Eetu Asikainen

Akustisen simuloinnin rinnakkaislaskentatoteutus

Insinööri (AMK)
Tieto- ja viestintätekniikka
Syksy 2019

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä: Asikainen Eetu

Työn nimi: Akustisen simuloinnin rinnakkaislaskentatoteutus

Tutkintonimike: Insinööri (AMK), tieto- ja viestintätekniikka

Asiasanat: rinnakkaislaskenta, näytönohjainlaskenta, tomografia

Tässä opinnäytetyössä tutkittiin näytönohjainten käyttöä yleiseen ohjelmointiin ja laskentaan. Yksi suurimmista saavutettavista hyödyistä tässä käytössä on huomattava laskennan nopeutuminen suuria datamääriä käsiteltäessä, kuten esimerkiksi tieteellisessä laskennassa.

Tavallinen nykyaikainen kuluttajille tarkoitettu mikroprosessori pystyy suorittamaan 2—16 tehtävää yhtä aikaa. Samaan aikaan hieman vanhemmallakin kuluttajille tarkoitetulla näytönohjaimella voidaan suorittaa satoja tai jopa tuhansia tehtäviä yhtä aikaa.

Työ aloitettiin tutkimalla kahta hallitsevaa näytönohjainten yleiskäyttöön tarkoitettua rajapintaa. Rajapinnan valinnan jälkeen laskennan kohteeksi valittiin ultraäänitomografia, jossa, tarkkuudesta riippuen, laskentaa suoritetaan satojen tuhansien tai jopa miljoonien elementtien suuruisilla matriiseilla.

Lopputuloksena saatiin valmistettua ohjelma, jolla voidaan simuloida ultraääniaaltojen etenemistä väliaineessa. Ohjelman alussa käyttäjä voi valita laitteeltaan haluamansa näytönohjaimen ja rajapinnan toteutuksen, mikäli laitteessa on näitä useampia. Simuloidut ultraäänisignaalit saadaan myös tallennettua ohjelman ulkopuolelle, jolloin käyttäjä voi tutkia näitä muilla ohjelmilla haluamallaan tavalla.

KAMK • University of Applied Sciences

Abstract

Author: Asikainen Eetu

Title of the Publication: Parallel Computing Implementation of Acoustical Simulation

Degree Title: Bachelor of Engineering, Information and Communication Technology

Keywords: parallel computing, GPGPU, tomography

In this thesis the usage of graphics processing units in generic programming and computation was investigated. One of the largest benefits with this method is the great speedup of computations with large batches of data, such as of those in scientific computations.

A regular, modern, consumer-grade CPU can execute 2 to 16 tasks at once. In contrast, with even a slightly older consumer-grade GPU hundreds or even thousands of tasks can be executed at the same time.

The work started with investigating the two dominant APIs meant for GPGPU. After selecting one of them, ultrasound tomography was selected as the computing task, where, depending on accuracy, the computations are done on matrices with hundreds of thousands or even millions of elements.

As the result a program was made, with which the propagation of ultrasound waves through a medium can be simulated. At the start of the program the user can select a GPU and API implementation, if their device has more than one of those. The simulated ultrasound signals can also be saved outside of the program, where the user can study them with other programs.

Sisällys

1 Johdanto 1

2 Laskenta näytönohjaimella 2

2.1 Miksi laskea näytönohjaimella 2

2.2 Hallitsevat näytönohjainrajapinnat 4

2.3 OpenCL 4

2.3.1 Isäntä ja laitteet 5

2.3.2 OpenCL ICD 5

2.3.3 Alusta ja laitteet 6

2.3.4 Konteksti ja komentojono 7

2.3.5 Indeksiavaruus 8

2.3.6 Muistinhallinta 9

2.3.7 Puskuriobjektit 11

2.3.8 Kernelien lähdekoodi 12

2.3.9 Ohjelmaobjektit 13

2.3.10 Kerneliobjektit 13

2.3.11 Tietojen hakeminen 15

3 Tomografia 16

3.1 Tomografia yleisellä tasolla 16

3.2 Ultraäänitomografia 17

4 Ultraäänen simulointi 18

5 Tulokset 19

5.1 Ohjelma 19

5.2 Simulointi 21

5.3 Suorituskyky 23

6 Yhteenveto 29

Lähteet 30

Symboliluettelo

API	Application Programming Interface
CPU	Central Processing Unit
CT scan	Computed Tomography scan, tietokonekerroskuvaus
CUDA	Compute Unified Device Architecture
GPGPU	General-Purpose computing on Graphics Processing Units
GPU	Graphics Processing Unit
ICD	Installable Client Driver, asennettava asiakasohjain
MRI	Magnetic Resonance Imaging, magneettikuvaus
NDRange	N-dimensional range
OpenCL	Open Computing Language
OpenCL C	OpenCL:n käyttämä C99:ään perustuva ohjelmointikieli
SIMD	Single Instruction, Multiple Data
SSE	Streaming SIMD Extensions

1 Johdanto

Tavallisessa nykyaikaisessa kuluttajille tarkoitetussa mikroprosessorissa on yleensä 2—8 ydintä, joista jokainen voi käsitellä yhtä tai kahta säiettä samanaikaisesti [1], [2], [3]. Jokainen säie suorittaa omaa prosessiaan ja näin ollen prosessori voi suorittaa 2—16 tehtävää yhtä aikaa. Työ- ja palvelinkäyttöön tarkoitetuissa prosessoreissa näitä suoritusytimiä voi olla huomattavasti enemmänkin [4], [5].

Laitteistosta, käyttötarkoituksesta ja suoritettavista prosesseista riippuen tietokone voi kyllä suorittaa satoja prosesseja näennäisesti yhtä aikaa ilman huomattavaa hidastumista, mutta näistä vain nämä muutamat ovat kulloinkin aktiivisessa suorituksessa. CPU:n aikatauluttaja valitsee, mitä prosessia minkäkin säikeen tulisi milloinkin suorittaa. Hieman vanhemmallakin näytönohjaimella voidaan puolestaan suorittaa satoja tai jopa tuhansia tehtäviä yhtä aikaa, sillä näytönohjaimet sisältävät satoja tai jopa tuhansia laskentaytimiä [6], [7], [8].

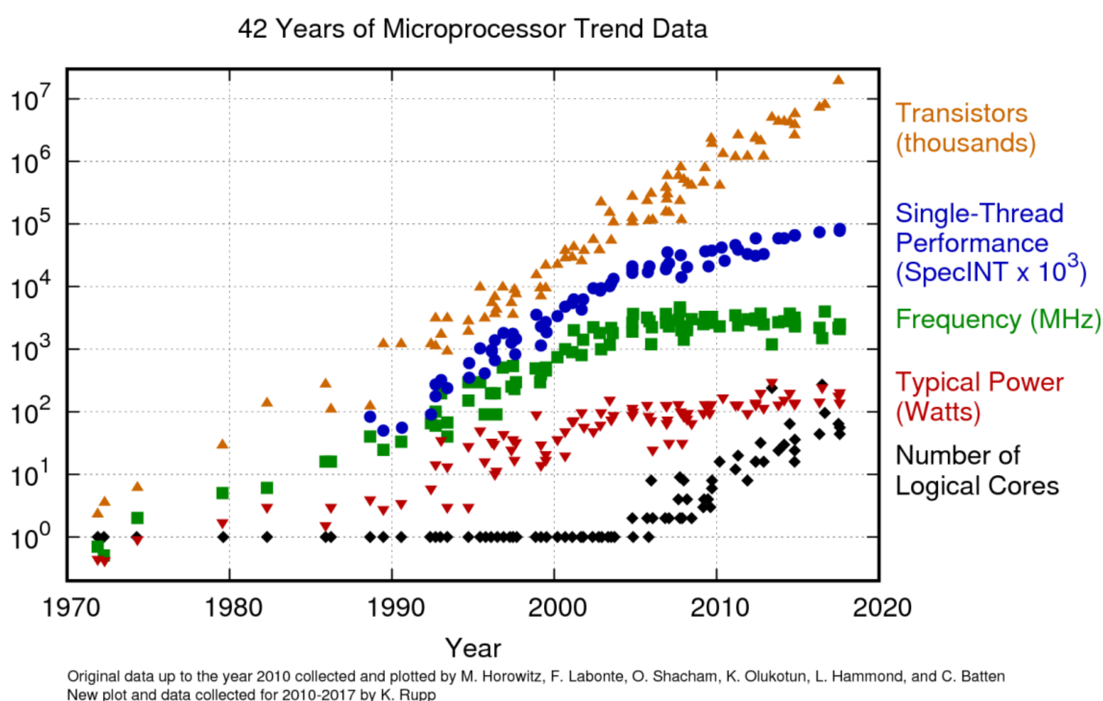
Näytönohjainten käyttäminen voi johtaa valtavaan suorituskyvyn parantumiseen tehtävissä, jotka voidaan rinnakkaistaa tehokkaasti. Tällaisissa tehtävissä esimerkiksi datan yksittäisen elementin arvo ei riipu muista saman datan elementeistä, eli laskentaa ei tarvitse suorittaa tietyssä järjestyksessä [9, s. 13—16]. Monet tieteellisen laskennan menetelmät hyödyntävät rinnakkaislaskentaa [10]. Suurissa määrissä tehtävästä rinnakkaislaskennasta voidaan myös käyttää termiä suurteholaskenta [11]. Tässä opinnäytetyössä rinnakkaistettavana tehtävänä käytettiin ultraäänitomografiaa. Lopputuloksena tuotetussa ohjelmassa suoritetaan konvoluutiota ja muita laskutoimituksia useamman, yli sadantuhannen elementin matriisin, välillä tuhansia kertoja jokaisessa ääniaallon simulaatiossa.

Näytönohjainten käyttämisestä yleiseen laskentaan pelkän grafiikan piirtämisen sijasta käytetään termiä GPGPU. Tämä opinnäytetyö aloitettiin tutkimalla kahta hallitsevaa GPGPU-rajapintaa: CUDAa ja OpenCL:ää. CUDA on Nvidian kehittämä rinnakkaislaskenta-alusta ja API heidän näytönohjaimilleen [12]. OpenCL on Khronos Groupin kehittämä avoin, rojaltivapaa standardi rinnakkaislaskentaan useilla erilaisilla mikroprosessoreilla ja laitteilla [13].

2 Laskenta näytönohjaimella

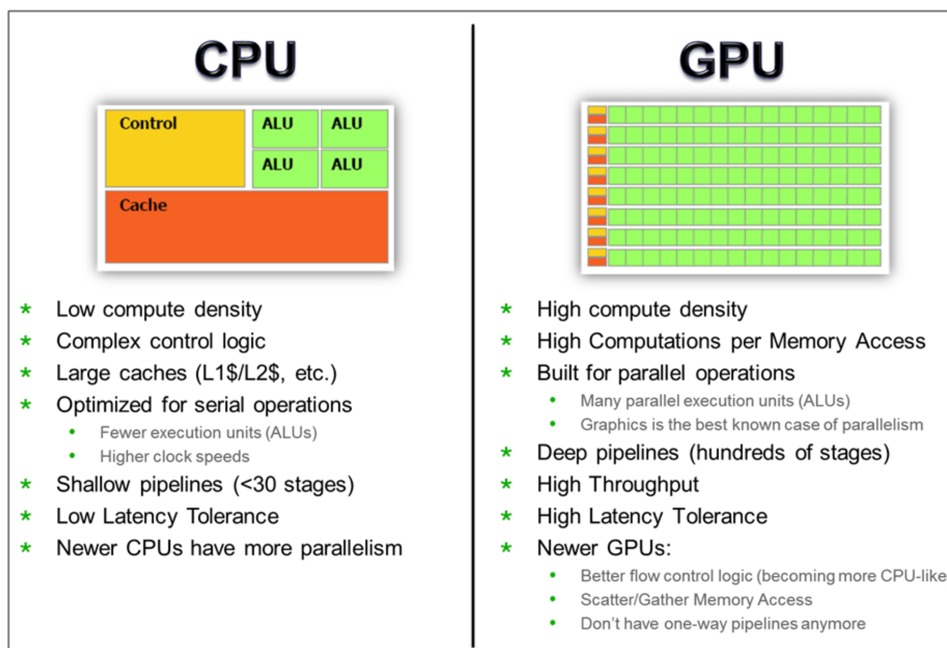
2.1 Miksi laskea näytönohjaimella

Tietotekniikassa on nautittu pitkään eksponentiaalisesta kasvusta monella osa-alueella. Vaikka transistorien määrä näyttää toistaiseksi jatkavan tällä linjalla, on prosessorien laskentaydinten kellotaajuus jäämässä 3—5 GHz:iin. Tietokoneiden suorituskyvyn parantamiseksi on 2000-luvulla alettu lisäämään laskentaytimien määrää samalla mikroprosessorilla. [14.] Kuva 1 näyttää tämän mikroprosessorien kehityksen.



