelman osat suoritetaan keskusprosessorilla ja rinnakkaistyyppiset osat grafiikkaprosessorilla. [1]

Heterogeenisen laskennan nopeutuksen rajoittava tekijä saadaan Amdahlin lain avulla [13]. Nopeutus riippuu siitä, kuinka suuri osa ohjelmasta on rinnakkaistettavissa, sekä kuinka paljon nopeammin rinnakkaiset osat voidaan suorittaa. Nopeuttaminen vaikuttaa kuitenkin vain rinnakkaiseen osaan, joten pullonkaulaksi muodostuu tällöin se osa ohjelmasta, jota ei voida rinnakkaistaa [18]. Amdahlin laki nähdään kaavasta 1.

$$S_T = \frac{1}{(1-p)+\frac{p}{s}} \tag{1}$$

Kaavassa S_T on teoreettinen nopeutuskerroin koko tehtävälle, p on se osa toiminta-ajasta johon rinnakkaisuus vaikuttaa ja s kertoo sen, kuinka monta kertaa nopeammin se osuus voidaan suorittaa. Kuvassa 4 näkyy erilaisia skenaarioita heterogeenisestä laskennasta ja kuinka paljon ohjelmia voi nopeuttaa.

Table 2. CPU+GPU coprocessing execution time, assuming that a CPU core is 5× faster and 50× the area of a GPU core.

	Configuration	Processing time for 1 CPU core	Processing time for 500 GPU cores	Processing time for 10 CPU cores	Processing time for 1 CPU core + 450 GPU cores
Program type	**Area**	**50**	**500**	**500**	**500**
Parallel-intensive program	0.5% serial code	1.0	5.0	1.0	1.00
	99.5% parallelizable code	199.0	0.4	19.9	0.44
	Total	200.0	5.4	20.9	1.44
Mostly sequential program	75% serial code	150.0	750.0	150.0	150.0
	25% parallelizable code	50.0	0.1	5.0	0.11
	Total	200.0	750.1	155.0	150.11

Kuva 4: *Amdahlin laki [1]*

Kuvassa 4 oletuksena on, että keskusprosessorin ydin on 5 kertaa nopeampi sekä 50 kertaa suurempi kuin grafiikkaprosessorin ydin. Ensimmäisissä sarakkeissa on prosessointiajat erikseen keskusprosessorille sekä grafiikkaprosessorille. Viimeisessä sarakkeessa on prosessointiaika heterogeeniselle laskennalle. Ohjelman tyyppejä on merkittävästi rinnakkaistettava ohjelma sekä enimmäkseen sarjamainen ohjelma. Prosessointiajoista huomataan, että enimmäkseen sarjamaisella ohjelmalla ei saada niin suurta nopeuseroa, koska sarjamaisen koodin suorittamisessa menee eniten aikaa.

3. RINNAKKAISOHJELMOINTI

Rinnakkaisohjelmointi eroaa tavallisesta sarjamaisesta ohjelmoinnista laitteistoeroista johtuen. Keskusprosessorilla suoritettava sarjamainen ohjelma suoritetaan vain yhdellä ytimellä, joten se ei käytä hyödykseen koko mahdollisesti käytettävissä olevaa laskentakapasiteettiä. Rinnakkaisohjelmoinnilla taas on potentiaalisesti käytettävissä koko grafiikkaprosessorien prosessointivoima. [14] Rinnakkaisohjelmoinnissa ongelma jaetaan osiin, jotka prosessoidaan samanaikaisesti.

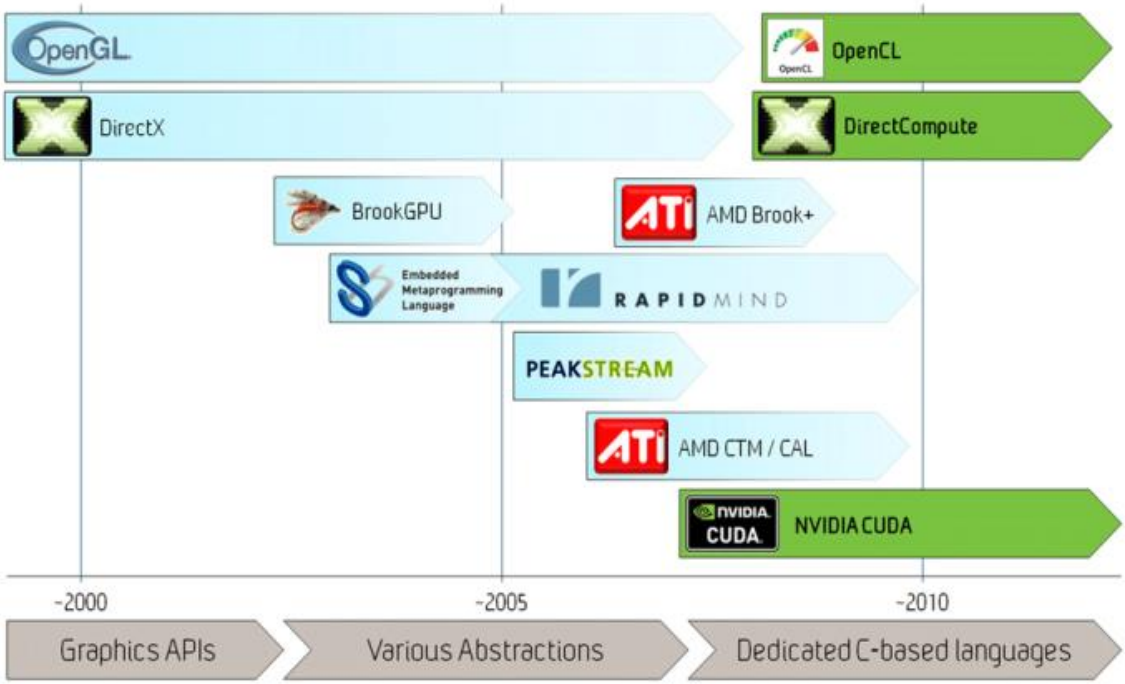
Hyvä esimerkki rinnakkaisohjelmoinnin ja sarjamaisen ohjelmoinnin eroista on kahden vektorin yhteenlasku. Tulosvektorin tietty alkio ei ole riippuvainen muista tekijöistä, kuin sitä vastaavista yhteenlaskettavien vektoreiden alkioista, joten se on hyvin rinnakkaistettavissa. Sarjamaisessa ohjelmoinnissa operaatio suoritetaan silmukalla (loop) ja jokainen alkoiden yhteenlasku tapahtuu järjestyksessä yksitellen. Rinnakkaisohjelmoinnissa jokainen yhteenlasku suoritetaan samanaikaisesti. Lasku pilkotaan niin moneen osaan kuin vektorissa on alkioita, niille annetaan järjestysnumerot ja laskut suoritetaan toisistaan riippumatta.

