TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Jaanus Varus 163577IAPM

GRAAFIKAPROTSESSORI PÕHINE ARVUTUSTE KIIRENDAMINE LÄBI .NET LINQ LIIDESE

Magistritöö

Juhendaja: lektor Ants Torim
Ph.D.

Tallinn 2017

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Jaanus Varus

05.05.2017

Annotatsioon

Käesoleva töö eesmärgiks on võimaldada läbi tuttava laialt levinud programmeerimisliidese kiirendada arvutustöid, kasutades selleks graafikaprotsessorit. Sealjuures pööratakse tähelepanu loomisprotsessile ning lähenemise plussidele ja miinustele.

Eesmärgi saavutamiseks tutvutakse esmalt kogukonna poolt tehtud töödega ning seejärel luuakse vajalikud tarkvaralised teegid. Teostatakse jõudluse võrdlus võrdlemaks kolme meetodit, kus rakendatakse arvutustöö läbiviimiseks protsessori ühte, mitut või graafikaprotsessori tuumasid. Oluline on, et liides programmeerija jaoks on iga meetodi puhul täpselt sama.

Töö tulemuseks on eksperimentaalne teek *.NET LINQ* liidese näol ning selle loomise kirjeldus. Teek on võimeline silmnähtavateks jõudlusparendusteks, kuid pakub ainult piiratud funktsionaalsust. Lisaks on tulemuseks teek, mis lubab *.NET* keskkonnas kasutada *Vulkan* graafika- ja arvutusteeki.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 50 leheküljel, 6 peatükki, 19 joonist, 6 tabelit.

Abstract

Graphics Processor Based Compute Acceleration Through .NET LINQ

The purpose of this thesis is to allow accelerating compute workloads through a familiar well known API surface using a graphics processor. Emphasis is on the creation process and the pros and cons of the method.

To achieve this goal, existing work on the subject is studied and based on that, necessary software libraries are developed. A performance measurement is made where the methods of using processor's single, multiple and graphics card's cores are compared. It is important that the interface for a programmer is the same for all these methods.

The outcome of this thesis is an experimental library that enables GPU acceleration through *.NET LINQ* API. It is capable of significant performance gains but only supports limited functionality. Additionally, a library is created that allows to use *Vulkan* graphics- and compute library in a *.NET* environment.

The thesis is in Estonian and contains 50 pages of text, 6 chapters, 19 figures, 6 tables.

Lühendite ja mõistete sõnastik

GPU	*Graphics Processing Unit*, graafikakaardil paiknev mikroprotsessor (graafikaprotsessor), mis on projekteeritud spetsiaalselt graafikainformatsiooni töötlemiseks ja kuvamiseks
GPGPU	*General-Purpose computing on GPU*, graafikaprotsessori kasutusjuht, kus kasutatakse protsessorit graafika kuvamise asemel rakenduse arvutuste läbiviimiseks
Shader	varjutaja ehk 3-mõõtmelises arvutigraafikas väike programm või algoritmikomplekt, mis määrab ära objektide 3-mõõtmeliste pinnaomaduste kujutamisviisi
Compute kernel	arvutustuum, mis on analoogselt varjutajale väike programm teostamaks arvutusi graafikaprotsessoril (tuntud kui ka *compute shader*)
SPIR-V	*Standard Portable Intermediate Representation V*, binaarne vahekeel (*intermediate language*) graafiliste varjutajate või arvutustuumade kirjeldamiseks
Vulkan	Programmeerimisliides graafika- ja arvutusriistvara jaoks
.NET	Üldotstarbeline arendusplatvorm tarkvararakenduste kirjutamiseks
.NET Standard	Hulk programmeerimisliideseid, mida kõik *.NET* platvormid peavad implementeerima (tagab porditavuse operatsioonisüsteemide vahel)
LINQ	*Language Integrated Query*, hulk standardseid päringuoperaatoreid, mis lubavad deklaratiivselt väljendada läbimis-, filtreerimis- ja projektsioonioperatsioone *.NET*-põhises programmeerimiskeeles
CLR	*Common Language Runtime*, *.NET* platvormi käitamisaja keskkond, mis jooksutab koodi ja pakub teenused arendusprotsessi lihtsustamiseks
Nuget	Paketihaldur teekide jagamiseks *.NET* platvormil
CUDA	*Compute Unified Device Architecture*, *NVIDIA* poolt loodud paralleelarvutusplatvorm ja programmeerimise mudel
FLOPS	*FLoating point OPerations per Second*, protsessori jõudluse mõõtühik: ujukomatehet sekundis
MIT litsents	*Massachusetts Institute of Technology* litsents on väga lubav ehk seab minimaalsed piirangud tarkvara taaskasutamisele

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

1.1 Taust

Mitmed valdkonnad nagu näiteks pilditöötlus, videotöötlus, füüsika, andmebaasid, masinõpe tegelevad tihti suurte andmehulkadega, mille töötlemine nõuab palju ressursse nii ajalises kui ka mahulises (mälu) mõttes. On oluline, et taolised operatsioonid kasutaksid efektiivselt kõiki riistvara poolt pakutavaid ressursse.

Tänapäeva protsessorid ei arene enam kiiruse mõttes (taktsagedus) nii nagu see oli veel eelmise sajandi lõpus. Selle asemel on jõudluskasv muutunud pigem horisontaalseks, kus arengusuund on protsessori tuumade arvu kasvul (vähendatakse transistoride suurusi). See aga tähendab vajadust kirjutada keerukamat tarkvara koodi, mis suudaks tuumasid tõhusalt ja korrektselt käidelda.

Selleks, et efektiivsemalt teostada operatsioone suurtel andmehulkadel, kasutatakse tehnikat, mis on tuntud kui paralleelandmetöötlus. Taolise töötluse puhul jaotatakse andmestik alamhulkadeks, kus igale hulgale rakendatakse sama arvutusalgoritmi. See tähendab, et on võimalik algoritmi koos iga alamhulgaga määrata erinevale lõimele ning neid lõimi töödelda samaaegselt mitmel protsessori tuumal.