Kuva 1. Mikroprosessorien eri osa-alueiden yleinen suuntaus 42 vuoden ajalta [14]. Transistorien määrä jatkaa kasvuaan, ytimien määrä on alkanut kasvaa vasta viimeisellä 15:lla vuodella ja muilla osa-alueilla kasvu on lähestulkoon pysähtynyt.

Mutta siinä, missä CPU:lla voidaan nykyään käyttää muutamaa ydintä ja toista kymmentä säiettä yhtäaikaisesti, voidaan GPU:lla käyttää satoja ytimiä ja tuhansia säikeitä [6], [7], [15]. Tämä johtuu näytönohjainten rakenteellisista eroista tavallisiin prosessoreihin verrattuna, joita on listattu kuvassa 2. Näytönohjaimet ovat kehittyneet pääasiassa videopelien piirtämistä varten, jotka ovat vaatineet samojen operaatioiden suoritusta suurille datamäärille mahdollisimman nopeasti. Tästä syystä näytönohjaimilla saadaan — sopivia algoritmeja käytettäessä — suoritettua laskentaa huomattavasti nopeammin kuin tavallisilla prosessoreilla [10], [16].



Kuva 2. CPU:n ja GPU:n toimintaperiaatteet [15]. CPU:t voivat suorittaa vain muutamaa tehtävää kerrallaan, mutta voivat hyppiä suorituskäskyjen välillä helpommin. GPU:t voivat suorittaa valtavia määriä laskentaoperaatioita rinnakkain, mutta niiden kontrollilogiikka on jokseenkin rajoittunutta.

Tietotekniikka on kuitenkin täynnä kompromisseja, ja näytönohjainten heikkouksiin kuuluvat esimerkiksi joissain määrin yksinkertaisempi kontrollilogiikka ja tiedonsiirtoon kuluva aika CPU:n kanssa. Tiedonsiirron hitaus on samankaltainen ongelma kuin CPU:lla: CPU - RAM tiedonsiirto on erittäin hidasta verrattuna CPU:n omaan välimuistiin. CPU - GPU tiedonsiirto kulkee pääasiassa PCIe-väylää pitkin, jonka nopeus riippuu CPU:n PCIe-versiotuesta, emolevyn PCIe-väylän versiosta sekä sen toimintanopeudesta (x1, x2, x4, x8, x16). Taulukossa 1 nähdään tiedonsiirron nopeuksia RAM- ja PCIe-väylissä.

Väylä	Kaistanleveys (gigatavua sekunnissa)
DDR3 (RAM)	6.25 - 16.67
DDR4 (RAM)	12.5 - 25
PCIe 3.0 (GPU)	1 - 16
PCIe 4.0 (GPU)	2 - 32

Taulukko 1. Tiedonsiirtojen nopeuksia [17], [18], [19], [20, s. 16].

2.2 Hallitsevat näytönohjainrajapinnat

Näytönohjainten muuttumisen pelkästä grafiikan käsittelystä yleiskäyttöiseen laskentaan sopiviksi voidaan sanota alkaneen vuosituhannen taitteen jälkeen, kun näytönohjaimet alkoivat tukea liukulukuja ja varjostimia. Varjostimilla pystyttiin ohjelmoimaan muuttuvaa logiikkaa näytönohjainten aiemmin hyvin rajoittuneiden ohjelmakäskyjen sekaan. Näytönohjainten yleiskäyttö helpottui, kun tekselit, fragmentit ja pikselit vaihtuivat säikeisiin, vektoreihin ja jaettuihin muistialueisiin CUDAn julkaisun myötä. Tätä seurasivat OpenCL ja DirectCompute. [21. s. 1—2] Näistä kolmesta DirectCompute on ns. laskentavarjostin ja se käyttää HLSL-varjostinkieltä muiden käyttäessä C-tyylisiä ohjelmointikieliä [22].

Työn tekeminen aloitettiin tutkimalla kahta alaa hallitsevaa näytönohjainrajapintaa: Nvidian CUDAa ja Khronos Groupin OpenCL:ää.

CUDA on Nvidian luoma rinnakkaislaskentaohjelmointirajapinta, joka toimii ainoastaan heidän näytönohjaimillaan. Tästä rajatusta laitearkkitehtuurista johtuen CUDAn suorituskyky voi olla joissakin tehtävissä parempi kuin OpenCL:n [23, s. 11—16], [24, s. 9—12].

OpenCL on Khronos Groupin luoma rinnakkaisohjelmointiin tarkoitettu viitekehys. CUDAsta poiketen OpenCL:ää voi käyttää useilla eri laitteilla, esim. CPU, GPU, DSP ja FPGA. Valtavasta valikoimasta johtuen alustavalmistajat tekevät yleensä itse omat OpenCL-toteutuksensa.

Työn tekoon valittiin rajapintojen tutkinnan ja muutamien kokeilujen jälkeen OpenCL. Päätöstä puolsi merkittävästi tämän alustariippumattomuus.

2.3 OpenCL

Esityksen OpenCL:stä kehitti Apple yhdessä AMD:n, Nvidian, Intelin, IBM:n ja Qualcommin kanssa. Tämä esitys annettiin kesäkuussa 2008 Khronos Groupille, joka teki esityksen pohjalta puolessa vuodessa virallisen spesifikaation. OpenCL:n tarkoitus on tehdä alustariippumattomasta laitteistokiihdytetystä rinnakkaislaskennasta yksinkertaisempaa.

OpenCL itsessään ei ole rajapinta, vaan viitekehys, joka määrittelee C- ja C++-rajapinnat sekä OpenCL C-ohjelmointikielen. OpenCL:n standardin mukainen toteutus sisältää OpenCL C-kääntä-

jän halutuille laitteille sekä ohjelmakirjaston, joka sisältää toteutuksen C-rajapinnan ja kohdelaitteiden välille. OpenCL C on C99:ään pohjautuva ohjelmointikieli, jolla kirjoitetaan OpenCL:n rinnakkain suoritettavat funktiot: kernelit. Näistä lisää kappaleessa 2.3.10.

C++-rajapinta kulkee cl2.hpp-tiedoston muodossa ja kätkee C-rajapinnan vapaat funktiot olio-ohjelmoinnin mukaisiin luokkiin ja olioihin. Tämä yksinkertaistaa monia toimintoja ja automatisoi suuren osan muistinhallinnasta ja objektien elinajasta. Tästä syystä tämän dokumentin koodiesimerkit ovat C++-rajapinnan mukaisia. Seuraavien kappaleiden funktiosignatuurien kuvat on otettu cl2.hpp:n dokumentaatiosta [25].

2.3.1 Isäntä ja laitteet

OpenCL käyttää termejä isäntä (host) ja laite (device) toimintansa jaottelussa. Isäntä on lähes aina CPU, joka kertoo OpenCL-laitteille, mitä niiden tulee tehdä. Laite tai laitteet voivat olla muita prosessoreita, näytönohjaimia ynnä muita mikropiirejä. Isännällä ajettava ohjelma kysyy OpenCL:n APIn kautta järjestelmältä, mitä OpenCL:ää tukevia laitteita järjestelmässä on. Isäntä valitsee haluamansa laitteet, luo kontekstit ja komentojonot, kääntää näille suoritettavat funktiot (kernelit) ja lisää käskyjä komentojonoihin suoritettaviksi.

Kernelit vastaavat suurissa määrin algoritmien sisempiä silmukoita, eli ne ovat funktioita, joissa tapahtuu algoritmin varsinainen laskentatehtävä. Ulommat silmukat, eli suoritusmäärät, määräytyvät käytettävän indeksiavaruuden koon mukaan. Tästä lisää kappaleessa 2.3.5. Huomattavaa on, että kernelit käännetään kohdealustalleen emäohjelman suorituksen aikana. Khronos Group ei kuitenkaan itse voisi mitenkään tukea kaikkia mahdollisia alustoja, joten alustariippumattomuus tapahtuu OpenCL ICD:n kautta.

2.3.2 OpenCL ICD

OpenCL ICD on ratkaisu, jolla sama lähdekoodi saadaan käännettyä lähes mille tahansa alustalle. Tämä mahdollistaa myös useamman yhtäaikaisen OpenCL-toteutuksen olemassaolon samalla alustalla. Kun isäntäohjelma pyytää tietoja laitteiston OpenCL-tuesta, ICD käy läpi alustan rekisterin (Windows) tai tiettyjä hakemistoja (Linux) etsien asennettuja OpenCL-toteutuksia. Isäntäohjelma voi täten valita käyttöönsä yhden tai useamman löydetyistä toteutuksista. Tämä prosessi

tapahtuu automaattisesti, kun ohjelma on linkitetty OpenCL:n dynaamiseen kirjastoon (OpenCL.dll Windowsilla, libOpenCL.so Linuxilla) ja kysyy alustatietoja. Jos käytetään useampaa toteutusta yhtä aikaa, isäntäohjelman vastuulla on laitteiden välisen tiedonsiirron ja tilan synkronointi.

2.3.3 Alusta ja laitteet

OpenCL:n käyttäminen alkaa hakemalla lista laitteiston OpenCL:ää tukevista alustoista kuvan 3 mukaisella funktiokutsulla. Nämä alustat vastaavat käyttöjärjestelmälle asennettuja laitteiston ajureita. Alustaobjektista voidaan hakea niiden tukemia laitteita kuvan 4 metodilla. Laitteet vastaavat käytettävän tietokoneen fyysisiä komponentteja, joilla laskenta tullaan suorittamaan. Näitä laitteita käytetään seuraavassa kappaleessa kontekstin ja komentojonon luomiseen.

```
static cl_int cl::Platform::get ( vector< Platform > *  platforms )
```

Gets a list of available platforms.

Wraps clGetPlatformIDs().

Kuva 3. OpenCL-alustojen hakeminen. Parametri ”platforms” on ns. out parameter, eli siihen säilötään löydetyt alustat. Varsinainen paluuarvo sisältää mahdollisen virhekoodin, jos funktio epäonnistui.

```
cl_int cl::Platform::getDevices ( cl_device_type       type,
                                  vector< Device > *   devices
                                )                      const
```

Gets a list of devices for this platform.

Wraps clGetDeviceIDs().

Kuva 4. OpenCL-laitteiden hakeminen. Parametri ”type” kertoo, minkä tyyppisiä laitteita halutaan etsiä. Näitä voivat olla CPU:t, GPU:t, kiihdyttimet, alustan oletuslaitteet tai kaikki laitteet. Funktio säilöö löydetyt laitteet parametriin ”devices”. Varsinainen paluuarvo ilmaisee mahdollisen virhekoodin.

2.3.4 Konteksti ja komentojono

Seuraavaksi täytyy määritellä OpenCL-konteksti. Konteksteilla hallinnoidaan muun muassa komentojonoja, muistiobjekteja ja kerneleitä. Konteksti tehdään kuvan 5 mukaisesti cl::Context-luokan konstruktorilla.

Kontekstin pohjalta luodaan komentojono, joka hallinnoi kernelien suorituksia kontekstiin sidotuilla laitteilla. Isäntä asettaa komentojonoon suoritettavia käskyjä. Näitä ovat esimerkiksi:

- **Kernelien suorituskomennot**: Isäntä käskee tietyn kernelin suoritettavaksi.
- **Muistikomennot**: Datan siirto laskentayksiköille, laskentayksiköiden välillä tai niiltä isännälle. Puskuriobjektien käyttöön- ja poisotto.
- **Synkronointikomennot**: Komentojen suoritusjärjestyksen rajoitus.

Komentojono aikatauluttaa komennot laitteille. Asetettaessa komentoja jonoon voidaan käyttää blokkaavaa tai blokkaamatonta versiota: blokkaavan version funktiokutsu palautuu isäntäohjelmalle vasta, kun annettu komento on suoritettu. Blokkaamaton versio lähettää komennon laitteille ja palautuu välittömästi isännälle, joka voi komentojen suorituksen aikana tehdä muita asioita. Komennot voidaan myös määritellä suoritettaviksi kahdessa eri tilassa:

- **Järjestyksessä**: Komennot suoritetaan samassa järjestyksessä kuin ne tulevat komentojonoon, ja komentojen suoritusta odotetaan.
- **Vapaassa järjestyksessä**: Komentoja suoritetaan samassa järjestyksessä kuin ne tulevat komentojonoon, mutta komentojen suoritusta ei jäädä odottamaan. Näin ollen useampi pieni ja nopea komento voidaan suorittaa samaan aikaan, kun yhtä suurempaa ja hitaampaa komentoa suoritetaan.

```
cl::Context::Context ( const vector< Device > &                                              devices,
                       cl_context_properties *                                               properties = NULL,
                       void(CL_CALLBACK *notifyFptr)( const char *, const void *, size_type, void *) = NULL,
                       void *                                                                data = NULL,
                       cl_int *                                                              err = NULL
                     )
```

Constructs a context including a list of specified devices.