3.1. Ohjelmointirajapinnat

Rinnakkaisohjelmointia varten on nykyisin kehitetty siihen tarkoitettuja ohjelmointirajapintoja. Ennen rinnakkaisohjelmointia jouduttiin suorittamaan grafiikkarajapintojen kautta, joka oli virheherkkää ja hankalaa [18]. Möys ohjelmat piti suunnitella graphics pipelinen mukaisesti ja ohjelmoitavat yksiköt olivat käytettävissä vain yhtenä askeleena pipelinen sisällä [6].

Grafiikan tuottamiseen tarkoitettuja rajapintoja ovat esimerkiksi OpenGL (Open Graphics Library) ja DirectX, jotka näkyvät kuvasta 5. Tämän jälkeen alkoi tulla ensimmäisiä rajapintoja yleisempää laskentaa varten, joissa grafiikan osuutta oli pienennetty. Nämä olivat akateemisia tai muita kolmannen osapuolen julkaisemia rajapintoja. Esimerkkejä näistä on BrookGPU ja RapidMind. Tämän jälkeen vuonna 2007 yleiseen tarkoitukseen suunnitellut ohjelmointirajapinnat tulivat markkinoille. [18] CUDA (Compute Unified Device Architecture) tarjoaa C:n tapaisen syntaksin rinnakkaisohjelmointiin, ja kun BrookGPU mahdollisti vain datatasoisen rinnakkaisuuden, tarjoaa CUDA tämän lisäksi myös mahdollisuuden *multithreadingiin [6]*. Multithreading on useamman threadin hyödyntämistä prosessoinnissa, jossa toinen on prosessoitavana, kun toinen odottaa [20]. CUDA pystyy myös hyödyntämään laitteiston resursseja paremmin, sekä CUDA:n

kernelit ovat joustavampia [6]. Kuvassa 5 näkyy ohjelmointirajapintojen kehitys sekä yleisimmät rinnakkaisohjelmoinnissa käytetyt rajapinnat.



Kuva 5: *Rinnakkaisohjelmoinnissa käytetyt ohjelmointirajapinnat [18]*

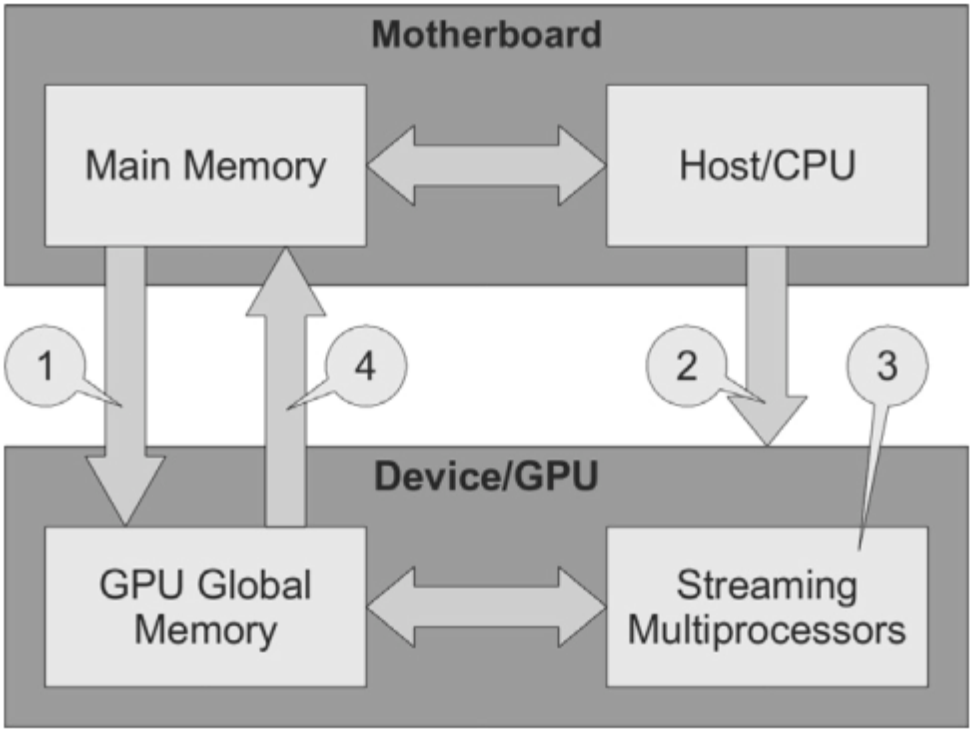
Nykyisin käytössä olevista rajapinnoista suurimmat ovat CUDA, OpenCL (Open Computing Language) sekä DirectCompute. CUDA on Nvidian kehittämä rajapinta, joka on käytettävissä vain Nvidian grafiikkaprosessoreilla. OpenCL on Khronos Groupin kehittämä avoimen lähdekoodin ohjelmointirajapinta, jota voidaan hyödyntää monilla erilaisilla alustoilla kuten grafiikkaprosessoreilla, moniytimisillä keskusprosessoreilla sekä Cell-prosessoreilla [12]. DirectCompute on Microsoftin suunnittelema GPGPU rajapinta, jota voidaan käyttää erilaisilla grafiikkaprosessoreilla [21].

3.2. Toimintaperiaate

Grafiikkaprosessorien luontainen rinnakkainen arkkitehtuuri keskittyy prosessoimaan mahdollisimman monta threadia samanaikaisesti hieman hitaammin sen sijaan, että se prosessoisi yhden threadin mahdollisimman nopeasti. Massiivisella rinnakkaisuudella pyritään pääsemään yli viiveistä, joita syntyy laitteiden kommunikoinnista toistensa kanssa. [14]

Esimerkkinä toimii CUDA -ohjelmointi. CUDA -ohjelmointi käyttää sekä keskus-prosessoria että grafiikkaprosessoria. Keskusprosessori eli *host* siirtää dataa

grafiikkaprosessorille eli *devicelle* ohjelmointia varten. Kuvassa 6 näkyy, miten CUDA -prosessointi toimii laitetasolla.