Kui võrrelda *CPU* jõudlust *GPU* jõudlusega, on üheks võimalikuks võrdluse meetrikaks *FLOPS*, mis iseloomustab ujukomatehete arvu sekundis. Näiteks, hetkel tarbijaturul oleva protsessori *Intel Core i7-7700* puhul on selleks näitajaks 134,4 *GFLOPS* [1] ning graafikaprotsessori *GeForce GTX 1080* puhul 8872,96 *GFLOPS* [2]. Erinevus on ligi kaks suurusjärku ning tingitud kummagi riistvarakomponendi eesmärkide eripärast. Suurim erinevus peitub tuumade arvus, mis on eelmainitud protsessori puhul 4, kuid graafikaprotsessori puhul 2560. Paralleelandmetöötluse puhul on suurem tuumade arv igati sobilik.

1.2 Probleem

Paljude programmeerijate jaoks on *GPU* paralleeljõudluse kasutamise lävend liiga kõrge. Selleks, et adekvaatselt seda jõudlust ära kasutada, on vaja algoritmid ümber töötada paralleeltöötluse jaoks sobivaks ning taolist implementatsiooni pakkuvad programmeerimismudelid (näiteks *CUDA* [3], *OpenCL* [4], *Vulkan* [5], *Direct3D* [6], *Metal* [7]) on suhteliselt madalatasemelised ja riistvaralähedased. Seetõttu ongi tihti sellise lähenemise kulu väga kõrge ning kasu ei ole kulu väärt.

Eelnimetatud programmeerimismudelitele on loodud mitmeid raamistikke (näiteks *Alea Reactive Dataflow* [8], *PTask* [9]), mille eesmärk on pakkuda abstraktsioon madala taseme detailidele ning luua kasutajale lihtsustatum programmeerimisliides hallatud keskkonnas nagu näiteks *Java* [10], või *.NET* [11]. Probleemiks aga jääb endiselt keerukus ning vajadus aru saada graafikaprotsessori printsiipidest, sest taoliste raamistike eesmärk ei ole muuta *GPU* kasutust läbinähtavaks.

Minnes sammu võrra edasi kasutajasõbralikkuse poole, on välja pakutud ideid kasutada ära olemasolevaid tarkvaraarendajatele tuttavaid liideseid, et võimaldada suuresti läbinähtavalt graafikaprotsessori ressursse kasutada [12]. Üheks taoliseks liideseks on *LINQ (Language Integrated Query)* [13], mis pakub hulga standardseid päringuoperaatoreid, millega on võimalik deklaratiivselt väljendada läbimis-, filtreerimis- ja projektsioonioperatsioone *.NET*-põhises programmeerimiskeeles. Sellele liidesele ka käesolev töö keskendub, millele luuakse graafikaprotsessorit kasutatav realisatsioon. Tuleb arvestada, et „lihtsus“ ei tule tasuta ning selline lähenemine ohverdab jõudluses ning üldistusvõimes: *LINQ* toetab ainult piiratud arvul päringuoperaatoreid [8].

1.3 Eesmärk

Võttes aluseks *.NET* platvormil antud valdkonnas juba tehtud tööd [12] [14], realiseerida teek *LinqToCompute*, mis põhisuunitlusega peaks vastama küsimustele:

- Kuidas eelnevas alampeatükis väljapakutud meetodit implementeerida?
- Milliste probleemide lahendamiseks on see meetod sobiv?
- Kas alampeatükis 2.1 kirjeldatud jõudlusnäidikud on ka praktiliselt saavutatavad?

Lisaks on eesmärgiks kasutada avatud standardeid, mis ei piiraks realisatsiooni ühegi operatsioonisüsteemi või riistvaratarnijaga.

1.4 Metoodika

Rakendatakse nii kvantitatiivset meetodit jõudlusuuringu näol võrreldes tulemusi teoreetiliste alustega kui ka kvalitatiivset meetodit vaatluse ja kirjelduse näol, mis hõlmab autori omapoolset hinnangut käsitletud lähenemisele. Eesmärkideni jõutakse läbi kahe teegi kavandamise ja realisatsiooni.

Esimene teekidest on õhuke (pakub täieliku funktsionaalsuse ning ei lisa abstraktsioone) objektorienteeritud mähis *Vulkan'i C* keelse teegi ümber, mis võimaldab ligipääsu graafikaprotsessorile läbi *.NET* programmeerimiskeele.

Teine teek realiseerib *LINQ* päringu tõlkimise graafikaprotsessorile arusaadavaks vahekeeleks *SPIR-V* (vahekeelt käitab konkreetse riistvara jaoks loodud *Vulkan'i* draiver). Selleks väljastab *LINQ* päringu jaoks avaldiste puu (*expression tree*), mille läbimisel teostatakse tõlkimine. Lähteandmestiku kopeerimine graafikakaardile ligipääsetavasse mällu, päringu käitamine ning tulemusandmestikku tagasi kopeerimine toimub läbi esimese teegi.

Idee teekide eraldamises seisneb selles, et kui teine teek on pigem eksperimentaalse iseloomuga (uurimuslik), siis esimene on praktiliselt kasutatav ning panustab avatud lähtekoodi kogukonda.

Teekide arendamisel võetakse kvaliteedi tagamiseks aluseks *IBM FURPS* mudel [15]. Võetakse arvesse järgnevaid aspekte:

- **Funktsionaalsus (functionality)**. Teegid peavad olema funktsionaalselt terviklikud. *Vulkan*'i sidemete puhul tähendab see, et kõik *Vulkan*'i avaliku liidese käsud on ligipääsetavad *.NET* keskkonnas. See tagatakse süstemaatiliselt utiliidiga, mis kontrollib vastavalt liidese registri järgi, kas kõik käsud on teegis olemas. *LINQ* laienduse puhul võetakse registri asemel aluseks *.NET* raamistiku baasklasside teegi *System.Math* klass ja *System.Numerics* nimeruumi tüübid, mis peavad olema toetatud graafikaprotsessoril käitamiseks.