Wraps clCreateContext().

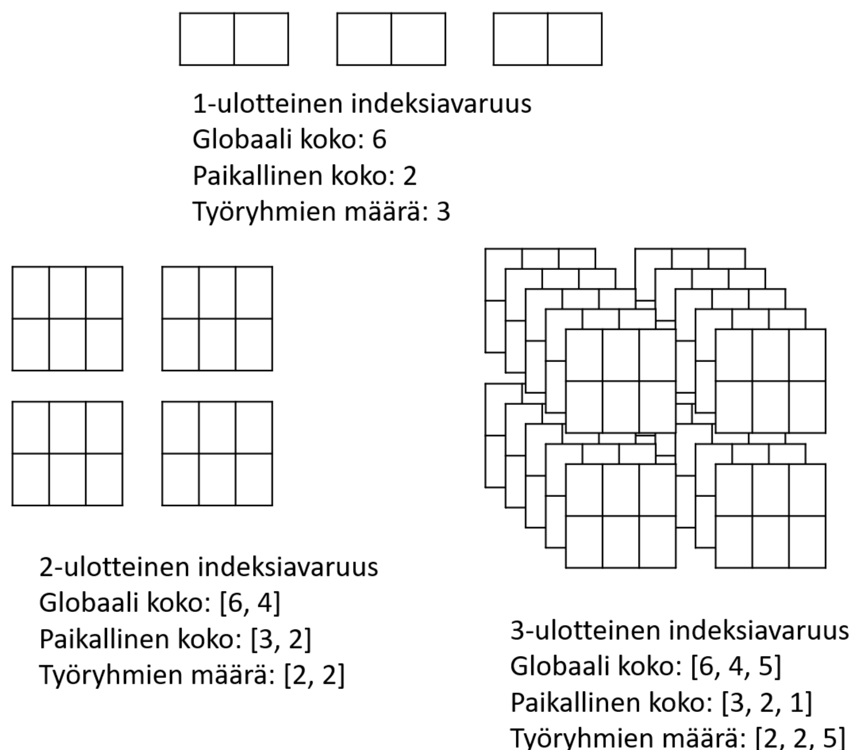
Kuva 5. Yksi useista cl::Context-luokan konstruktoreista.

2.3.5 Indeksiavaruus

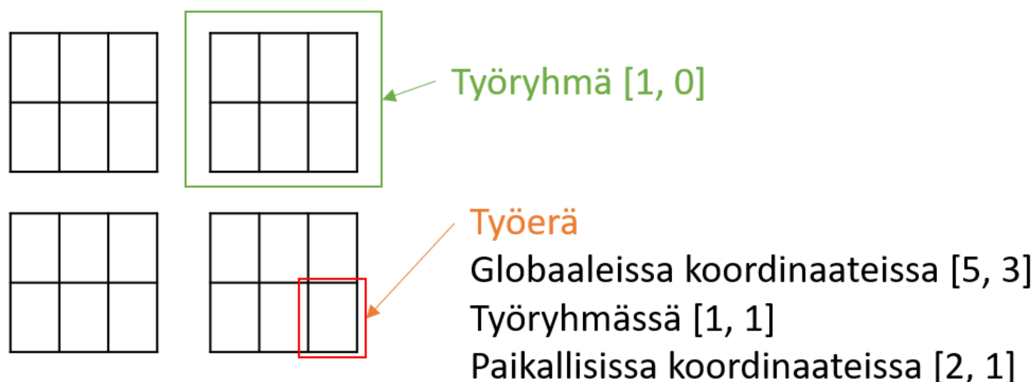
Kun isäntäohjelma lähettää kernelin suoritettavaksi, suoritukselle määritellään indeksiavaruus, joka kulkee OpenCL:ssä nimellä NDRange. Tämä on N-ulotteinen indeksiavaruus, jossa N on joko 1, 2 tai 3. Kuvassa 6 nähdään esimerkit kaikista indeksiavaruuksista.

Kerneli suoritetaan jokaisessa indeksiavaruuden pisteessä. Yksittäiset suorituspisteet kulkevat nimellä työerä (work-item). Kaikki työerät suorittavat saman kernelin, mutta suoritetut käskyt voivat vaihdella työerien välillä esimerkiksi kernelin ehtologiikan takia. Työerät voidaan myös jaotella suurempiin kokonaisuuksiin, työryhmiin (work-group). Työryhmän koosta käytetään nimitystä paikallinen koko. Työryhmän kernelit suoritetaan joko rinnakkain useilla laskentayksiköillä tai samanaikaisesti yhdellä laskentayksiköllä [26, s. 172].

OpenCL:ssä työerien ja -ryhmien tunnisteet sekä globaalit tunnisteet ovat N-ulotteisia taulukoita, jotka vastaavat indeksiavaruuden sijainteja. Indeksointi alkaa nollasta ja ulottuvuudet ovat taulukossa järjestyksessä X, Y, Z. Yksittäinen työerä voidaan yksilöidä joko työryhmän sijainnin ja paikallisen sijainnin tai pelkän globaalin sijainnin perusteella. Kuvassa 7 nähdään esimerkkejä tästä. Tätä yksilöintiä voi käyttää esimerkiksi kernelin parametreinä tulevien taulukoiden indeksointiin. Tästä lisää kappaleessa 2.3.8.



Kuva 6. Esimerkkejä indeksiavaruuksista.



2-ulotteinen indeksiavaruus
Globaali koko: [6, 4]
Paikallinen koko: [3, 2]
Työryhmien määrä: [2, 2]

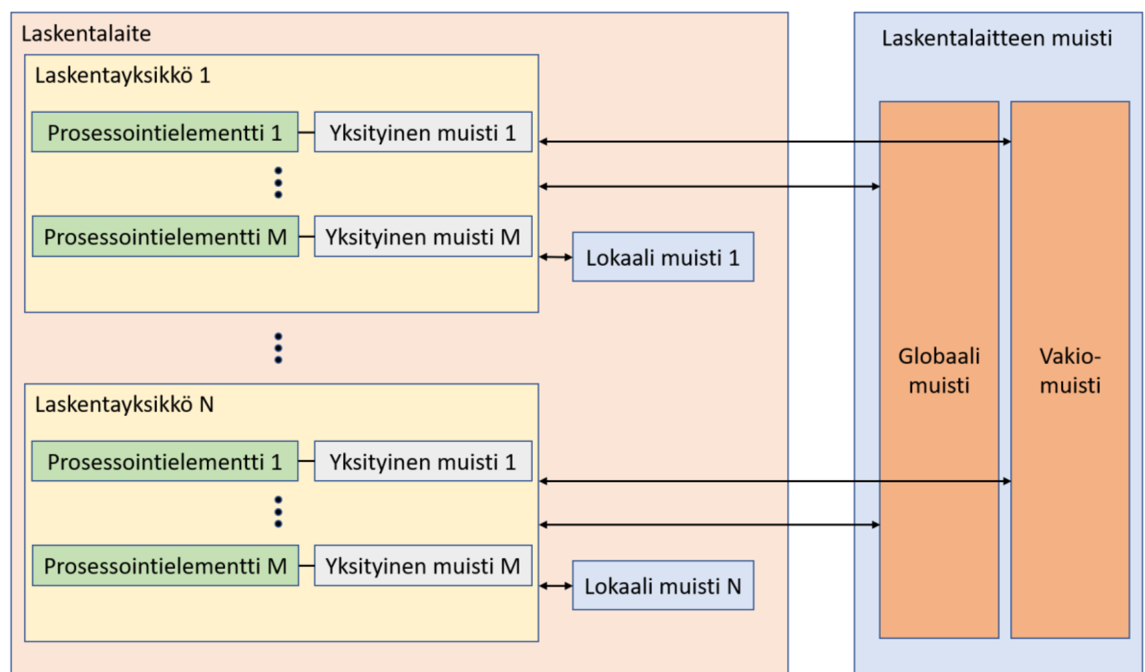
Kuva 7. Indeksiavaruuden koostumus. Indeksit alkavat nollasta ja ovat järjestyksessä [X, Y]. X-akseli kasvaa vasemmalta oikealle ja Y-akseli ylhäältä alas. Z-ulottuvuuden koko on tässä yksi ja sen indeksi on nolla.

2.3.6 Muistinhallinta

Jokaisella työerällä on pääsy neljään muistialueeseen, jotka on listattu taulukossa 2. Näillä hallinnoidaan ketkä voivat lukea ja kirjoittaa muistialueisiin. Nämä muistialueet sisältävät pääasiassa irrallisia muuttujia ja puskuriobjekteja, jotka esitellään seuraavassa kappaleessa. Nämä objektit ja muuttujat ovat pääasiassa kerneliobjektien argumentteja, joista lisää kappaleessa 2.3.10. OpenCL:n muistimallin rakenteen graafinen ilmaisu näkyy kuvassa 8. Laitteen fyysinen muistirakenne voi poiketa tästä, mutta OpenCL-toteutuksen tulee toimia kuvan mukaisesti.

Muistialue	Näkyvyys	Isännän käyttöoikeudet	Laitteen käyttöoikeudet
Globaali muisti	Kaikille tässä indeksiavaruudessa	Luku + kirjoitus	Luku + Kirjoitus
Vakiomuisti	Kaikille tässä indeksiavaruudessa	Kirjoitus	Luku
Lokaali muisti	Kaikille tässä työryhmässä.	Ei oikeuksia	Luku + kirjoitus
Yksityinen muisti	Vain tälle työerälle	Ei oikeuksia	Luku + kirjoitus

Taulukko 2. OpenCL:n muistialueet, niiden näkyvyys ja käyttöoikeudet [26, s. 224—227], [27].



Kuva 8. OpenCL:n muistimalli. Jokainen laskentayksikkö näkee globaalin muistin sekä vakiomuistin. Jokaisella laskentayksiköllä on oma lokaali muisti ja jokaisella prosessointielementillä on oma yksityinen muisti.

2.3.7 Puskuriobjektit

Datan siirtäminen CPU:lta OpenCL:n käyttämille laitteille ja takaisin tapahtuu puskuriobjektien (buffer objects) kautta. Nämä ovat pääasiassa osoittimia edellisen kappaleen muistiosioihin. Puskuria luodessa määritellään missä kontekstissa se toimii, kuinka paljon muistia se vie ja kuinka sitä voidaan käyttää. Tärkeimmät asetukset puskurin luomiseen löytyvät taulukosta 3.

Asetus	**Toiminta**
CL_MEM_READ_WRITE	Kerneli voi lukea ja kirjoittaa tähän puskuriin.
CL_MEM_WRITE_ONLY	Kerneli ainoastaan kirjoittaa tähän puskuriin.
CL_MEM_READ_ONLY	Kerneli ainoastaan lukee tätä puskuria.
CL_MEM_USE_HOST_PTR	Käyttäjän antamaa osoitinta käytetään muistin sijaintina.
CL_MEM_COPY_HOST_PTR	Käyttäjän antaman osoittimen sisältö kopioidaan laitteen muistiin.

Taulukko 3. Puskuriobjektien käyttöoikeuksien tärkeimmät asetukset.

Puskuriobjektit ovat cl::Buffer-luokan olioita. Kuvassa 9 nähdään yksi näistä konstruktoreista. Kaikkia edellisen taulukon asetuksia ei tarvitse eikä voi käyttää yhtä aikaa. Oletuksena puskurit käyttävät arvoa CL_MEM_READ_WRITE. CL_MEM_COPY_HOST_PTR kopioi datan käyttäjän antamasta osoittimesta laitteelle, jonka jälkeen tämä käyttäjän antama muistialue voidaan vapauttaa. CL_MEM_USE_HOST_PTR puolestaan käyttää käyttäjän antamaa osoitinta muistin sijaintina ja OpenCL-implementaatiot voivat säilöä tämän datan tilan laitteen muistiin. Tämän käyttäminen kuitenkin aiheutti tehdyn ohjelman kehitysversiossa suuria viiveitä, kun datan tila ilmeisesti synkronoitiin jokaisella päivityskerralla takaisin isäntäkoneelle. Puskurien sisältöä voidaan myös muokata luomisen jälkeen kirjoitusmetodeilla ja niiden sisältö voidaan lukea takaisin isännän antamaan muistiosoitteeseen lukumetodeilla.

```
cl::Buffer::Buffer ( const Context &  context,
                     cl_mem_flags     flags,
                     size_type        size,
                     void *           host_ptr = NULL,
                     cl_int *         err = NULL
                   )
```

Constructs a **Buffer** in a specified context.