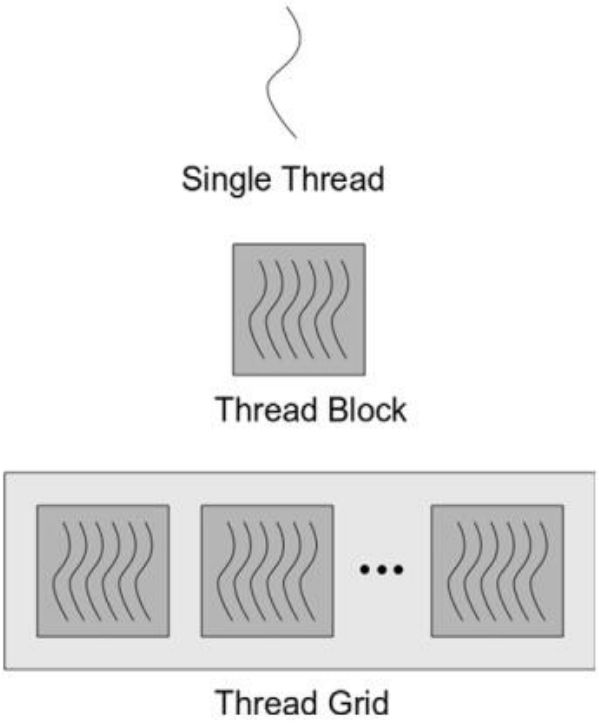
Kuva 6: *CUDA -prosessointi [14]*

Kuvassa 6 näkyy tiedonsiirto CUDA -prosessoinnissa. Ensimmäisessä vaiheessa data siirretään päämuistista grafiikkaprosessorin yleiseen muistiin. Toisessa vaiheessa keskusprosessori antaa ohjeet grafiikkaprosessorille. Kolmannessa vaiheessa tapahtuu varsinainen rinnakkainen prosessointi grafiikkaprosessorilla ja neljännessä vaiheessa tulokset siirretään jälleen päämuistiin grafiikkaprosessorin muistista.

Varsinainen rinnakkaisohjelmointi sisältää muutamia peruskomponentteja, jotka on esitelty kuvassa 7. Grafiikkaprosessoreilla suoritettavista funktioista käytetään nimeä kerneli. Thread on pienin yksikkö jonka käyttöjärjestelmä voi aikatauluttaa prosessoitavaksi. Threadien generointi ja aikataulutus on todella nopeaa grafiikkaprosessorilla. Se vie vain muutaman kellosyklin. Kaikista kernelin muodostamista threadeista muodostuu *gridejä*, joissa threadit ovat järjestyksessä *thread blockeissa*. [14]

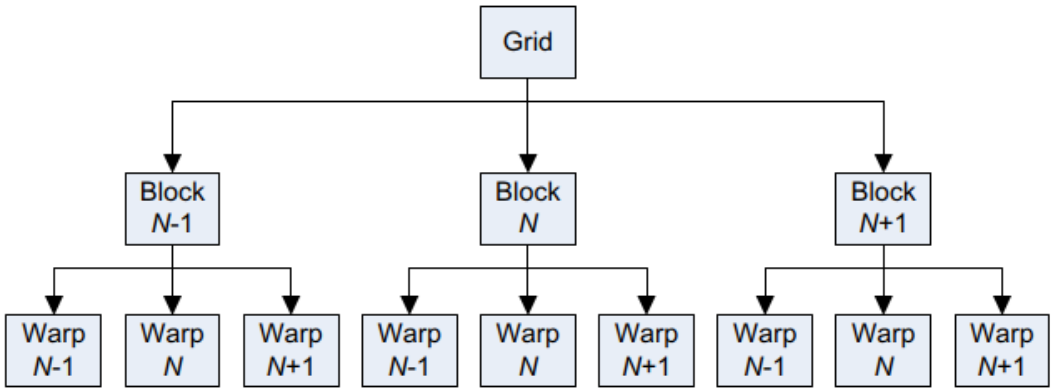
Lisäksi käytössä on vielä yksi tyyppi threadien järjestämiseen eli *warp*. Warpit ovat yksikkö threadien järjestämiseen streaming multiprosessoreissa. Multiprosessori voi prosessoida vain yhden warpin kerrallaan. Threadien lukumäärä warpin sisällä riippuu käytettävissä olevasta grafiikkaprosessorista. Warppien tarkoituksena on mahdollisimman tehokas hyötykäyttö laitteistolle. Esimerkiksi jos warpilla on pitkän

viiveen operaatio, josta se odottaa tuloksia ennen kuin se voi jatkaa, se laitetaan odottamaan ja toinen warp voi suorittaa laskentaa sillä aikaa. Tällä vältetään toimettomina olevia multiprosessoreita. Kun tulokset saapuvat, alkuperäinen warp voi taas jatkaa toimintaansa. Riittävällä warppien lukumäärällä multiprosessoreilla on todennäköisesti työkuormitusta jatkuvasti pitkän viiveen operaatioista riippumatta. Threadien lukumäärä thread blockeissa olisi hyvä olla jokin warpin sisältämän threadien lukumäärän kerroin. [14]



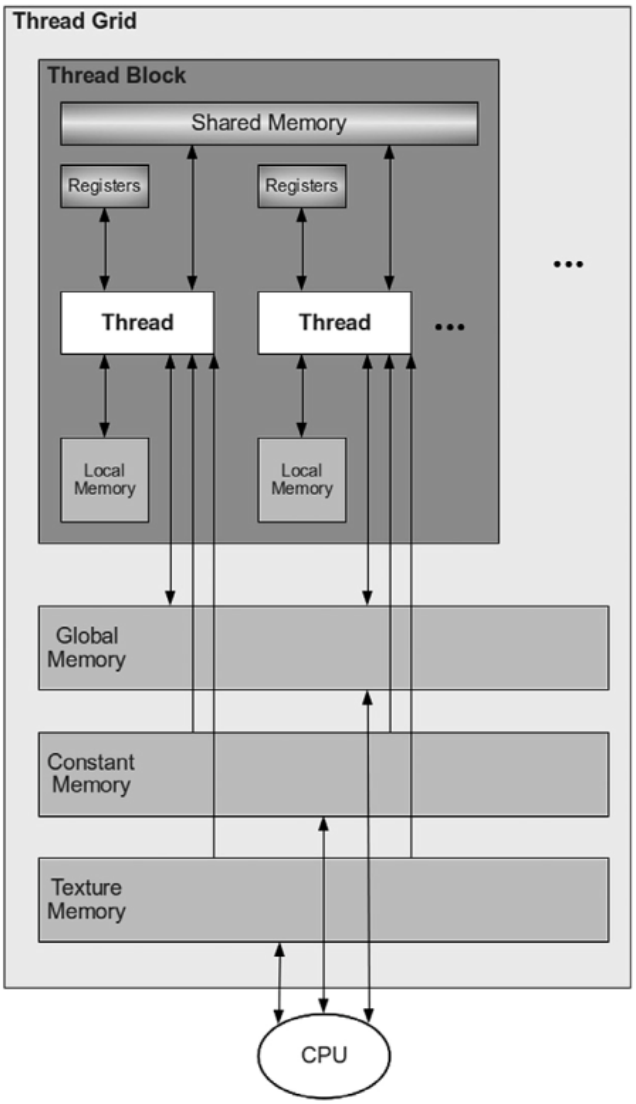
Kuva 7: *Rinnakkaisohjelmoinnin komponentit [14]*

Kuvassa 7 näkyy, kuinka threadit muodostavat thread blockeja ja blockeista muodostuu grid. CUDA -ohjelmointi on ideaalista erittäin rinnakkaisille (embarrassingly parallel) ongelmille. Englanninkielinen termi tulee siitä, että ongelman rinnakkaistamiseen on vaivatonta. Tällöin myös threadien tai blockien välistä kommunikointia ei tarvita paljoa. Threadien välinen kommunikointi onnistuu täsmällisillä primitiiveillä, mutta blockien välinen kommunikointi tarvitsee suorittaa kernelien kautta. [13] Kuvassa 8 näkyy rinnakkaisohjelmoinnin hierarkia.