- **Kasutatavus (usability)**. Teegid peavad olema „turvaliselt“ kasutatavad - ilma võtmesõna *unsafe* kasutamata. See tähendab, et teegi liides ei avalikusta *pointer* tüüpi muutujaid. See on eriti oluline, kui peaks tekkima soov kasutada teeki keelest *Visual Basic*, mis on keelanud otse *pointeritega* töötamise [16]. Lisaks arvestatakse *C#* keele poolt pakutavat „süntaktilist suhkrut“ (vaikeparameetrid, võtmesõnad *params* ja *using*).

- **Usaldusväärsus (reliability)**. Pakutav funktsionaalsus peab toimima tõrgeteta. Garanteeritakse teatud piirini läbi ühiktestide täites täieliku lauseadekvaatsuse kriteeriumid. *Vulkan*'i sidemete puhul tähendab see, et kõik põhikäskudega seotud koodilõigud on kaetud ühe või rohkem testiga. Ei arvestata laiendustega sissetoodavaid käske, milleks on vajalik spetsialiseeritud riistvara olemasolu (puuduvad vahendid testimiseks). *LINQ* laienduse puhul kaetakse kasutusjuhud, kus on rakendatud funktsionaalselt toetatud tüüpide meetodeid.

- **Jõudlus (performance)**. Pidev suhtlus hallatud (*.NET)* ja haldamata (*native C*) keskkondade vahel ning nende mälumudelite erinevused tagavad vajaduse andmestruktuuride teisendamiseks keskkonna vahetusel. Taolised teisendused peaksid olema võimalikult tõhusad: mitte kopeerima üleliigselt mälu või koormama *CLR'i* prügikoristajat (*garbage collector*). Seda eriti programmi radadel, mille käitamispotentsiaal on mitmekümnetes kordades sekundis (*hot paths*). Selleks luuakse kõik lühikese elueaga andmestruktuurid, mille ainus eesmärk on andmete transport rakenduse ja *Vulkan*'i liidese vahel, kasutades võtmesõna *struct*. Küsimustes, kus mitme variandi vahel pole teada, kumb on jõudluse poole pealt tõhusam, teostatakse jõudlustest.

- **Toetatavus (supportability)**. Vaadeldakse hallatavust ja porditavust. Hallatavus tagatakse lähtekoodi avalikustamisega keskkonnas *GitHub*. Kolmandal osapoolel koodist arusaamist lihtsustatakse läbi koodi kommentaaride ning *Vulkan*'i sidemete puhul otsusega mitte rakendada koodi generaatori põhist lähenemist, mis nõuab arusaamist generaatorist. Porditavuse puhul sihitakse *.NET Standard* tüüpi teeki, mis on garanteeritud toimima *Windows, Linux, Android, iOS* ja *macOS* platvormidel [17].

Töö kirjutamisel üritatakse eelistada võimalusel eestikeelset sõnavara. Infotehnoloogiliste terminite tõlkimisel on aluseks võetud e-teatmik [18].

1.5 Ülevaade tööst

Teine peatükk annab ülevaate teema teoreetilistest alustest. Vaadeldakse juba tehtud töid, nende tulemusi ja puudujääke. Kirjeldatakse, kuidas on antud töö loogiline jätk nendele alustele.

Kolmas peatükk sukeldub tehnilistesse detailidesse. Antakse lühiülevaade kasutatud tehnoloogiatest koos valiku põhjendustega. Infot esitatakse võimaluse korral jooniste, tabelite ja loendite kaudu lugemise ja arusaamise lihtsustamiseks.

Neljandas peatükis kirjeldatakse realisatsioon ehk autoripoolne praktiline osa.

Viies peatükk teostab realisatsiooni põhjal vajalikud testid, mis annavad piisava andmestiku analüüsiks. Analüüsi põhjal vastatakse küsimustele, mis sai seatud sissejuhatava peatüki eesmärgis. Lisaks annab autor omapoolse hinnangu käesoleva töö realisatsioonile ning vaadeldakse võimalikke edasiarendusi.

2 Teoreetilised alused

Teema spetsiifilisuse tõttu materjale napib, küll aga tagab spetsiifilisus suurema praktilise väärtuse konkreetses segmendis. Lisaks tasub mainida, et riistvarast rääkides peetakse vaikimisi silmas tüüpilist lauaarvuti seadistust, kus saab selgelt eristada protsessorit ja diskreetset graafikaprotsessorit.

Järgnevad alampeatükid toovad välja järeldused ja puudujäägid olemasolevatest töödest. Esmalt vaadeldakse ühte magistritööd [12], mille teema kattub suures joones antud töö teemaga ning millele käesolev töö oleks loogiliseks jätkuks. Seejärel uuritakse paari väiksema kaaluga teemakohast materjali [8] [14] [19].

2.1 Magistritöö „*Parallelizing LINQ Program for GPGPU*“

Tegu on 2012 aastal kirjutatud magistritööga autori *Pritesh Agrawali* poolt ülikoolis *Indian Institute of Technology Kanpur* [12].

Töö eesmärk on väga sarnane antud töö eesmärgile. Rääkides valdkonnaspetsialistidest, puudub nendel tarkvaraarendajatel tihti teadmised graafikaprotsessori paralleliseeritud võimsust ära kasutada. Otsitakse võimalust pakkuda tuttav arendusliides, mis seda probleemi leevendab.

Realisatsioonis on valitud graafikakaardiga suhtlemiseks *CUDA*. Tegemist on väga levinud *NVIDIA* paralleelarvutus platvormi ja programmeerimisliidesega. Üheks suureks puudujäägiks on asjaolu, et riistvara, mis *CUDA*'t toetab, on piiratud üldjuhul *NVIDIA* toodanguga. Samas võimaldab see hõlpsamini kasutada ära riistvara eripära ning tagada jõudluse eeliseid konkurentide ees. [3]

Aluseks on võetud *Barnes-Hut* algoritm, mis on töös läbivaks teemaks. Selle algoritmi eesmärk on kolmemõõtmelises ruumis grupeerida *n* keha hoiustades neid kaheksandik puusse (*octree*). Realiseeritakse algoritmi lihtsustatud variant, kus dimensioon on taandatud kahemõõtmeliseks. Tuuakse välja *.NET* realisatsioon keeles *C#* ning *CUDA* realisatsioon keeles *C*. Võrreldakse mõlemat ning tuuakse välja eripärad/puudused.