Wraps clCreateBuffer().

Parameters

host_ptr Storage to be used if the CL_MEM_USE_HOST_PTR flag was specified. Note alignment & exclusivity requirements.

Kuva 9. Yksi puskuriobjektin konstruktoreista. Parametreinä tämä ottaa kontekstin, muistinkäytön asetukset, muistin määrän tavuina ja osoittimet mahdollisesti valmiiseen dataan ja virhekoodiin.

2.3.8 Kernelien lähdekoodi

Koska OpenCL C pohjautuu C99:ään, ovat OpenCL-kernelit kuin C-funktioita muutamalla lisämäärityksellä ja erikoisrajoituksella. Tärkeimpiä näistä ovat:

- Määrittely aloitetaan avainsanalla __kernel.
- Palautustyyppi täytyy olla void. Paluuarvot tapahtuvat osoittimien kautta. (ns. out parameter)
- Osoitinmuuttujat täytyy määritellä joko __global, __constant tai __local määrityksellä. Nämä vastaavat suoraan kappaleen 2.3.6 globaalia-, vakio- ja lokaalia muistia.
- C99:n standardikirjaston tiedostoja ei voi sisällyttää eikä niiden funktioita käyttää.

OpenCL määrittelee myös useita funktioita, joita voidaan kutsua kerneleissä. Näitä ovat monet matemaattiset funktiot, sekä kernelin yksilöitiin indeksiavaruudessa (kappale 2.3.5) käytettävät funktiot.

Kuvassa 10 nähdään esimerkki kernelistä, joka laskee yhteen taulukoiden A ja B arvot elementeittäin ja säilöö lopputuloksen taulukkoon C. Huomattavaa on, että tässä tapauksessa taulukoiden täytyy olla vähintään yhtä suuria, kuin X-ulottuvuus indeksiavaruudessa, jossa tämä kerneli suoritetaan. Toinen huomion arvoinen seikka on, että yksikään datan elementti ei tässä tapauksessa saa riippua muista elementeistä.

```
__kernel void sum(
    const __global float* A,
    const __global float* B,
    __global float* C
) {
    const int index = get_global_id(0);
    C[index] = A[index] + B[index];
}
```

Kuva 10. Yksinkertaisen kernelin lähdekoodi. Otetaan parametreinä kolme taulukkoa ja summataan elementeittäin arvot kahdesta ensimmäisestä kolmanteen.

2.3.9 Ohjelmaobjektit

OpenCL:n ohjelmaobjektit sitovat kontekstin ja suoritettavan lähdekoodin yhteen. Kuvassa 11 nähdään yksi konstruktoreista. Huomaa, että parametri sources on vektori lähdekoodeja merkkijonoina. Ohjelmaobjektit täytyy vielä rakentaa luokan build-metodilla ennen kuin niitä voi suorittaa. Tämä toiminto muuttaa lähdekoodin suoritettaviksi käskyiksi laitteille.

```
cl::Program::Program ( const Context &  context,
                       const Sources &  sources,
                       cl_int *         err = NULL
                     )
```

Create a program from a vector of source strings and a provided context. Does not compile or link the program.

Kuva 11. Yksi ohjelmaobjektin konstruktoreista. Parametreinä tämä ottaa kontekstin, vektorin lähdekoodeja merkkijonoina ja virhekoodin muistiosoitteen.

2.3.10 Kerneliobjektit

Kerneliobjektit vastaavat lähdekoodin yksittäisiä kernelifunktioita. Ne rakennetaan ohjelmaobjektista ja halutun kernelifunktion nimestä kuvan 12 mukaisesti. Ennen käyttöä kerneliobjektiin täytyy sitoa kyseisen funktion käyttämät argumentit kuvan 13 mukaisilla setArg-metodeilla. Tehdyssä ohjelmassa käytettiin eniten kuvan viimeistä metodia. Ensimmäinen argumentti, index, on

0-alkuinen indeksi, jolla kerrotaan mitä funktion argumenttia ollaan sitomassa. Yksittäisten argumenttien kohdalla seuraavaksi annetaan argumentin koko ja osoitin argumenttiin. Puskuriobjektien kohdalla annetaan muistiobjektin koko, sizeof(cl_mem), ja puskuriobjektin hallinnoiman muistiobjektin osoite.

Kun kerneliobjektiin on sidottu kaikki sen käyttämät argumentit, se voidaan viimeinkin suorittaa komentojonon kautta kuvan 14 metodilla. Tämä metodi ottaa argumentteinaan myös kappaleessa 2.3.5 esitellyn indeksiavaruuden mukaiset objektit (kuva 15) määrittelemään, kuinka monessa pisteessä kerneli suoritetaan.

Kernelien suoritusta voidaan synkronoida usealla tavalla. Yksi tapa on kuvan 14 kahdella viimeisellä argumentilla: Toiseksi viimeinen argumentti on lista tapahtumista, joiden täytyy tapahtua ennen, kuin tämän kernelin suoritus aloitetaan. Viimeisellä argumentilla voidaan tunnistaa tämä nimenomainen suorituskäsky. Toinen tapa synkronoida on asettaa kernelin lähdekoodiin mem_fence() -funktiokutsu. Tämä pakottaa kaikki luku- ja kirjoitusoperaatiot valmistumaan ennen, kuin kernelin suoritusta jatketaan.

```
Kernel (const Program &program, const char *name, cl_int *err=NULL)
```

Kuva 12. Kerneliobjektin konstruktori. Parametreinä tämä ottaa ohjelmaobjektin, kernelifunktion nimen ja osoittimen virhekoodiin.

```
template<typename T , class D >
cl_int setArg (cl_uint index, const cl::pointer< T, D > &argPtr)
    setArg overload taking a shared_ptr type

template<typename T , class Alloc >
cl_int setArg (cl_uint index, const cl::vector< T, Alloc > &argPtr)
    setArg overload taking a vector type.

template<typename T >
std::enable_if< std::is_pointer< T >::value, cl_int >::type setArg (cl_uint index, const T argPtr)
    setArg overload taking a pointer type

template<typename T >
std::enable_if<!std::is_pointer< T >::value, cl_int >::type setArg (cl_uint index, const T &value)
    setArg overload taking a POD type

cl_int setArg (cl_uint index, size_type size, const void *argPtr)
```

Kuva 13. Kerneliobjektin argumenttien sidontametodit. Nämä ottavat parametreinään sidottavan argumentin indeksin ja joko pelkän argumentin tai argumentin koon ja osoittimen.

```
cl_int enqueueNDRangeKernel (const Kernel &kernel, const NDRange &offset, const NDRange &global, const NDRange &local=NullRange,
       const vector< Event > *events=NULL, Event *event=NULL) const
```

Kuva 14. Kernelin suorituskäsky komentojonon metodina. Tämä ottaa parametreinään kerneliobjektin, indeksiavaruuden poikkeaman, jos ei haluta aloittaa nollasta, globaalin indeksiavaruuden koon, lokaalin indeksiavaruuden koon, vektorin tapahtumista, joiden täytyy tapahtua ennen kuin annettua kerneliä aletaan suorittamaan ja osoittimen tapahtumaan, jolla voidaan tunnistaa tämä nimenomainen kernelin kutsumakerta.

```
NDRange ()
Default constructor - resulting range has zero dimensions.

NDRange (size_type size0)
Constructs one-dimensional range.

NDRange (size_type size0, size_type size1)
Constructs two-dimensional range.

NDRange (size_type size0, size_type size1, size_type size2)
Constructs three-dimensional range.
```

Kuva 15. Indeksiavaruuden objektien konstruktorit.

2.3.11 Tietojen hakeminen

Lähes jokaisesta OpenCL-luokan objektista voidaan hakea tietoa kahdella getInfo-metodilla. Näistä nähdään esimerkit kuvassa 16. Ensimmäinen metodi ottaa parametreinään OpenCL-makron ja infon muistiosoitteen ja palauttaa mahdollisen virhekoodin. Jälkimmäinen metodi ottaa OpenCL-makron templaattiargumenttina, virhekoodin osoitteen tavallisena parametrinä, ja palauttaa infon. Huomattavaa on, että infon tyyppi riippuu molemmissa metodeissa OpenCL-makrosta, eli siitä, mitä tietoa ollaan hakemassa. Näitä infon hakuun tarkoitettuja OpenCL-makroja löytyy useita kymmeniä ja ne ovat muotoa: CL_[luokan nimi]_[haettava tieto], esimerkkinä CL_DEVICE_GLOBAL_MEM_SIZE palauttaa kyseisen laitteen globaalin muistin määrän.

```
template<typename T >
                                                  cl_int  getInfo (cl_context_info name, T *param) const
                                                          Wrapper for clGetContextInfo().

template<cl_int name>
detail::param_traits< detail::cl_context_info, name >::param_type  getInfo (cl_int *err=NULL) const
                                                          Wrapper for clGetContextInfo() that returns by value.
```

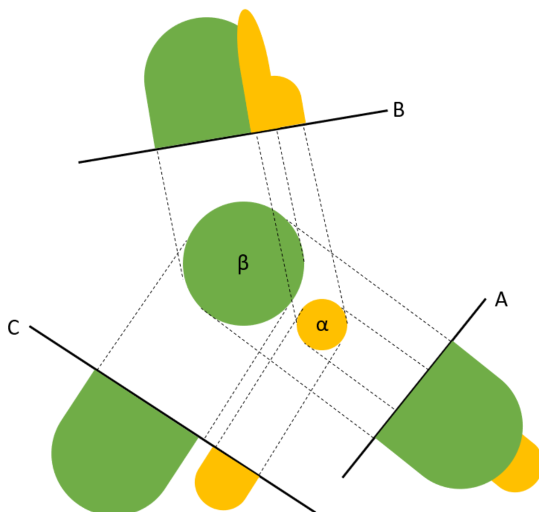
Kuva 16. getInfo-metodit.

3 Tomografia

3.1 Tomografia yleisellä tasolla

Tomografian tavoitteena on tuottaa useita poikkileikkauskuvia kuvattavasta kohteesta. Yleisimpiä ja parhaiten tunnettuja esimerkkejä lääketieteellisestä tomografiasta ovat tietokonekerroskuvaus ja magneettikuvaus. Tietokonekerroskuvauksessa kuvataan samaa kohdetta useasta eri suunnasta, useimmiten röntgenkuvauksella. Yksinkertaistaen magneettikuvauksessa ihmiskehon vetyatomit saadaan asettumaan voimakkaan magneettikentän suuntaisesti. Kun näihin atomeihin kohdistetaan radioaaltoja, ne asettuvat toiseen asentoon. Radioaaltolähetyksen päättyessä atomit palautuvat magneettikentän mukaisiksi ja vapauttavat energiaa, jonka perusteella voidaan muodostaa kuvia kohteen sisällöstä [28], [29]. Molemmista kuvauksista voidaan muodostaa 2- tai 3-ulotteisia malleja kuvatusta kohteesta.

Tomografialla saadaan parempi käsitys kohteen muodosta ja koostumuksesta. Tästä nähdään esimerkki kuvassa 17, jossa projisoidaan kaksi kappaletta kolmelle eri suoralle. Tomografiassa käytettävä kuvaaja on suorasta nähden kappaleiden vastakkaisella puolella. Kuvaaja saa tietoonsa kappaleen tiheyden, joka on suoran taakse muodostuva muoto. Suoralle A projisoidessa huomataan, että kappale on keskustastaan tiheämpi. Suoralle B projisoidessa nähdään yksi kappale, jonka oikea reuna on huomattavasti vähemmän tiheä. Suoralle C projisoidessa nähdään, että kohde koostuu kahdesta eri kappaleesta. Tällä menetelmällä voidaan rekonstruktoida kuva, joka näyttää tietoja kuvatun kappaleen sisällöstä.