Kuva 8: *Rinnakkaisohjelmoinnin hierarkia [13]*

CUDA -ohjelmointiin liittyy myös vahvasti muistit ja grafiikkaprosessorin muistityypeistä voidaan ohjelmoinnissa hyödyntää useimpia. Kuva 9 selventää muistien toimintaa ja vuorovaikutuksia.



Kuva 9: *CUDA -ohjelmoinnin muistimalli [14]*

Yleinen muisti hoitaa vuorovaikutuksen keskusprosessorin kanssa. Myös vakiomuistin kautta voi käydä vuorovaikutusta, mutta grafiikkaprosessori ei voi kirjoittaa siihen. Se kuitenkin tarjoaa nopeampia polkuja datan tavoittamiseen kernelien suorittamiseen kuin yleinen muisti. CUDA antaa ohjelmoijalle myös mahdollisuuden käyttää tekstuuriyksikköjä ja sen välimuistia. [14]

Lisäksi käytössä on muisteja, joiden käyttö pysyy blockin sisällä. Rekisterit ovat yksilöllisiä threadeille, eli jokainen thread voi käyttää vain omaa rekisteriään. Rekisterejä käytetään tyypillisesti säilyttämään muuttujia, joita thread tarvitsee, mutta joita sen ei tarvitse jakaa. Myös paikallinen muisti on yksilöllistä threadeille, mutta se on huomattavasti hitaampikäyttöinen. Tämän takia rekisterien täyttämistä kannattaa välttää, ettei dataa vuoda paikalliseen muistiin. Jaettu muisti (shared memory) on kohdennettu blockeille, joten jokainen thread tietyn blockin sisällä pääsee käyttämään jaetun muistin muistipaikkoja, jotka ovat kohdennettu juuri kyseiselle blockille. Jaettu muisti on yhtä nopeaa kuin rekisterien muisti. [14]

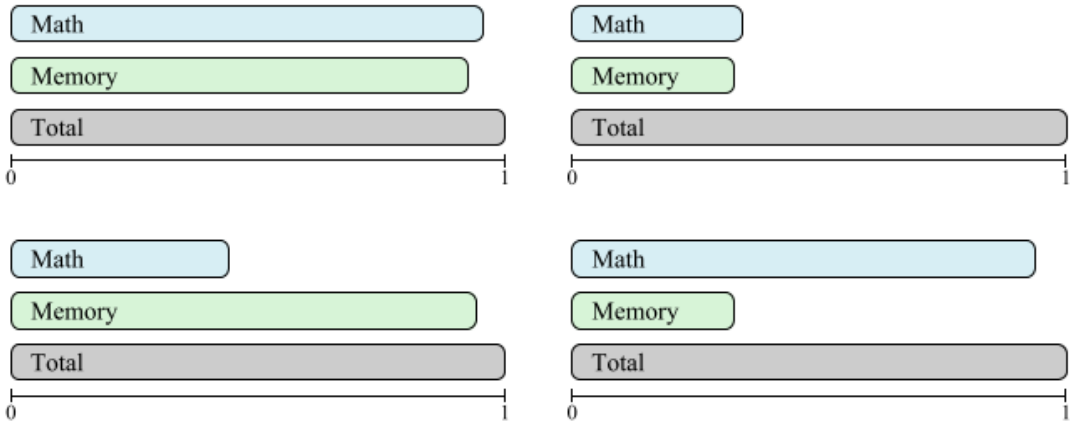
3.3. Haasteet

Usein voi olla helppoa hyödyntää grafiikkaprosessoria ja saada jonkinlainen nopeutus keskusprosessorin koodiin verrattuna. Ensimmäiset yritykset grafiikkaprosessorin ohjelmointiin eivät kuitenkaan usein ole kovin optimaalisia, eivätkä hyödynnä käytettävissä olevaa laitteistoa tarpeeksi tehokkaasti. Korkean suorituskyvyn koodin saavuttaminen, joka myös hyödyntää laitteistoa tehokkaasti, on hankalampaa. [18]

3.3.1. Pullonkaulat

Tietokoneella suoritettavalla laskennalla voi olla monen tyyppisiä pullonkauloja. Pullonkaula on laskentaa hidastava tekijä, jonka suorittaminen vie eniten aikaa ja täten hidastaa koko laskennan tuloksen saamista. Pullonkaulan paikantaminen voi olla vaikeaa keskusprosessorissa, mutta se voi olla vielä vaikeampaa grafiikkaprosessorissa [18].

Pullonkaulan tunnistamiseen on tärkeää käyttää sopivia suorituskyvyn mittareita, jonka jälkeen mitattua suorituskykyä verrataan teoreettiseen huippusuorituskykyyn. Kerneleille on olemassa 3 erilaista pullonkaulatyyppiä. Kernelin toimintaa voi rajoittaa joko laskentaohjeet, muisti tai viiveet. Pullonkaulana voi toimia myös keskusprosessorin ja grafiikkaprosessorin välinen tiedonsiirto tai suoritettavan sovelluksen toiminta. [18] Kuvassa 10 näkyy erilaiset pullonkaulatyypit, joita voi esiintyä kerneleissä.



Kuva 10: *Pullonkaulat kerneleissä [18]*

Kuvassa sinisellä on merkitty laskentaan kuluva aika, vihreällä muistioperaatioihin kuluva aika ja harmaalla on kernelin kokonaissuoritusaika. Ylhäällä vasemmalla on hyvin tasapainossa oleva kerneli, ylhäällä oikealla on viiveen rajoittama kerneli, alhaalla vasemmalla pullonkaulana toimii muistioperaatiot sekä alhaalla oikealla on laskennan rajoittama kerneli. [18]

Pullonkaulojen paikallistamisen voi tehdä kahdella tavalla. Joko käyttämällä CUDA *profileriä* tai muuttamalla lähdekoodia ja vertailemalla eri tavalla modifioitujen koodien suoritusaikoja. [18] CUDA profiler on työkalu, joka näyttää aikajanan keskusprosessorin ja grafiikkaprosessorin toiminnasta ja jossa on automaattinen analysointi optimisointimahdollisuuksille [22].