Kahjuks ei käsitle töö automaatset teisendusprotsessi, et ühest realisatsioonist teine luua. See on üks põhiaspektidest, millega käesolev töö erineb - pakkudes vaadet, kuidas sellist teisendust implementeerida.

Tulemustes on teostatud jõudlusmõõtmised *Barnes-Hut* algoritmil, et võrrelda samade sisendite korral kulunud aega *LINQ* vs *CUDA* realisatsiooni korral. Järgnevalt on esitatud võrdluse tulemused.

	CPU	***GPU***			***CPU***	***GPU***	
Nr of Particles	***Computation (Sec)***	***Computation (Sec)***	***Memory copy (Sec)***	***Nr of Particles***	***Computation (Sec)***	***Computation (Sec)***	***Memory copy (Sec)***
100	0.08	0.014	4.037	60000	13.553	0.045	4.25
200	0.084	0.003	4.048	70000	15.965	0.052	4.282
300	0.1	0.009	4.044	80000	18.989	0.063	4.316
400	0.11	0.007	4.045	90000	21.293	0.063	4.353
500	0.12	0.009	4.044	100000	24.546	0.073	4.386
600	0.122	0.01	4.045	200000	55.647	0.14	4.727
700	0.124	0.011	4.045	300000	89.33	0.208	5.056
800	0.136	0.011	4.045	400000	123.236	0.299	5.416
900	0.152	0.011	4.047	500000	159.738	0.354	5.606
1000	0.164	0.011	4.047	600000	193.08	0.435	5.902
2000	0.276	0.01	4.049	700000	232.619	0.432	6.228
3000	0.384	0.0073	4.0537	800000	279.92	0.566	6.475
4000	0.536	0.009	4.056	900000	313.99	0.692	6.802
5000	0.692	0.01	4.058	1000000	347.1	0.574	7.146
6000	0.824	0.008	4.063	2000000	755.46	1.126	10.174
7000	1.008	0.011	4.066	3000000	T.O.	2.119	12.889
8000	1.144	0.01	4.07	4000000	T.O.	2.987	15.95
9000	1.324	0.013	4.072	5000000	T.O.	3.44	18.803
10000	1.488	0.016	4.075	6000000	T.O.	3.947	21.663
20000	3.436	0.021	4.111	7000000	T.O.	4.748	24.704
30000	5.68	0.028	4.148	8000000	T.O.	5.394	27.805
40000	8.137	0.032	4.181	9000000	T.O.	6.386	30.3758
50000	10.581	0.039	4.216	10000000	T.O.	7.018	33.374

Tabel 1. Barnes-Hut algoritmi jõudluse erinevused CPU ja GPU vahel

Tabelist tuleb selgelt välja *GPU* realisatsiooni eripära, kus arvestatav osa aega kulub mälu kopeerimisel graafikakaardile. Huvitavaks murdepunktiks on 30000 osakese piir, millest alates hakkab graafikaprotsessori kasutamine ennast jõudsalt ära tasuma. See annab kindlust meetodi kasulikkuse osas.

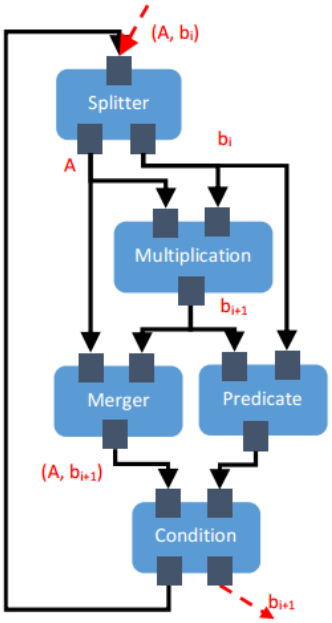
Loogilise jätkuna tegeleb käesolev töö teisendusprotsessi realiseerimisega ning hindamisega. Vaadeldakse, kui suur on lisatud ajakulu *LINQ* avaldiste tõlkimisest ning lisaks võetakse jõudluse mõõtmisel arvesse nii ühe- kui ka mitmelõimelise *CPU* põhised algoritmid. Viimaks annab autor ka omapoolse hinnangu realisatsiooni tarvilikkusest ja esinenud probleemidest.

2.2 Artikkel „*Alea Reactive Dataflow: GPU Parallelization Made Simple*“

Tegu on 2014 aastal kirjutatud artikliga, mille autoriteks on *Luc Bläser*, *Daniel Egloff*, *Oskar Knobel*, *Philipp Kramer*, *Xiang Zhang* ja *Daniel Fabian* [8].

Artikkel pakub lahendust samale probleemile nagu ka eelnev töögi. On mainitud, et kulu, mida nõuab arendajate õpetamine käsitlemaks madala-taseme graafikaprotsessori liideseid, vaadeldakse tihti kui liiga kõrgena.