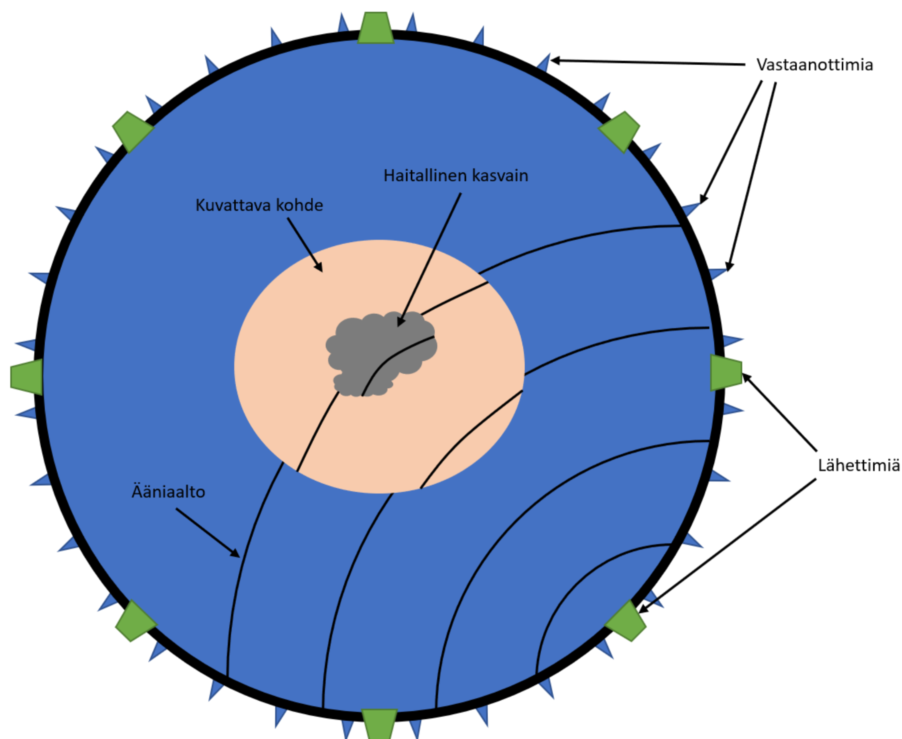


Kuva 17. Kahden kappaleen projisoinnit kolmelle eri suoralle.

3.2 Ultraäänitomografia

Ultraäänitomografia tuottaa kuvat nimensä mukaisesti ultraääniaaltojen avulla. Röntgenistä poiketen nämä aallot tarvitsevat väliaineen liikkuakseen sekä leviävät ympäriinsä lähetyspisteestä. Kohde upotetaan väliainetta sisältävään säiliöön, jonka reunoilla on ultraäänen lähettimiä ja vastaanottimia. Kuvassa 18 nähdään havainnollistava esimerkki ultraäänitomografian toiminnasta. Väliaineena käytetään useimmiten vettä, jossa äänen nopeus tiedetään ja sitä voidaan kontrolloida tarvittaessa helposti lämpötilan ja suolapitoisuuden avulla.

Jokainen lähetin lähettää omalla vuorollaan ultraääntä, jota vastaanottimet kuuntelevat. Kuvattavan kohteen koostumus vaikuttaa ääniaaltojen liikkeeseen. Kun tiedetään äänen lähetyksen alkamisen ajankohta ja milloin mikäkin vastaanotin kuuli äänen, voidaan kuluneista ajoista laskea äänennopeusjakauma kuvattavassa alueessa [30], [31], [32].



Kuva 18. Havainnekuva ultraäänitomografian toiminnasta.

4 Ultraäänen simulointi

Ultraäänilähettimet ja -vastaanottimet sijaitsevat ympyrän kehällä diskretisoidun nopeuskentän $C \in \mathbb{R}^{mxn}$ sisäpuolella, mutta eivät aivan alueen reunalla. Nopeuskentän arvoina ovat pääasiassa äänen nopeus väliaineessa sekä tutkittavat kohteet, joiden tiheyden vuoksi äänen nopeus niissä poikkeaa muusta kentästä.

Painekentät, muotoa $U \in \mathbb{R}^{mxn}$, sisältävät lähetetyn ääniaallon arvot jokaisessa alueen pisteessä. Painekenttiä ovat alueen nykyinen tilanne U^t, kentän tilanne edellisellä ajanhetkellä U^{t-1} ja kentän tilanne seuraavalla ajanhetkellä U^{t+1}.

Käytettäessä merkintöjä ajan muutoksesta päivityskertojen välillä Δt ja laskenta-alueen pisteiden välimatkasta Δx, saadaan uusi painekentän arvo ajan hetkellä $t + 1$ pisteessä (i, j) yhtälöstä (1) [33], [34]. Muuttujat (i, j) saavat seuraavia arvoja: $i = 1, \dots, m, j = 1, \dots, n$. Huomaa, että yhtälöön (1) on sijoitettu yhtälö (2) ensin mainitun selventämiseksi.

Painekentän laskennassa käytetään myös vaimennuskerrointa $\beta \in \mathbb{R}^{mxn}$. Tämä saa lähetinkehällä ja sen sisäpuolella arvon 0 ja lähetinkehän ulkopuolella alati voimakkaampia arvoja. Tämä vaimentaa mahdolliset heijastukset ultraääniaallon osuessa laskenta-alueen reunoihin.

$$U^{t+1} = \frac{\frac{F * U^t}{\Delta \mathrm{x}^2} + U^t(2Z+\beta) - ZU^{t-1}}{Z+\beta} \tag{1}$$

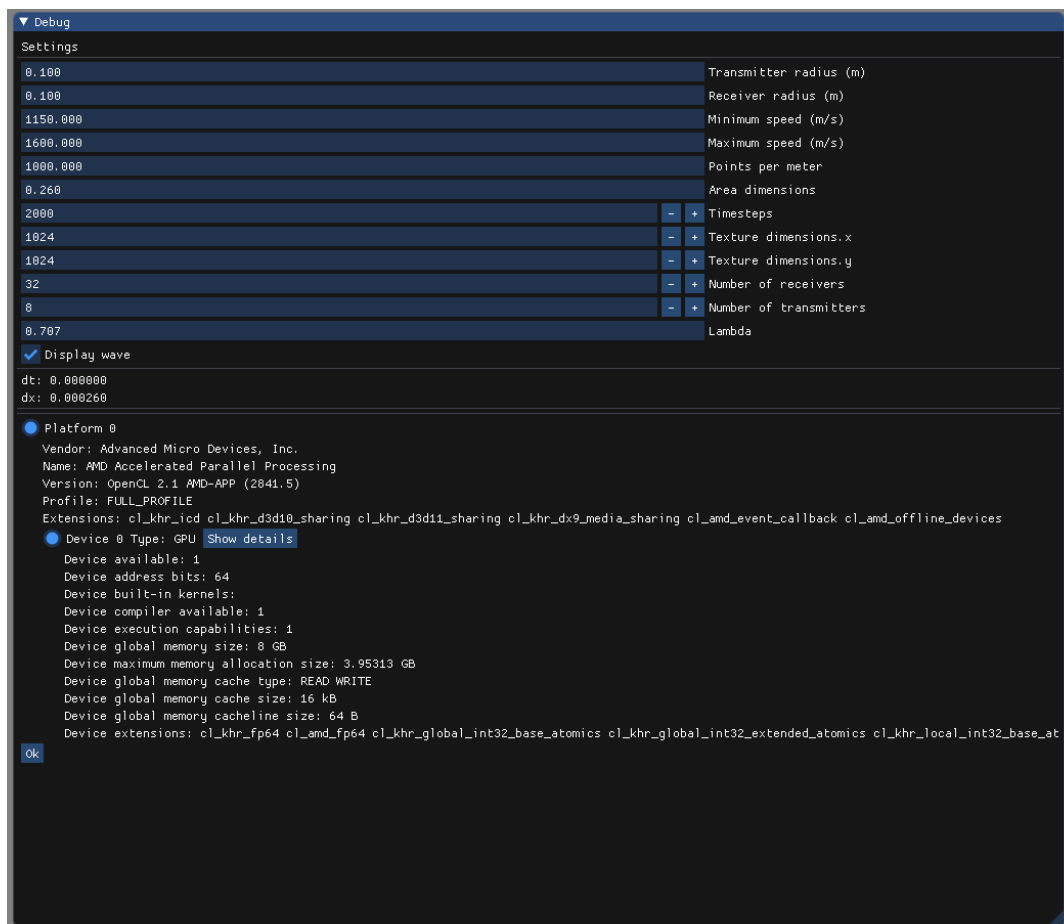
$$Z = \frac{1}{\Delta \mathrm{t}^2 C^2} \tag{2}$$

Yhtälössä (1) käytetty merkintä $F * U^t$ tarkoittaa konvoluutiota suodatusmatriisi F:n ja painekenttä U^t:n välillä. Konvoluutiota voidaan kuvata pinta-alana, joka muodostuu, kun yksi funktio vieritetään toisen funktion ylitse [35], [36]. Tämä vieritys tapahtuu asettamalla toinen funktio ”takaperin” ja aikaistamalla sitä niin, ettei yksikään funktioiden piste tapahdu samaan aikaan toisen funktion pisteen kanssa. Takaperin olevaa signaalia siirretään sitten yksi aika-askel kerrallaan toisen funktion ”ylitse”. Jokaisella siirrolla summataan kohdakkain olevien funktioiden arvojen tulot [37, s. 83—91], [38, s. 179—181].

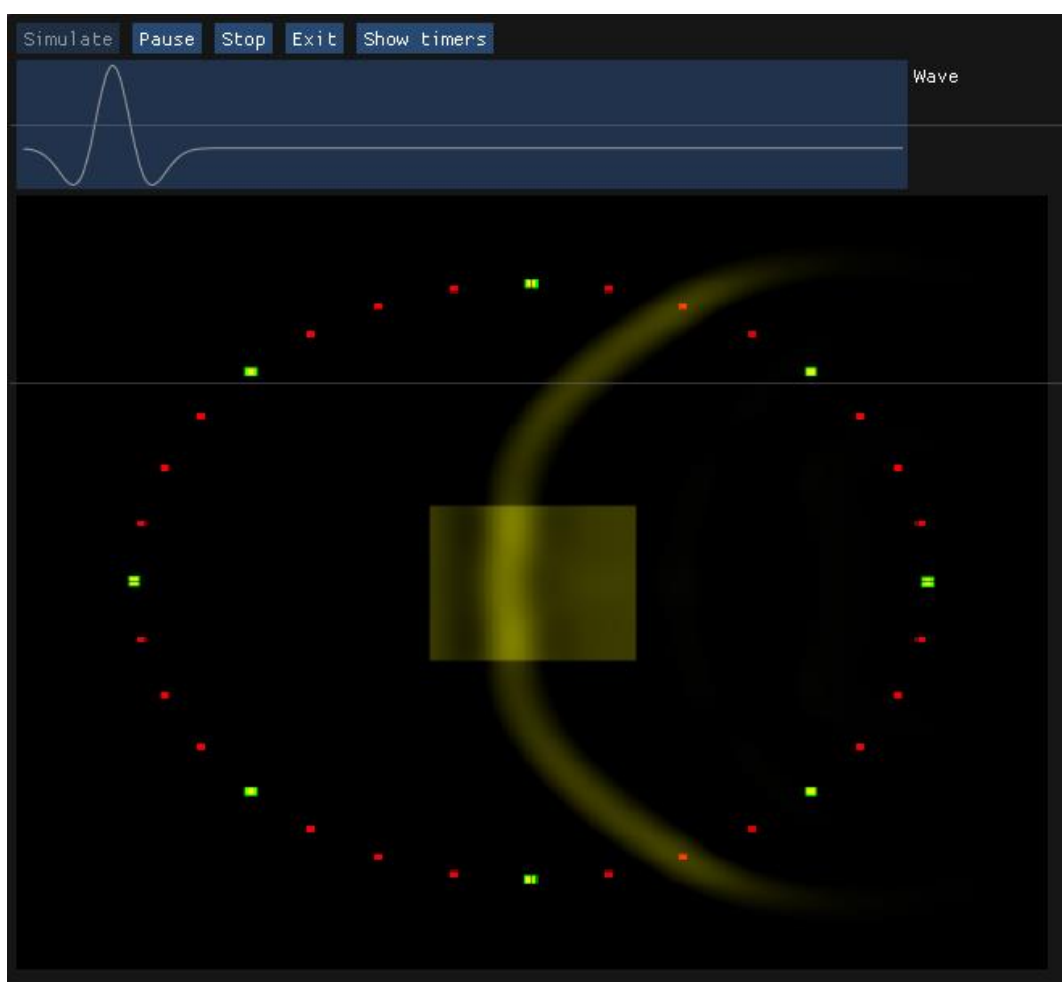
5 Tulokset

5.1 Ohjelma

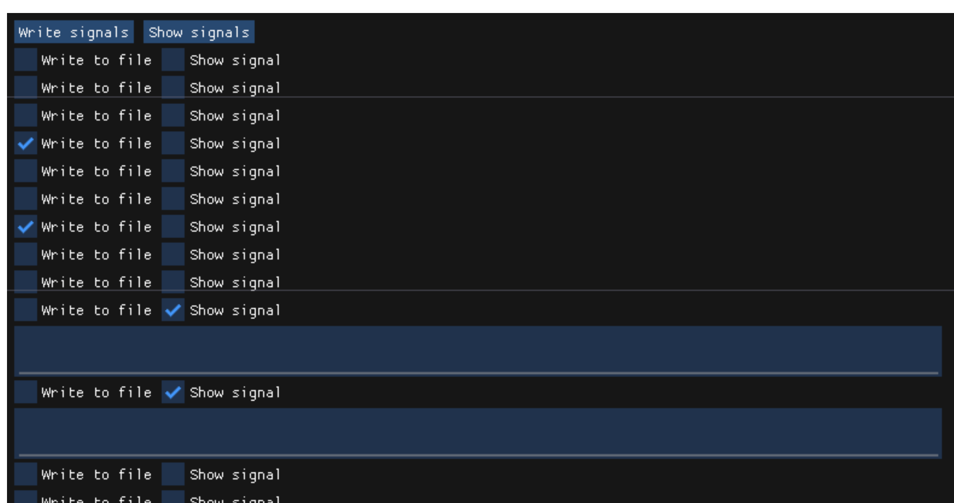
Ohjelma saatiin onnistuneesti valmistettua ja yhtälö (1) toteutettua OpenCL-kernelissä. Kuvassa 19 nähdään ohjelman aloitusnäkymä. Tässä vaiheessa voidaan vaihtaa simulaation aloitusarvoja ja laskentaan käytettävää laitetta. Ilman OpenCL-tukea käännetyssä ohjelmassa voidaan laitteiden sijasta valita laskentaan ja piirtoon käytettävien säikeiden määrät. Kuvassa 20 nähdään ohjelma toiminnassa. Ohjelmalla voidaan simuloida erimuotoisia ja -tiheyksisiä kappaleita. Kuvassa vihreät kappaleet ovat lähettimiä ja punaiset vastaanottimia. Kuvassa 21 näkyy osa vastaanotinten hallintanäkymää. Vastaanotinten lukemat signaalit voidaan tallentaa tiedostoihin, jonka jälkeen niitä voidaan prosessoida käyttäjän haluamilla ohjelmilla.