3.3.2. Optimointi

Vaikka korkean tason ohjelmointirajapintojen kuten CUDA:n ja OpenCL:n myötä grafiikkaprosessorien ohjelmoinnista on tullut helpompaa, silti ohjelmoijan tarvitsee optimoida koodia saavuttaakseen ideaalisen suorituskyvyn. Kuitenkin optimointisäännöt, jotka toimivat CUDA -ohjelmoinnissa Nvidian prosessoreilla, eivät välttämättä päde AMD:n prosessoreille ja OpenCL -ohjelmoinnille. [12] Optimointi on pullonkaulojen minimointia [23].

CUDA -ohjelmoinnille voidaan määrittää viisi optimisointisääntöä, jotka ovat [12]:

1. **Useiden threadien käyttö**. Jos threadeja on vähemmän kuin ytimiä, potentiaalista laskentatehoa ei käytetä kokonaan. Ytimien määrän ylittäminen auttaa piilottamaan viiveitä, kun odottavan threadin voi ottaa suorittamaan samalla, kuin toinen tarvitsee yleistä muistia.

2. **Sirulla olevan muistin käyttö**. Paikallisen muistin tai vakiomuistin järkevä käyttö vähentää yleisen muistin käyttökertoja lisäämättä kuitenkaan rekisterien käyttöä.
3. **Datan organisointi muistissa**. Lukeminen yleisestä muistista pitäisi olla yhteen sulautunutta. Samassa warpissa olevien threadien pitäisi päästä vierekkäisiin muistin elementteihin samanaikaisesti. Threadien pitäisi myös käyttää eri muistipankkeja (memory bank) jaetussa muistissa välttääkseen pankkikonflikteja (bank conflict). Muuten käytöstä tulee sarjamaista.
4. **Eriävien threadien minimointi**. Warpin sisällä olevien threadien tulisi seurata identtisiä toteutuspolkuja. Jos threadien ehdot eriävät, niin polkujen suorittamisesta tulee sarjamainen.
5. **Dynaamisten ohjeiden määrän vähentäminen**. Kernelin suoritusaika on suoraan verrannollinen sen suorittamien dynaamisten ohjeiden määrään. Ohjeiden määrän alhaisena pitäminen on ohjelmoijan vastuulla.

Koska Nvidian grafiikkaprosessorien optimointiin tarkoitetut säännökset eivät välttämättä päde AMD:n grafiikkaprosessoreille, seuraavat viisi optimointisääntöä ovat AMD:n prosessoreille, mutta eivät välttämättä päde Nvidian prosessoreille. [12]

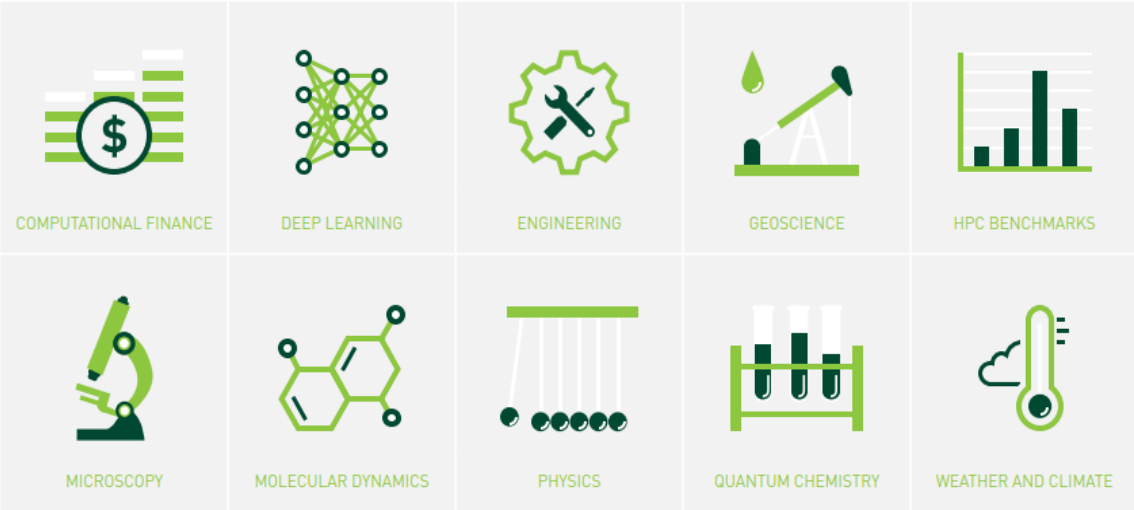
1. **Kernelien jakaminen (kernel splitting)**. Ehdot, joiden tulos voidaan ennustaa ennen kernelin alkua pitäisi siirtää grafiikkaprosessorin kernelistä keskusprosessorille ja jakaa kerneli kahteen osaan, joista molemmat ottavat eri haaran. Sen jälkeen keskusprosessori kutsuu kernelin, joka seuraa oikeaa haaraa.
2. **Local staging**. Paikallinen muisti ja vakiomuisti ovat AMD:n grafiikkaprosessoreilla huomattavasti hitaampikäyttöisiä kuin rekisterit. AMD:n prosessoreissa on kuitenkin enemmän rekisterejä kuin Nvidian prosessoreissa, joten ylimääräisten rekisterien käyttö on hyödyllistä.
3. **Vektorityypit**. OpenCL:ssä on sisäänrakennettuna erilaisia vektorityyppejä, joissa määritetään alkioiden tyyppi sekä määrä [24]. Esimerkiksi float3 vektorissa on 3 float-tyyppistä alkioita. AMD:n prosessoreiden muisti on optimoitu laskentaan 2-4 float -alkion vektoreilla. Silloinkin, kuin skalaareilla laskeminen on nopeampaa, muistin lataaminen float vektoreihin on tehokkaampaa kuin skalaarien lataaminen. Sen jälkeen voi poistaa silmukan, suorittaa laskennan skalaareina ja tallentaa tulokset jälleen vektoriin.

4. **Kuvamuisti**. Kuvamuisti tarjoaa paljon transformaatioita nopeuttaakseen pääsyä kuviin, mutta se pystyy myös tehokkaasti lukemaan mielivaltaista dataa float4 vektoreista.
5. **Optimointien yhdistäminen**. Jokainen optimointisääntö yksinään antaa jonkinlaisen nopeutuksen optimoimattomaan koodiin. Kaikkien optimointisääntöjen käyttäminen yhdessä ei kuitenkaan anna parasta mahdollista suorituskykyä. Jotkut optimointitavat toimivat paremmin toisten kanssa yhdessä ja yhteenlaskettu nopeutus hitaamman tavan kanssa voi olla suurempi kuin kahdella yksistään nopeammalla tavalla.