Artikkel aga võtab veidi teise lähenemissuuna, luues programmeerimismudeli, millega on võimalik lahendada suuremat hulka probleeme, ohverdades sealjuures kasutuslihtsust. Suureks erinevuseks on paradigma käsitlus, kus antud artikkel ei vaatle arvutusprobleeme kui *pull-based* vaid *push-based* ehk reaktiivsete probleemidena. Loodud liidese kaudu kirjeldatakse deklaratiivselt arvutusprogrammi luues operatsioonide graaf, kus graafi iga sõlm kujutab arvutustehet ning sõlmi on võimalik omavahel erinevat pidi ühendada.




```
var source = new Splitter
                    <Matrix, Vector>();
var mult = new Multiplication
                    <Matrix, Vector>();
var cmp = new Predicate<Vector>
        ((a, b) => Abs(a - b) > 1E-6);
var next = new Merger<Matrix, Vector>();
var cond = new Condition<Vector>();

source.First.ConnectTo(mult.First);
source.Second.ConnectTo(mult.Second);
source.First.ConnectTo(next.First);
mult.Output.ConnectTo(next.Second);
mult.Output.ConnectTo(cmp.First);
source.Second.ConnectTo(cmp.Second);
next.Output.ConnectTo(cond.PassThrough);
cmp.Output.ConnectTo(cond.Criterion);
cond.True.ConnectTo(source.Input);

cond.False.OnReceive(Console.WriteLine);
source.Send(new Tuple(A, b0));
```

Joonis 1. Artikli "Alea Reactive Dataflow" pakutud lahenduse visualisatsioon

Kuigi eelnimetatud artikkel ei käsitle *.NET LINQ* liidest, mainib ta ära selle, kui ühe võimaliku variandi lihtsa paralleeltöötluse võimaldamiseks. Plussidena tuuakse välja lihtsus ja asjaolu, et *.NET* arendajad üldjuhul teavad juba seda liidest. Miinusena tuuakse välja, et selle lähenemise puhul on raske üldistusi luua, sest *LINQ* pakub ainult piiratud arvul päringufunktsioone, milleks on projektsioon, filtreerimine, sorteerimine ja grupeerimine.

2.3 Muud allikad

Microsoft on ka ise pakkunud välja paralleeltöötlus võimaluse läbi *LINQ* päringuoperaatorite, mis on tuntud kui *PLINQ* ehk *Parallel LINQ* nime all [19]. Kahjuks jääb see käsitlus *CPU* mitmelõimelisuse piiridesse ning ei käsitle *GPU*'d võimaliku ressursina.

Avatud lähtekoodina on *GitHub*'is saadav ka teemakohane projekt, mis on realiseeritud keeles *F#* ning oskab teisendada *LINQ* avaldist *OpenCL* arvutustuuma koodiks [14]. Kahjuks puudub projektil dokumentatsioon ja analüüs, mis teeb projektist arusaamise ja järelduste loomise teema keeruka iseloomu tõttu raskeks.

3 Tehnoloogiate ülevaade

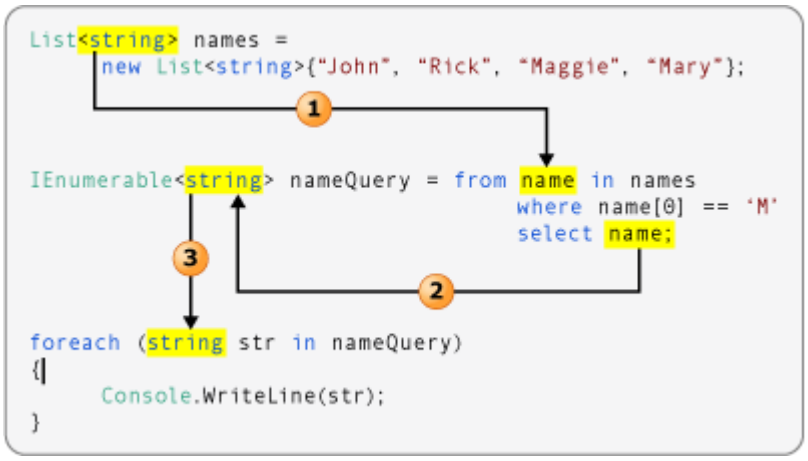
Enne kui laskutakse töös loodud teegi detailidesse, antakse ülevaade tööd võimaldavatest tehnoloogiatest.

3.1 LINQ

LINQ (*Language-Integrated Query*), mis tutvustati *.NET* raamistiku versioonis 3.5, on tehnoloogia sidumaks objektide maailma andmete maailmaga.

Käesolevas peatükis seletatakse lahti peamised *LINQ* päringuoperaatorid ning tuuakse välja, miks just need operaatorid sobiksid liideseks massarvutuste ja graafikaprotsessori vahele.

Traditsiooniliselt kirjutati päringuid andmete pihta sõneliste lausetega (näiteks *SQL* päringud), mida ei olnud kompilaatoril võimalik valideerida ning mille puhul ei suutnud *.NET* arendustööriistad automaattuvastust pakkuda. Probleemiks oli ka erinevate andmeallikate erinev dialekt (näiteks erinev süntaks iga andmebaasiserveri puhul), mis sundis igast andmeallikas andmete pärimiseks antud andmeallikale kohandatud päringuid kirjutama. Standardiseerimaks *.NET* platvormil päringute esitamist, loodi abstraktsioon ja vajalikud programmeerimiskeele täiendused (keeltele *C#*, *Visual Basic* ja *F#*) eelnevate probleemide lahendamiseks. See abstraktsioon ei ole sõltuv ühestki andmeallikast ning pakub arendusliidest mistahes andmeallika pakkuja realiseerimiseks. Teiselt on *LINQ* suuresti inspireeritud relatsioonilisest andmeallikatest, mistõttu meenutab *LINQ* süntaks *SQL* süntaksit ning on ideaalne esindama *SQL* päringuid struktureeritult objektorienteeritud keeles.



Joonis 2. LINQ avaldise näide filtreerimisoperaator puhul

Eelneval joonisel on välja toodud *LINQ* operaatorid *Where* ja *Select*. Mõlemad operaatorid on esitatud *query syntax* kujul, mis on üks *LINQ* eripäradest. Selle süntaksi eesmärk on lihtsustada lugemist. Kindlasti ei ole nõutud operaatorite väljendamine just sellisel viisil, vaid võib piirduda ka tavalise süntaksiga.

```
IEnumerable<string> nameQuery = names.Where(name => name[0] == 'M');
```

Päringuoperaatorid jagunevad suures plaanis kaheks selle põhjal, millal päring käitatakse. Operaatorid, mis tagastavad üksiku väärtuse (näiteks *Count*, *Sum*, *Average*) käitatakse koheselt. Operaatorite, mis tagastavad väärtuste jada, käitamisega viivitatakse, kuni esimese itereerimise või agregeerimiseni. Taoline viivitamine võimaldab päringuid järk-järgult üles ehitada ning päring teostada rakendusele sobival hetkel.