Kuva 19. Ohjelman aloitusnäkymä. Ohjelman parametrejä voidaan vaihtaa tässä näkymässä. Lisäksi tässä OpenCL-versiossa voidaan valita käytettävä alusta ja laite. CPU-versiossa voidaan valita säikeiden käyttömäärät.



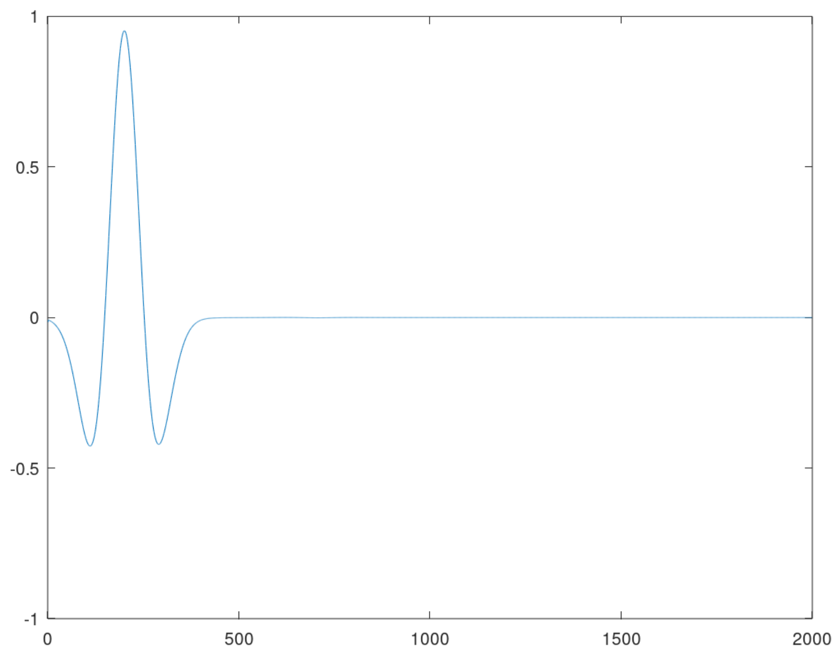
Kuva 20. Ultraääniaallon simulaatio.



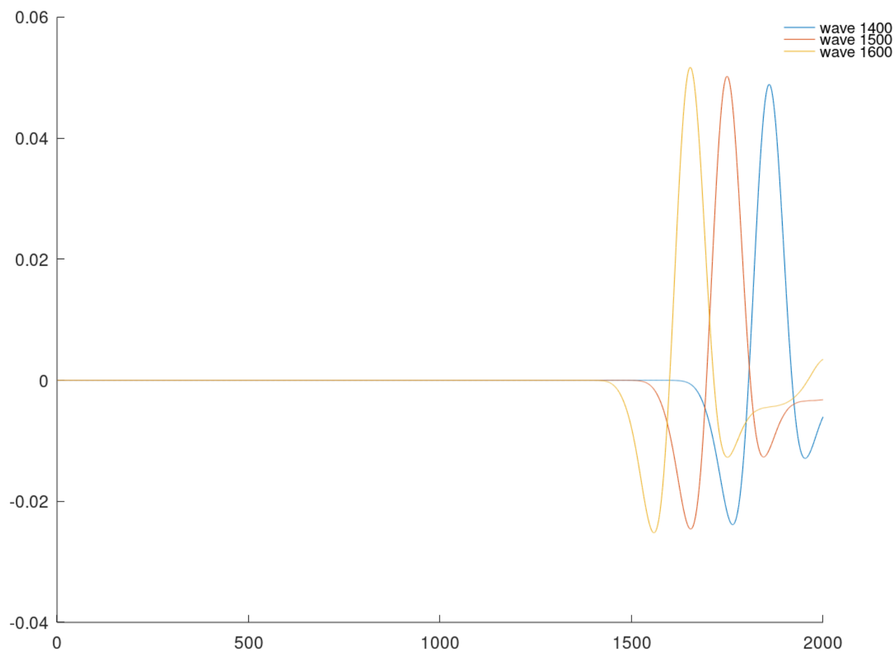
Kuva 21. Osa vastaanotinten hallintanäkymää.

5.2 Simulointi

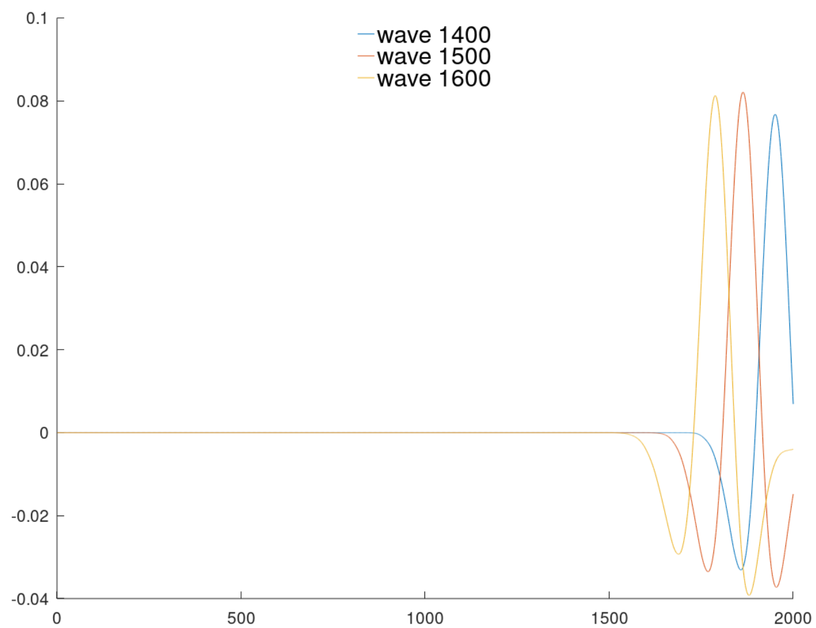
Ohjelman lähdeterminä käytettiin Ricker-aaltomuotoa [39]. Signaali on diskretisoitu 2000:een pisteeseen, jotka ovat seuraavissa kuvissa X-akseleilla. Y-akseleilla nähdään signaalin arvo. Kuvassa 22 nähdään kaikissa esimerkeissä lähetetty ultraääniaalto. Kuvassa 23 puolestaan nähdään vastaanotettu signaali yhtenäisen painekentän vastakkaisella puolella käytettäessä väliaineen nopeuksina 1400, 1500 ja 1600 m/s. Näistä signaaleista nähdään, kuinka nopeammassa väliaineessa signaali saapuu perille aikaisemmin. Kuvassa 24 nähdään sama simulaatio, mutta tällä kertaa laskenta-alueella oli hidastava kappale, jonka tiheyden vuoksi äänennopeus tässä kappaleessa oli 1150 m/s. Kuvassa 25 nähdään sama signaali jokaisen vastaanottimen lukemana. Koska sekä kappale että painekenttä ovat symmetrisiä, signaali saapuu lähettimestä katsoen vasemman- ja oikeanpuoleisille vastaanottimille yhtä aikaa. Huomattavaa on myös se, että kappaleen läpi kulkenut osuus signaalista pysyy korkeampiarvoisena, kuin pelkän väliaineen läpi kulkeva osuus.



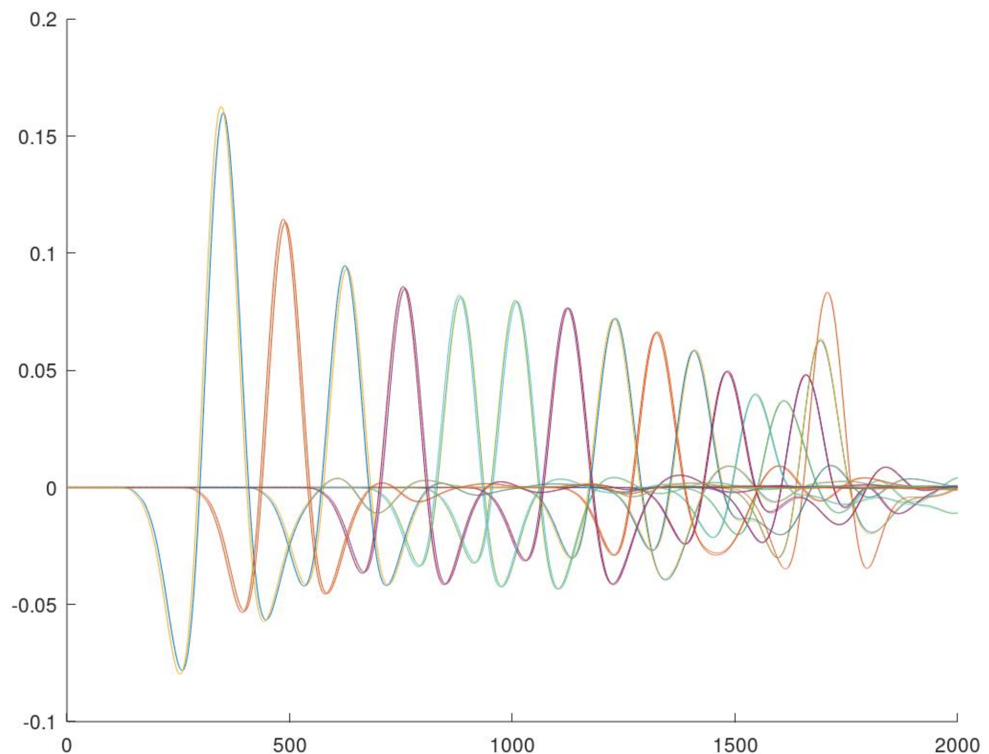
Kuva 22. Lähetetty ultraääniaalto.



Kuva 23. Vastaanotettu signaali yhtenäisen nopeuskentän vastakkaisella laidalla, kolmessa eri väliaineessa.



Kuva 24. Vastaanotettu signaali hidastavan kappaleen läpi nopeuskentän vastakkaisella laidalla, kolmessa eri väliaineessa.



Kuva 25. Kaikkien vastaanotinten lukemat hidastavan kappaleen läpi, kappale 1150 m/s ja tausta 1500 m/s.

5.3 Suorituskyky

Ohjelman suorituskyvystä mitattiin suorituksen kokonaisaika kahdeksan ultraääniaallon simuloinnista 32:lle vastaanottimelle. Jokaisella simuloinnin laskentahetkellä päivitettiin ultraääniaallon tilaa, laskettiin painekentän uusi tilanne ja tallennettiin vastaanotinten tila laskentalaitteelle. Jokaisen aallon jälkeen vastaanotinten data luettiin isäntäkoneen muistiin.

Suorituskykyä mitattiin kahdella kokoonpanolla, jotka ovat taulukossa 4. Kokoonpanolla A testattiin kaikkia taulukossa 5 löytyviä tapoja. Kokoonpanolla B testattiin ainoastaan säikeistettyjä toteutuksia, jotta nähtäisiin, kuinka suurta hyötyä kaksinkertainen säikeiden määrä tuottaa. Kaikki simuloinnit tehtiin kolmella eri tarkkuudella: 100, 500 ja 1000 simulointipistettä metrillä. Tuloskuvissa käytetyn laitteen perässä suluissa on joko simulointitapa tai käytetty näytönohjaimen ajuri.