Kun ohjelman rinnakkaistaminen on suoritettu, ohjelmoija voi siirtyä optimoimaan toteutusta parhaan suorituskyvyn saamiseksi. Koska optimointiin on useita eri vaihtoehtoja, ymmärrys ohjelman tarpeista auttaa tekemään optimointiprosessista helpomman. Optimointi on kuitenkin iteratiivinen prosessi, eli kun huomataan mahdollisuus optimoinnille, käytetään ja koitetaan jotain tekniikkaa, katsotaan saatu nopeutus ja toistetaan testi. Tämä tarkoittaa myös, että ohjelmoijan ei myöskään tarvitse muistaa kaikkia optimointitekniikkoja, vaan niitä voi alkaa käyttää vähitellen, kun niitä oppii. [25]

4. SOVELLUSKOHTEET TIETEESSÄ

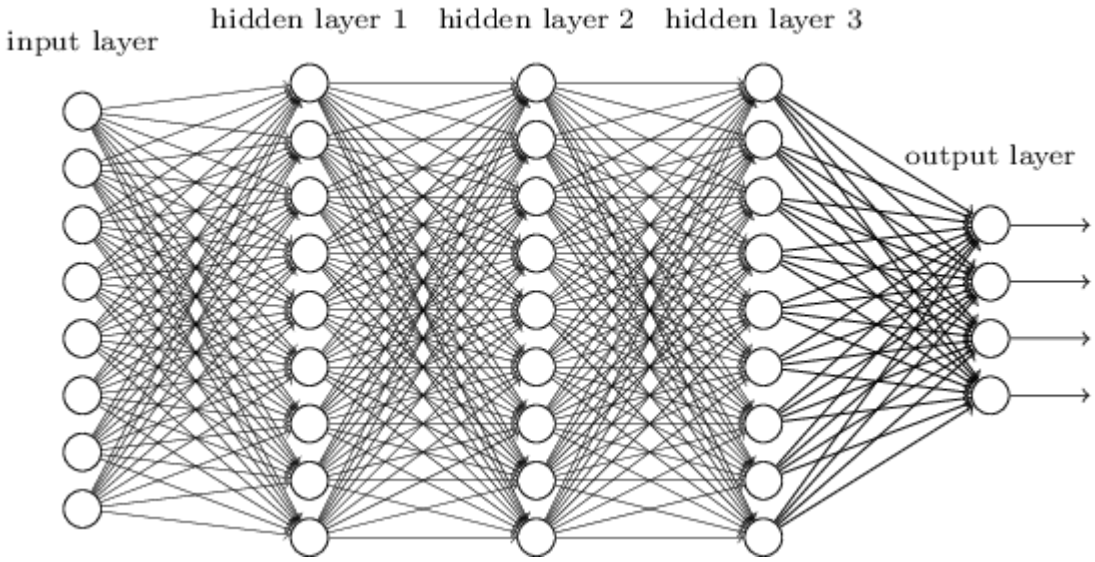
Grafiikkaprosessoreja voidaan hyödyntää monissa erilaisissa tieteenhaaroissa laskennan apuna. Sen avulla voi laskennan nopeuttamisen lisäksi myös vähentää virrankulutusta, servereiden määrää sekä saada merkittäviä rahallisia säästöjä [26, 27]. Myös syväoppimisella opetetun tekoälyn tarkkuus on parempi grafiikkaprosessorikiihdytetyllä laskennalla [27]. Kuvassa 11 näkyy tärkeimpiä sovelluskohteita grafiikkaprosessorin käytölle tieteen saralla.



Kuva 11: *Grafiikkaprosessorin käyttökohteita tieteellisessä laskennassa [26]*

Rahoitusmatematiikassa firmat tekevät muun muassa markkina- ja riskianalyysejä, jotka ovat laskennallisesti intensiivisiä. Syväoppiminen sekä tekniikan alan simulaatiot ja mallit ovat myös tärkeitä käyttökohteita. Geotieteen simulaatioilla paikannetaan öljyä ja kaasuja sekä tehdään geologisia mallinnuksia. Korkean suorituskyvyn (HPC, high-performance computing) vertailuarvot antavat viitteitä jonkin systeemin suorituskyvystä tuotannossa. Mikroskopia vaatii paljon laskentaresursseja tuotettujen kuvien analysointiin. Molekyylidynamiikkojen simulaatiossa kaikki applikaatiot ovat grafiikkaprosessorikiihdytettyjä, mikä tuo mahdollisuuden suorittaa simulaatioita, joita ei pystytä suorittamaan pelkällä keskusprosessorilla. Fysiikassa on myös simulaatioita esimerkiksi fuusiolle tai korkean energian partikkeleille, jossa grafiikkaprosessorikiihdytys tarjoaa uusia mahdollisuuksia. Kvanttikemian simulaatiot ovat avain uusien lääkkeiden ja materiaalien löytämiseen ja hyötyvät myös grafiikkaprosessoreista. Sään ennustamisella ja simuloinnilla pystytään pelastamaan ihmishenkiä äärimmäisten sääolosuhteiden ennustamisella sekä sitä käytetään hyväksi muun muassa lentoalalla. [26]

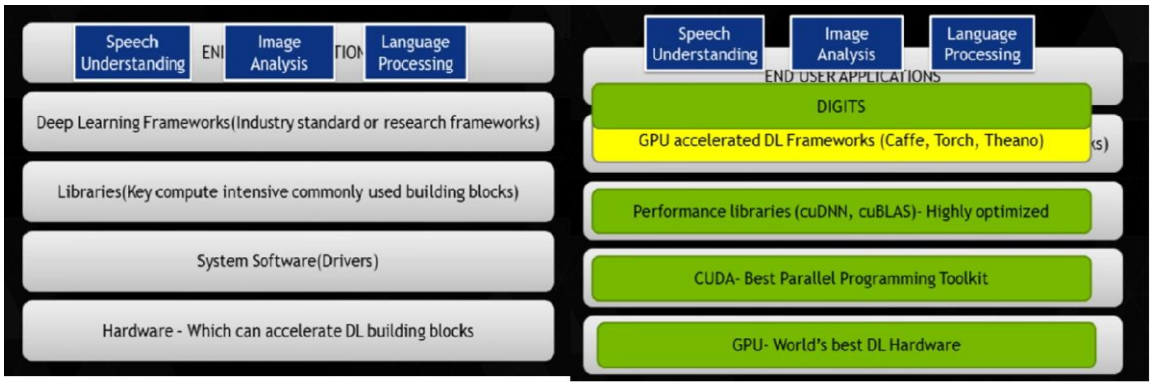
Syväoppiminen (Deep learning) on nykyään merkittävä tapa tekoälyn luomiseen. Siinä käytetään usean kerroksen neuroverkkoa esimerkiksi objektintunnistukseen tai puheen tunnistukseen. [28] Se myös kehittyy kovaa vauhtia kolmesta syystä. Massadataa (big data) on nykyään todella hyvin saatavissa, uusia tekniikoita neuroverkkojen luomiseen on saatavilla sekä grafiikkaprosessorien tekninen kehitys ja saatavuus markkinoilla mahdollistaa hyvän suorituskyvyn. [27] Kuvassa 12 näkyy esimerkki syvästä neuroverkosta.