Pakutavad päringuoperaatorid on tabelikujul koos näidetega kirjeldatud *Microsoft*'i dokumentatsiooniportaalis [20]. Nendest lähemalt vaadeldakse kahte operaatorit *Select* ja *Zip*, mis on ideaalsed kandidaadid paralleeltöötluse jaoks, kuna andmestiku iga elemendiga opereeritakse sõltuvalt ülejäänud andmestikust.

```
public static IQueryable<TResult> Select<TSource, TResult>(
      this IQueryable<TSource> source,
      Expression<Func<TSource, TResult>> selector
)
```

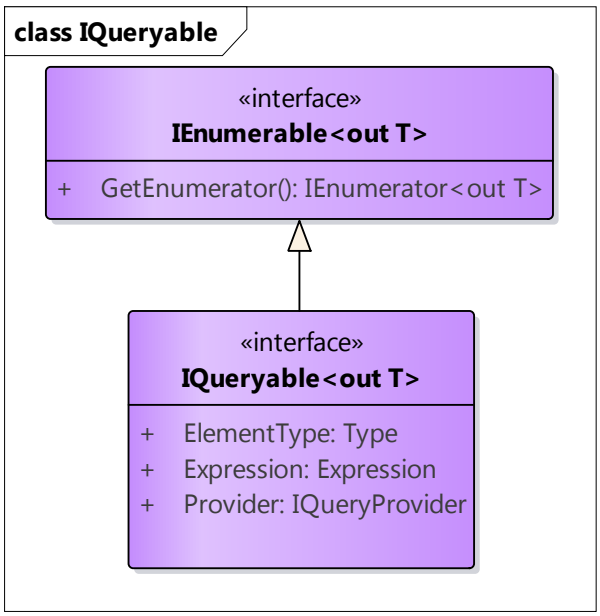
Select operaatori puhul on tegemist relatsioonialgebrast tuntud projektsiooniga, mis projekteerib iga elemendi jadas uude vormi. Meetodi signatuuri *source* muutuja on päritav andmeallikas ning *selector* muutuja kirjeldab projektsiooni funktsiooni, mis võtab

sisendiks geneerilise andmetüübi *TSource* ning tagastab geneerilise andmetüübi *TResult*. Sobib näiteks maatriksite inverteerimiseks maatriksjadal.

```
public static IQueryable<TResult> Zip<TFirst, TSecond, TResult>(
       this IQueryable<TFirst> source1,
       IEnumerable<TSecond> source2,
       Expression<Func<TFirst, TSecond, TResult>> resultSelector
)
```

Zip operaator võtab sisendiks kaks andmeallikat *source1* ja *source2* ning iga elemendi kohta esimesest allikast võtab talle vastava elemendi teisest allikast ning rakendab funktsiooni, mis võtab sisendiks geneerilised muutujad *TFirst*, *TSecond* ning tagastab geneerilise muutuja *TResult*. Sobib näiteks skalaarkorrutiste teostamiseks kahe vektorjada vahel.

LINQ päringuid teostatakse andmejadadel. Andmejada võib esitada läbi kahe liidese: *IEnumerable* või *IQueryable*, mis no mõeldud fundamentaalselt erinevate kasutusjuhtude jaoks. *IEnumerable* on laiemalt levinud liides, mis iseloomustab itereeritavat andmejada ning võimaldab keeletasemel luua iteraatormustrile loomulikke konstruktsioone nagu *foreach* või *yield return*. Ideaalne iseloomustamaks muutmälus paiknevaid andmejadasid. Küll aga ei sobi ta andmejadadele, mis asuvad väljaspool muutmälu – näiteks kõvakettal, võrgus või videomälus, sest ei paku oma liideses piisavalt infot. Selle probleemi lahendamiseks on loodud *IQueryable* liides, mis täiendab *IEnumerable* liidest vajalike andmetega.



Joonis 3. IQueryable liidese relatsioon IEnumerable liidesega

IQueryable liidese atribuut *Provider* iseloomustab implementatsiooni, mis oskab sisendjada ja talle rakendatud päringuoperaatorite põhjal käivitada päringud mõnes välises keskkonnas ning on täpselt see, mida käesolevas töös hakatakse täpsemalt käsitlema graafikaprotsessori keskkonna näol.

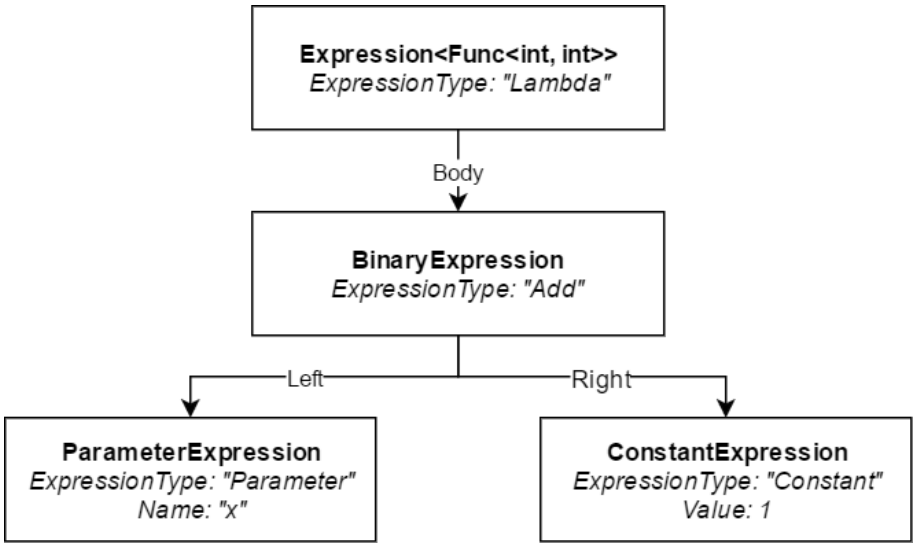
Expression on teine tähtis tüüp, millele keelekompilaator pakub erilist tuge. Talle saab omistada suvalise avaldise *lambda* funktsiooni näol teatud piiranguga. Nimelt ei ole lubatud kehandiga avaldised ehk järgnev kood ei kompileeru:

```
Expression<Func<int, int>> incrExpression = x => { return x + 1; };
```

Küll aga on lubatud:

```
Expression<Func<int, int>> incrExpression = x => x + 1;
```

Expression tüüpi muutujale avaldise omistamise puhul ei ole tegu mitte viitega funktsioonile, vaid kompilaator destruktureerib etteantud avaldise puuks, mis koosneb kõikidest avaldise primitiividest ehk teisisõnu lubab vaadelda koodi kui andmeid.