Komponentti	Kokoonpano A	Kokoonpano B
Emolevy	Gigabyte GA-Z97X-Gaming G1	ASRock Fatal1ty Z87 Killer
Prosessori	Intel Core i5-4670K 3.4 GHz (turbo 3.8 GHz) 4 ydintä, 4 säiettä	Intel Xeon E3-1241 v3 3.5 GHz (turbo 3.7 GHz) 4 ydintä, 8 säiettä
Prosessorin jäähdytin	Alpenföhn Ben Nevis + 120mm tuuletin	Alpenföhn Ben Nevis + 120mm tuuletin
Näytönohjain	Vaihtuva	AMD R9 390X (ei laskentakäytössä tässä kokoonpanossa)
Muisti	4x Team Group 8GB DDR3 1333 MHz Elite	2x Fury HyperX 8GB DDR3 1600 MHz + 2x Gskill RipjawsX 8GB DDR3 1600 MHz
Levy	Seagate Enterprise SATA SSD 240 GB MLC	ADATA SU800 500GB SSD
Virtalähde	HEC Cougar 750 W	Lepa B1000-MB MaxBron 1000 W
Käyttöjärjestelmä	Windows 10, versio 1903	Windows 10, versio 1903

Taulukko 4. Testikokoonpanot.

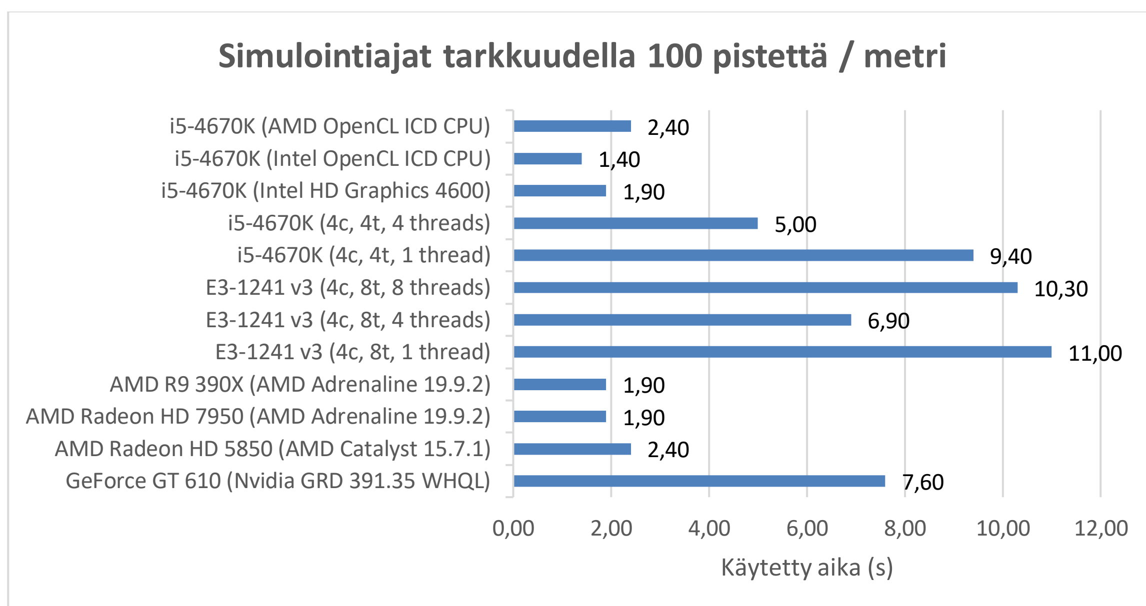
Laite	Simulointitapa
Näytönohjain	OpenCL: Isäntä: CPU Laite: GPU
Integroitu näytönohjain	OpenCL: Isäntä: CPU Laite: GPU
Prosessori (ICD)	OpenCL: Isäntä: CPU Laite: CPU
Prosessori (ytimet, säikeet, käytetyt säikeet)	Yksinkertainen säikeistetty toteutus (ei OpenCL)

Taulukko 5. Simulointitavat.

Kuvassa 26 nähdään käytetyt ajat pienimmällä tarkkuudella simuloitaessa. Yhtä vaille kaikki OpenCL:ää käyttävät laskennat mahtuvat sekunnin sisälle toisistaan. Ainoastaan GT 610 jää näistä jälkeen, mutta tämä näytönohjain onkin passiivijäähdytteinen, vain noin 30 Watin laite [40]. Muut käytetyt näytönohjaimet ovat aktiivijäähdytteisiä ja ne on luokiteltu 150 – 275 Watin kuormitukselle [8], [41], [42].

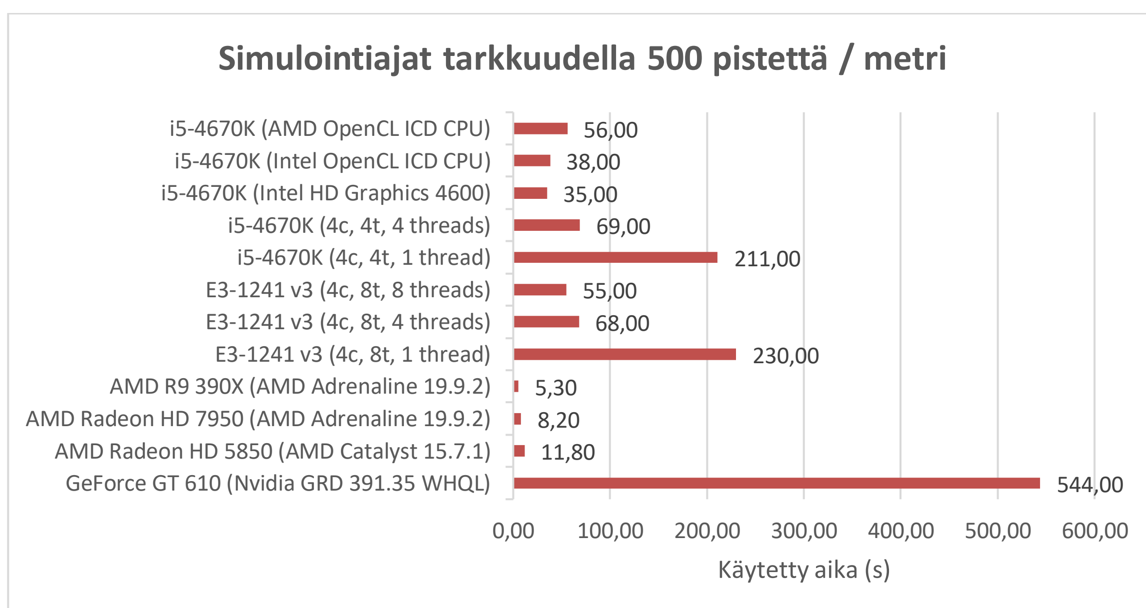
Nopeimmin laskennasta suoriutuu Intelin neliydinprosessori oman OpenCL-toteutuksensa siivittämänä. Tämä toteutus erittäin todennäköisesti onnistui kääntämään kernelit käyttämään tehokkaita SSE-käskyjä suorituksissaan [43].

Pelkällä prosessorin säikeistyksellä laskenta tapahtuu huomattavasti hitaammin ja Xeon-prosessorista voidaan nähdä, että dataa on liian vähän käsiteltäväksi, sillä suurempaa säiemäärää käyttäessä ohjelman suoritusaika hidastuu takaisin lähes sarjamuotoisen suorituksen tasalle. Tämä johtuu siitä, että säikeiden käyttäminen ei ole ilmainen operaatio, vaan niiden valmistelu aiheuttaa ylimääräistä työtä prosessorille ja käyttöjärjestelmälle.



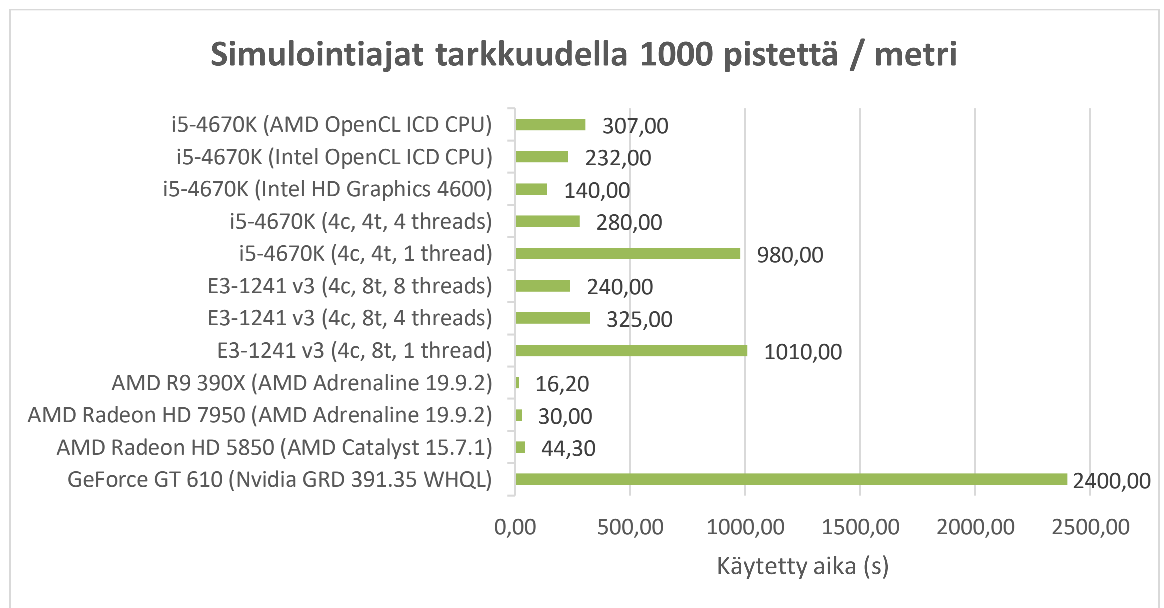
Kuva 26. Simulointiajat testauksen pienimmällä tarkkuudella.

Kuvassa 27 nähdään tulokset keskisuurella tarkkuudella simuloitaessa. Laskentatarkkuuden kasvaessa yhdellä ytimellä laskettaessa suoritusaika kasvaa molemmilla prosessoreilla useampaan minuuttiin, kun täysin säikeistettyinä molemmat ovat minuutin tuntumassa. Intelin OpenCL-toteutus sekä integroidulla näytönohjaimella että prosessorilla suoriutuu laskennasta lähes puolet nopeammin. Valikoiman vähävirtaisin laite jää auttamatta kaikista muista laitteista jälkeen, kun tehokkaammat näytönohjaimet selviävät laskennasta muutamissa sekunneissa.

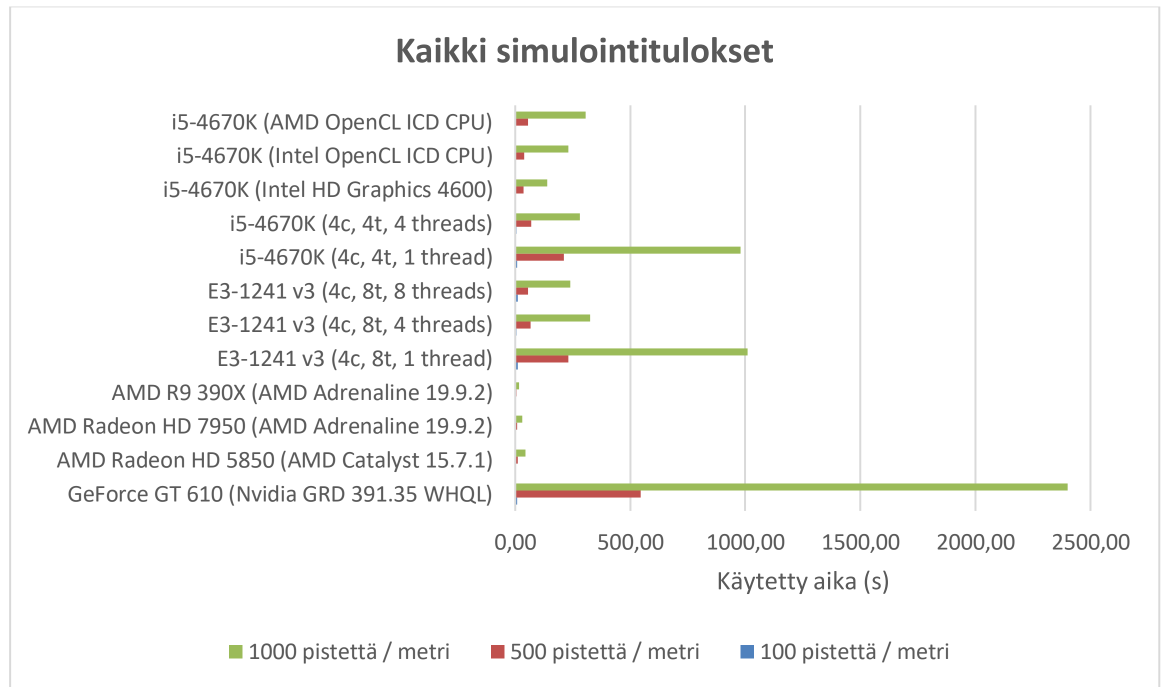


Kuva 27. Simulointiajat keskisuurella tarkkuudella.