Kuva 12: *Syvä neurovekko [29]*

Kuten kuvasta huomataan, on helppo nähdä, miksi grafiikkaprosessori sopii laskentaan neuroverkoilla. Neuroverkossa neuronien välisiä yhteyksiä on paljon ja niille suoritettava painokertoimien laskenta tuottaa tekoälyn. Ensiksi neuroverkolle syötetään dataa oppimista varten (training data), jonka jälkeen sen tarkkuutta voi testata (test data).

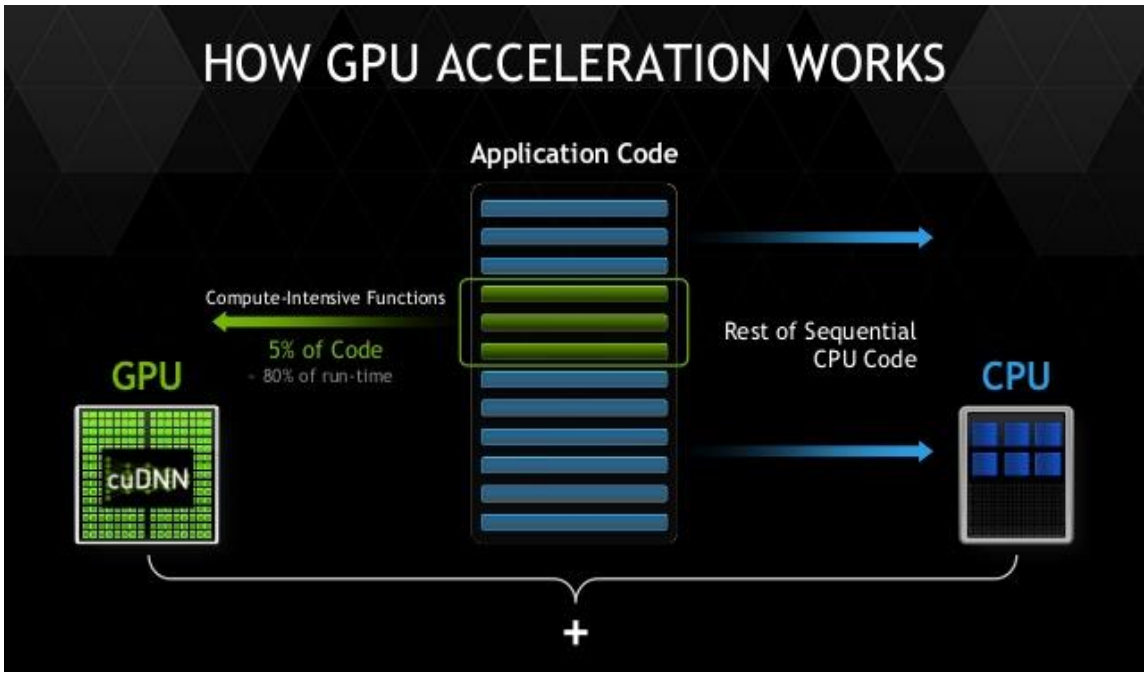
Grafiikkaprosessori sopii hyvin kiihdyttämään syväoppimista, koska neuroverkot ovat helposti rinnakkaistettavissa, niissä suoritetaan matriisioperaatioita sekä liukulukulaskentaa, mitkä sopivat grafiikkaprosessorille sekä niissä vaaditaan hyvää kaistanleveyttä, joka on grafiikkaprosessorilla tyypillisesti korkea. Grafiikkaprosessorin hyödyntäminen on yksi tärkeä palanen koko laskennassa, jossa on sekä laitteisto että sovelluskomponentteja. [27] Kuva 13 on esimerkki, jossa näkyy laskennan komponentit.



Kuva 13: *Tieteellisen laskennan komponentit ja esimerkkitapaus [27]*

Kuvassa vasemmalla näkyy vaadittavat komponentit laskennan suorittamiseen ja oikealla on esimerkkejä kyseisistä komponenteista. Alhaalla on kiihdyttävä laitteisto, joka syväoppimisesta puhuttaessa on grafiikkaprosessori. Sen yläpuolella on ohjelmisto, joka on CUDA Nvidian grafiikkaprosessoreille. Sen yläpuolella on kirjastoja syväoppimista varten, esimerkkinä cuDNN ja cuBLAS, jotka ovat optimoituja syväoppimiseen. Sen yläpuolella on ohjelmistokehykset esimerkiksi Caffe, Torch ja Theano. Kehykset toimivat ohjelmiston kautta ja käyttävät kirjastoja korkean suorituskyvyn saamiseksi [30,31]. Niiden yläpuolella on käyttäjän applikaatio eli vaikka puheentunnistus, kuvan analysointi tai kielen prosessointi.

Ohjelman kiihdyttäminen grafiikkaprosessorilla tehdään heterogeeniseen tyyliin ja vaatii tehtävien jakoa molemmille prosessorityypeille. Ohjelman koodista loppuen lopuksi melko pieni osa on sitä, joka jaetaan osiin ja suoritetaan rinnakkain. Se on kuitenkin laskennallisesti kaikkein vaativin osuus ja vie selvästi eniten aikaa. [27] Loppu sarjamainen koodi suoritetaan keskusprosessorilla. Kuva 14 visualisoi tilannetta.



Kuva 14: *Grafiikkaprosessorikiihdytys [27]*

Kuvasta näkyy, että laskennallisesti vaativin osuus ei ole koko koodista suuri osa, mutta se on kuitenkin selkeästi eniten aikaa vievä. Esimerkkitapaus on myös syväoppimiseen liittyvä, koska siinä käytetään myös cuDNN kirjastoa.