Joonis 4. Kompilaatori poolt loodud avaldiste puu eelnevalt kirjeldatud avaldisele

Taoline avaldiste puu on aluseks loomaks tõlkija, mis on suuteline programmikoodi teisendama graafikaprotsessorile sobivaks standardiseeritud koodiformaadiks *SPIR-V*.

Kui tavalise *LINQ* päringu käivitamisel rakendatakse ühte protsessori tuuma, siis *Microsoft* on loonud täiendava teegi, mis lubab ka mitme tuuma kasutamist. Selle teegi nimi on *Parallel LINQ* ehk *PLINQ* [19]. Idee seisneb selles, et paralleeltöötluse üksikasjad on kasutaja jaoks nähtamatud. Ülaltoodud näite jätkuks näeks päring paralleeltöötluse rakendamise korral välja järgnev:

```
IEnumerable<string> nameQuery = names.AsParallel()
    .Where(name => name[0] == 'M');
```

Ainuke erinevus peitub meetodi *AsParallel* väljakutses, mis seab päringu üles nii, et päringu käivitamisel toimub see võimaluse korral mitmel *CPU* tuumal. Mitme lõime rakendamine, töö jagamine nende vahel ja hiljem tulemuste kokku korjamisega kaasneb lisakulu, mistõttu väikeste andmehulkadega võib *PLINQ* rakendamine jõudlusele hoopis negatiivselt mõjuda. Punktis 5.1 teostatud jõudlusvõrdluses kaasatakse ühe meetodina ka *Parallel LINQ*.

Käesolev alampeatükk andis ülevaate *.NET LINQ* taustast, kirjeldas ära põhilised pakutavad päringuoperaatorit ning eristas *IEnumerable* ja *IQueryable* liidest, millest viimasest arusaamine on oluline just töö teema jaoks. Lisaks anti ülevaade terminist *PLINQ*.

3.2 SPIR-V ja GLSL

SPIR-V (*Standard Portable Intermediate Representation*) on binaarne vahekeel graafika varjutajate (*shader*) ja arvutustuumade (*compute kernel*) kirjeldamiseks. *SPIR-V* moodul võib omada mitut sisendpunkti ning koosneb standardis paika pandud instruktsioonidest. Need instruktsioonid ei ole sõltuvad ühestki konkreetsest riistvarast. Standardi lõi *Khronos Group* eesmärgiga luua lihtne vaheformaat, mis võimaldaks kõiki nende grupi tehnoloogiate poolt vajalikku funktsionaalsust, millel on läbinähtav ja sisutihe spetsifikatsioon ning mida on mugav sihtida kõrgema taseme keelte poolt nagu *GLSL* (*OpenGL Shading Language*) või *HLSL (High Level Shading Language*) [21].

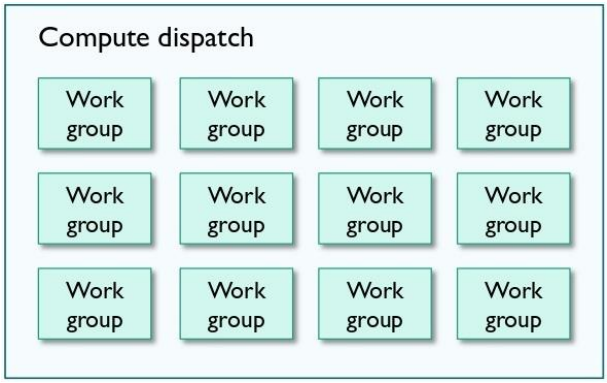
Käesoleva töö kontekstis oleks jõudluse poole pealt oleks kõige tõhusam tõlkida *LINQ* avaldis otse keelde *SPIR-V*. Teisalt, kuna tegu on binaarse keelega, teeb see genereeritud väljundkoodist arusaamise ja programmi silumise äärmiselt keeruliseks. Seetõttu on valitud sihtmärgiks kõrgetasemelisem keel *GLSL*, mis on inimloetav ning millele on olemas *Khronos Group*'i poolt pakutav tööriist *glslangValidator* [22], mille abiga on võimalik genereerida *GLSL*'ist *SPIR-V*. Seetõttu, rääkides *SPIR-V*'st, peab rääkima ka *GLSL*'ist.

GLSL ehk *OpenGL Shading Language* ei ole mitte üks keel, vaid mitu üksteisega tihedalt seotud keelt mõeldud erinevate graafikakonveieri protsessoretappide programmeerimiseks. See tähendab, et *GLSL* sisaldab keeli *vertex*, *tessellation control*, *tessellation evaluation*, *geometry*, *fragment* ja *compute* etappide kirjeldamiseks [23]. Antud töö keskendub ainult *compute* etapile, sest ülejäänud on mõeldud graafiliste operatsioonide teostamiseks ning jäävad töö skoobist välja.