Kuvassa 28 nähdään tulokset suurimmalla mittauksissa käytetyllä laskentatarkkuudella. Kun simuloinnin tarkkuutta kasvatetaan entisestään, tehokkaille näytönohjaimille ei ole vastustajia suorituskyvyssä. Neljänneksi nopeimpana tehtävästä suoriutuu Intelin integroitu näytönohjain, joka suoriutuu tehtävästä yli kaksi minuuttia nopeimman näytönohjaimen jälkeen. Tämä integroitu näytönohjain alkaa myös viimeinkin erottua CPU-version suorituskyvystä. AMD:n versio Intelin prosessorin käytöstä OpenCL:n laskentalaitteena laahaa edelleen säikeistettyjen toteutusten rinnalla. Yhdellä säikeellä laskenta-aika venyy jo yli 15:sta minuutin ja vähävirtaisimmalla laitteella aika alkaa lähestyä jo tuntia. Kaikki simulointitulokset näkyvät koostettuna kuvassa 29.



Kuva 28. Simulointiajat testauksen suurimmalla tarkkuudella.



Kuva 29. Kaikki simulointiajat koostettuna. 100:n pisteen tarkkuuden simuloinnit ovat lähes näkymättömiä tässä skaalassa. Tehokkailla näytönohjaimilla laskennasta suoriudutaan huomattavasti muita laitteita nopeammin.

6 Yhteenveto

Kuten kappaleesta 5.3 käy ilmi, näytönohjaimen käyttäminen nopeuttaa huomattavasti tietynlaisen laskennan valmistumista, kunhan käytössä on edes kohtuullisen tehokas laite. Valitettavasti tehokkaampia Nvidian näytönohjaimia ei saatu testaukseen, mutta OpenCL:n suorituskyky pitäisi olla samaa luokkaa molempien valmistajien toteutuksilla [44].

Ennen siirtymistä GPGPU-laskentaan täytyy kuitenkin tutkia, onko tämä omalla kohdalla edes mahdollista ja saadaanko siitä merkittävää hyötyä. Täytyy pitää mielessä, että näytönohjainlaskenta hyötyy eniten sellaisista operaatioista, jotka voidaan rinnakkaistaa helposti. Jo suhteellisen pienillä datamäärillä voidaan saada laskenta tapahtumaan moninkertaisella nopeudella, mutta täytyy harkita, onko tämä ohjelman suorituksen kannalta kriittinen paikka. Ylimääräisten APIen käyttäminen voi lisätä myös lähdekoodin kompleksisuutta ja tehdä siitä vaikeammin luettavaa ja järkeiltävää. Myös datan siirtäminen prosessorilta toiselle laskentalaitteelle ja takaisin vie ylimääräistä aikaa.

Sopivissa tilanteissa sovellettaessa GPGPU-laskennasta voidaan kuitenkin saada valtavia parannuksia ohjelmien suorituskykyyn. Sekä AMD että Nvidia tekevätkin nykyään varsinaisten näytönohjainten lisäksi nimenomaan laskentaan tarkoitettuja laitteita palvelinympäristöihin [45], [46].

Lähteet

1 AMD Ryzen Processors. Haettu 13.11.2019, osoitteesta: https://www.amd.com/en/ryzen#Products

2 Intel Core i7 Desktop Processors Comparison Chart. Haettu 13.11.2019, osoitteesta: https://www.intel.com/content/dam/support/us/en/documents/processors/core/intel-core-i7-comparison-chart.pdf

3 Intel Core i7 Mobile Processor Comparison Chart. Haettu 13.11.2019, osoitteesta: https://www.intel.com/content/dam/support/us/en/documents/processors/Intel-Core-i7-Mobile-Compare-Chart.pdf

4 Model Specifications. Haettu 13.11.2019, osoitteesta: https://www.amd.com/en/products/ryzen-threadripper#paragraph-136431

5 Intel Xeon Phi 72x5 Processor Family. Haettu 13.11.2019, osoitteesta: https://ark.intel.com/content/www/us/en/ark/products/series/132784/intel-xeon-phi-72x5-processor-family.html

6 Caulfield B. (2009, 16. joulukuuta). What’s the Difference Between a CPU and a GPU? Haettu 13.11.2019, osoitteesta: https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/

7 (2018, 30. marraskuuta) Difference between CPU and GPU. Haettu 13.11.2019, osoitteesta: https://techdifferences.com/difference-between-cpu-and-gpu.html

8 GIGABYTE R9 390X G1 Gaming. Haettu 13.11.2019, osoitteesta: https://www.techpowerup.com/gpu-specs/gigabyte-r9-390x-g1-gaming.b3394

9 Katajisto J. Rinnakkaislaskenta MPI-ympäristössä. Metropolia Ammattikorkeakoulu. 2009. URN: http://urn.fi/URN:NBN:fi:amk-200912036906

10 Salian I. (2019, 7. marraskuuta). Clearing the Air: NASA Scientists Use NVIDIA RAPIDS to Accelerate Pollution Forecasts. Haettu 13.11.2019, osoitteesta: https://blogs.nvidia.com/blog/2019/11/07/nasa-rapids-air-quality-forecasts/

11 High Performance Computing Explained. Haettu 13.11.2019, osoitteesta: https://www.amd.com/en/technologies/hpc-explained

12 Nvidia High performance computing. CUDA. Haettu 6.11.2019, osoitteesta: https://developer.nvidia.com/cuda-zone

13 The open standard for parallel programming of heterogeneous systems. Haettu 6.11.2019, osoitteesta: https://www.khronos.org/opencl/

14 Rupp K. (2018, 15. helmikuuta). 42 Years of Microprocessor Trend Data. Haettu 31.10.2019, osoitteesta: https://www.karlrupp.net/2018/02/42-years-of-micropro-cessor-trend-data/

15 Rowe J. (2017, 16. maaliskuuta). The Continuing Importance of GPUs For More Than Just Pretty Pictures. Haettu 31.10.2019, osoitteesta: https://www10.mcad-cafe.com/blogs/jeffrowe/2017/03/16/the-continuing-importance-of-gpus-for-more-than-just-pretty-pictures/

16 Dr Zaius. (2009, 3. elokuuta). Cage Match (CPU vs. GPU). Haettu 13.11.2019, osoit-teesta: https://gpgpu-computing.blogspot.com/2009/08/cage-match-cpu-vs-gpu.html

17 Cunningham C. (2018, 7. elokuuta). Data Transfer Rates Compared (RAM vs PCIe vs SATA vs USB vs More!). Haettu 31.10.2019, osoitteesta: http://blog.logicalincre-ments.com/2018/08/data-transfer-rates-bandwidth-cpu-ram-pcie-m-2-sata-usb-hdmi/

18 Main Memory: DDR4 & DDR5 SDRAM. Haettu 31.10.2019, osoitteesta: https://www.jedec.org/category/technology-focus-area/main-memory-ddr3-ddr4-sdram

19 [PCI-standardisointijärjestön vastauksia PCI-express kanavasta]. Haettu 31.10.2019, osoitteesta: https://pcisig.com/search/node/what%20is%20pci%20express

20 PHY Interface For the PCI Express, SATA, USB 3.1, DisplayPort, and Converged IO Ar-chitectures (Version 5.2). Haettu 31.10.2019, osoitteesta: https://www.in-tel.com/content/dam/www/public/us/en/documents/white-papers/phy-interface-pci-express-sata-usb30-architectures-3-1.pdf

21 Du P, Weber R, Luszczek P, Tomov S, Peterson G, Dongarra J. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. Parallel Computing 2012;38(8):391-407.

22 Compute Shader Overview. Haettu 13.11.2019, osoitteesta: https://docs.microsoft.com/en-us/windows/win32/direct3d11/direct3d-11-advanced-stages-compute-shader

23 Sans F, Carmona R. A Comparison between GPU-based Volume Ray Casting Implementations: Fragment Shader, Compute Shader, OpenCL and CUDA. CLEI ELECTRONIC JOURNAL 2017;20(2).

24 Memeti S, Li L, Pllana S, Kolodziej J, Kessler C. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption. Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing: ACM; 2017. DOI: 10.1145/3110355.3110356

25 OpenCL C++ Bindings. Haettu 11.11.2019, osoitteesta: https://github.khronos.org/OpenCL-CLHPP/index.html

26 The OpenCL Specification. Version 1.2. Rev 19. Haettu 13.11.2019, osoitteesta: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

27 Dr Zaius (2009, 2. lokakuuta). Matrix Multiplication 3 (OpenCL). Haettu 13.11.2019, osoitteesta: https://gpgpu-computing4.blogspot.com/2009/10/matrix-multiplication-3-opencl.html

28 Mori S, Barker P. B. (26. joulukuuta 2002). Diffusion magnetic resonance imaging: Its principle and applications. Conventional MRI. Wiley Online Library. DOI: https://doi.org/10.1002/(SICI)1097-0185(19990615)257:3<102::AID-AR7>3.0.CO;2-6

29 Doctor Klioze. (5.11.2013). MRI: Basic Physics & a Brief History. Haettu 6.11.2019, osoitteesta: https://www.youtube.com/watch?v=djAxjtN_7

30 Gordon R, Bender R, Herman G. T. Algebraic reconstruction techniques (ART) for three-dimensional electron microscopy and x-ray photography. Journal of theoretical biology 1970;29(3):471-481.

31 Hormati A, Jovanovic I, Roy O, Vetterli M. Robust ultrasound travel-time tomography using the bent ray model. Medical Imaging 2010: Ultrasonic Imaging, Tomography, and Therapy, 76290I.

32 Andersen A. H, Kak A. C. Simultaneous Algebraic Reconstruction Technique (SART): A superior implementation of the ART algorithm. Ultrasonic Imaging 1984;6(1):81-94.

33 Koponen J, Huttunen T, Tarvainen T, Kaipio J. P. Bayesian Approximation Error Approach in Full-Wave Ultrasound Tomography. IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control 2014;61(10):1627-1637.

34 Roy O, Jovanovic I, Hormati A, Parhizkar R, Vetterli M. Sound Speed Estimation Using Wave-based Ultrasound Tomography: Theory and GPU Implementation. Medical Imaging 2010: Ultrasonic Imaging, Tomography, and Therapy, 76290J.

35 Convolution. Haettu 4.9.2019, osoitteesta: http://mathworld.wolfram.com/Convolution.html

36 The convolution integral. Haettu 4.9.2019, osoitteesta: https://www.khanacademy.org/math/differential-equations/laplace-transform#convolution-integral

37 Burrus C. S, Parks T. W. (1985). DFT/FFT and Convolution Algorithms. Theory and Implementation. Wiley-Interscience.

38 Henttonen J, Peltomäki J, Uusitalo S. Tekniikan Matematiikka 2. Edita Prima. 2006. 2. painos.

39 Ryan H. Ricker, Ormsby, Klauder, Butterworth – A Choice of Wavelets. Can.Soc.Explor.Geophys.Record 1994;19(7):8-9.

40 GeForce GT 610 Specifications. Haettu 19.11.2019, osoitteesta: https://www.geforce.com/hardware/desktop-gpus/geforce-gt-610/specifications

41 Sapphire HD 5850 Toxic 2 GB. Haettu 19.11.2019, osoitteesta: https://www.techpowerup.com/gpu-specs/sapphire-hd-5850-toxic-2-gb.b200

42 XFX Double D HD 7950. Haettu 19.11.2019, osoitteesta: https://www.techpowerup.com/gpu-specs/xfx-double-d-hd-7950.b1735

43 What is SSE and AVX? Haettu 19.11.2019, osoitteesta: https://www.codingame.com/playgrounds/283/sse-avx-vectorization/what-is-sse-and-avx

44 Stone J. E, Gohara D, Shi G. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. Computing in science & engineering 2010;12(3):66-72.

45 Radeon Server Solutions. Haettu 19.11.2019, osoitteesta: https://www.amd.com/en/graphics/servers-solutions

46 ACCELERATING DATA CENTER WORKLOADS WITH GPUs. Haettu 19.11.2019, osoitteesta: https://www.nvidia.com/en-us/data-center/