5. YHTEENVETO

Tässä työssä tutkittiin grafiikkaprosessorin soveltamista tieteelliseen laskentaan. Aluksi tutkittiin sen ominaisuuksia, josta saatiin kokonaiskuva, miksi grafiikkaprosessori olisi soveltuva tieteellisen laskennan sovelluksiin. Sitä vertailtiin myös tietokoneen toiseen prosessorityyppiin eli keskusprosessoriin ja tehtiin katsaus heterogeeniseen laskentaan, jossa molempien prosessorityyppien vahvuuksia käytetään hyödyksi. Tämän jälkeen käytiin läpi tekniset asiat grafiikkaprosessorin hyödyntämisestä. Ohjelmointi on nykyisin suuressa osassa tieteen sovelluksissa, eikä grafiikkaprosessorin käyttö poikkea tästä. Työssä käytiin läpi rinnakkaisohjelmointi, jolla on mahdollista hyödyntää grafiikkaprosessorin lukuisten ytimien koko prosessointivoima. Työssä myös tarkasteltiin mahdollisia hidastavia tekijöitä sekä mahdollisia ratkaisuja niihin. Tämän jälkeen tehtiin katsaus tieteen alan sovelluksiin, joissa grafiikkaprosessoria on mahdollista hyödyntää ja käytiin syväoppimista hieman tarkemmin läpi, josta sai paremman kuvan, mikä grafiikkaprosessorin osuus on itse laskennassa.

Grafiikkaprosessori on nykyaikana erittäin hyödyllinen työkalu myös tieteen saralla ja monet tieteen applikaatiot sekä supertietokoneet ovatkin heterogeenisiä. Myös tulevaisuudennäkymät ovat sen kaltaisia, että grafiikkaprosessorien merkitys tulee vain kasvamaan, koska nykyisin laskentatehoa on helpompi nostaa ytimiä lisäämällä kuin kellotaajuutta kasvattamalla. Laskentatehoa tarvitaan kuitenkin, koska tieteen sekä myös viihteen sovellukset ovat nykyisin todella vaativia laskennallisesti. Grafiikkaprosessori on myös vielä melko uusi prosessorityyppi ja rinnakkaisohjelmoinnin merkitys ja suosio on kasvamassa.

LÄHTEET

[1] J Nickolls, WJ Dally. The GPU Computing Era. IEEE Micro ;30(2) 2010 pp.56-69.

[2] Evolution of the GPU Device widely used in AI and Massive Parallel Processing. : IEEE; 2018.

[3] A white paper Peter N. Glaskowsky. NVIDIA's Fermi: The First Complete GPU Computing Architecture. 2009.

[4] Top 500 list. November 2018.
. Available at (viitattu 2.4.2019): https://www.top500.org/lists/2018/11/.

[5] The Best GPU Manufacturers, Ranked
. Available at (viitattu 10.4.2019): https://www.ranker.com/list/the-best-gpu-manufacturers/computer-hardware.

[6] JD Owens, M Houston, D Luebke, S Green, JE Stone, JC Phillips. GPU Computing. Proc IEEE ;96(5) 2008 pp.879-899.

[7] M Kenzel, B Kerbl, D Schmalstieg, M Steinberger. A high-performance software graphics pipeline architecture for the GPU. ACM Transactions on Graphics (TOG) ;37(4) 2018 pp.1-15.

[8] Kuva 1: The-graphics-pipeline-in-OpenGL-consists-of-these-5-steps-in-the-new-generation-of-cards.png. Available at (viitattu 12.4.2019): https://www.researchgate.net/profile/Christoph_Guetter/publication/235696712/figure/fig1/AS:299742132228097@1448475501091/The-graphics-pipeline-in-OpenGL-consists-of-these-5-steps-in-the-new-generation-of-cards.png.

[9] An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. : ACM; 2009.

[10] Nvidia. Nvidia Volta
. Available at (viitattu 12.4.2019): https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/.

[11] Nvidia. Nvidia Turing
. Available at (viitattu 12.4.2019): https://www.nvidia.com/fi-fi/geforce/turing/.

[12] Architecture-Aware Mapping and Optimization on a 1600-Core GPU. : IEEE; 2011.

[13] Cook S. Cuda Programming : A Developer's Guide to Parallel Computing with GPUs. Saint Louis: Elsevier Science & Technology; 2012.

[14] M Papadrakakis, G Stavroulakis, A Karatarakis. A new era in scientific computing: Domain decomposition methods in hybrid CPU–GPU architectures. Comput Methods Appl Mech Eng ;200(13) 2011 pp.1490-1508.

[15] CUDA programming. What is constant memory in CUDA
. 2012; Available at (viitattu 11.4.2019): http://cuda-programming.blogspot.com/2013/01/what-is-constant-memory-in-cuda.html.

[16] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro ;28(2) 2008 pp.39-55.

[17] Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. : ACM; 2010.

[18] AR Brodtkorb, TR Hagen, ML Sætra. Graphics processing unit (GPU) programming strategies and trends in GPU computing. Journal of Parallel and Distributed Computing ;73(1) 2013 pp.4-13.

[19] S Mittal, JS Vetter. A Survey of CPU-GPU Heterogeneous Computing Techniques. ACM Computing Surveys (CSUR) ;47(4) 2015 pp.1-35.

[20] Intel. Intel Hyper-Threading Technology
.

[21] Microsoft. Compute Shader Overview
. Available at (viitattu 10.5.2019): https://docs.microsoft.com/en-us/windows/desktop/direct3d11/direct3d-11-advanced-stages-compute-shader.

[22] Nvidia. Nvidia CUDA documentation. Profiler
. Available at (viitattu 18.5.2019): https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[23] P Wang, Nvidia. Fundamental Optimizations in CUDA
.

[24] Khronos Group. Vector Data Types
. Available at (viitattu 20.5.2019): https://www.khronos.org/registry/OpenCL/sdk/1.1/docs/man/xhtml/vectorDataTypes.html.

[25] Nvidia. Nvidia CUDA documentation. Best practices guide
. Available at (viitattu 20.5.2019): https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#optimizing-cuda-applications.

[26] Nvidia. Tesla V100 Performance Guide
. 2018.

[27] Nvidia. NVIDIA Deep Learning Course: Class #1 – Introduction to Deep Learning. 2015; Available at (viitattu 25.5.2019): https://www.youtube.com/watch?v=6eBpjEdgSm0.

[28] Nvidia. Deep Learning
. Available at (viitattu 25.5.2019): https://developer.nvidia.com/deep-learning.

[29] Neural networks and deep learning. tikz41.png. Available at (viitattu 25.5.2019): http://neuralnetworksanddeeplearning.com/images/tikz41.png.

[30] Nvidia. Deep Learning Software
. Available at (viitattu 25.5.2019): https://developer.nvidia.com/deep-learning-software.

[31] Nvidia. Deep Learning Frameworks
. Available at (viitattu 25.5.2019): https://developer.nvidia.com/deep-learning-frameworks.