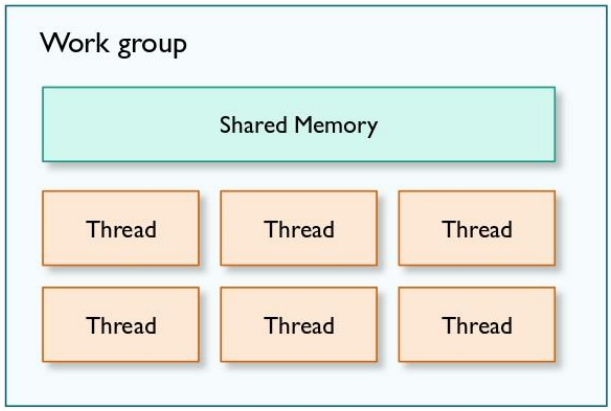
Compute ehk arvutusprotsessor on programmeeritav ühik, mis toimib iseseisvalt kõikidest ülejäänud varjutusprotsessoritest. Programmikood, mis on kirjutatud arvutusprotsessori jaoks nimetatakse *compute shader*'iks või arvutustuumaks (*compute kernel*). Esimest nime kasutatakse pigem graafiliste ning viimast mittegraafiliste rakenduste kontekstis.

Arvutustuumal on ligipääs rakenduse poolt ettemääratud ressurssidele nagu tekstuurid, puhvrid, muutujad ja loendurid. Tal ei ole väljundit ning tema tulemused tehakse nähtavaks läbi kõrvalnähtude etteantud muutujatesse (näiteks rakenduse poolt etteantud puhvrisse tulemuste kirjutamise).

Arvutustuum tegutseb tööühikute rühmadega (*group of work items*). Töörühm iseloomustab arvutustuuma käivituste kogumit, kus käivitused teostatakse potentsiaalselt paralleelselt. Käivitus rühma siseselt lubab sama rühma liikmetel jagada andmeid läbi jagatud muutujate ning kasutada konstruktsioone mälu sünkroniseerimiseks teiste rühma liikmetega. Lisaks pakutakse arvutustuumale sisseehitatud muutujaid nagu näiteks globaalse ja lokaalse töörühma (*workgroup*) indeks ning üldine globaalne indeks. Need indeksid on kasulikud näiteks, et programmikoodis teada, millise puhvri elemendi peal parasjagu arvutust tuleb teostada [23].



Joonis 5. Töörühmad graafikaprotsessoril arvutuste teostamiseks



Joonis 6. Töörühma sisene jagatud mälu ja lõimed paralleelkäivituseks

Eelpool toodud *LINQ* avaldisele, mis suurendab täisarvude jada iga elementi ühe võrra, võiks vastata näiteks järgnev arvutustuuma kood:

```
#version 450

layout (binding = 0) buffer layout0
{
    int in_data[];
    int out_data[];
};

void main()
```

```
{
    out_data[gl_GlobalInvocationID.x] = in_data[gl_GlobalInvocationID.x] + 1;
}
```

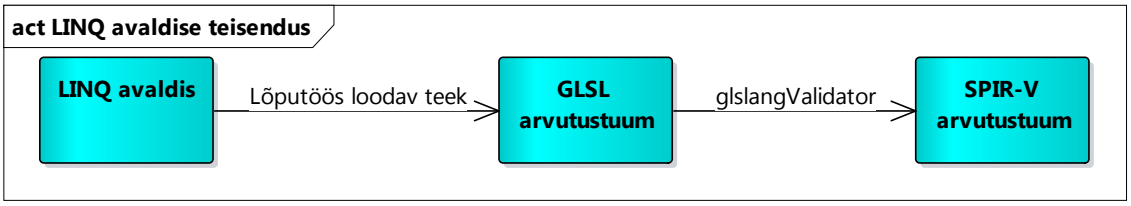
Version kirjeldab kompilaatorile *GLSL* versiooni, kus 450 vastab *GLSL* spetsifikatsiooni versioonile 4.50. *Layout* paneb paika mälu paigutuse ning kirjeldab ära sisendandmete massiivi *in_data* ja väljundandmete massiivi *out_data*. *Main* sektsioon sisaldab koodi, mis käivitatakse iga sisendelemendi peal. Sisendelement võetakse sisseehitatud muutuja *gl_GlobalInvocationID* põhjal, mis määrab ära käesoleva käivituse indeksi [23]. Võtmesõna *buffer* näol on tegemist *Shader Storage Buffer Object* tüüpi objektiga hoidmaks sisend- ja väljundandmeid. Alternatiivina võiks kasutada võtmesõna *uniform*, mis viitab *Uniform Buffer Object* tüüpi objektile. Küll aga lubab esimene neist hoiustada tunduvalt suuremaid andmehulkasid jõudluse arvelt [24].

Kasutades tööriista *glslangValidator*, saab ülaltoodud koodi kompileerida formaati *SPIR-V*. Lisaks binaarsele väljundile, on võimalik väljastada ka tekstikujuline inimloetav kirjeldus binaarkoodist, mis annab ülevaate binaarkoodis esinevatest instruktsioonidest:

```
// Module Version 10000
// Generated by (magic number): 80001
// Id's are bound by 29

                              Capability Shader
               1:             ExtInstImport  "GLSL.std.450"
                              MemoryModel Logical GLSL450
                              EntryPoint GLCompute 4  "main" 16
                              ExecutionMode 4 LocalSize 1 1 1
                              Source GLSL 450
                              Name 4  "main"
                              Name 9  "layout0"
                              MemberName 9(layout0) 0  "in_data"
                              MemberName 9(layout0) 1  "out_data"
                              Name 11  ""
                              Name 16  "gl_GlobalInvocationID"
                              Decorate 7 ArrayStride 4
                              Decorate 8 ArrayStride 4
                              MemberDecorate 9(layout0) 0 Offset 0
                              MemberDecorate 9(layout0) 1 Offset 0
                              Decorate 9(layout0) BufferBlock
                              Decorate 11 DescriptorSet 0
                              Decorate 11 Binding 0
                              Decorate 16(gl_GlobalInvocationID) BuiltIn
GlobalInvocationId
               2:             TypeVoid
               3:             TypeFunction 2
               6:             TypeInt 32 1
               7:             TypeRuntimeArray 6(int)
               8:             TypeRuntimeArray 6(int)
      9(layout0):             TypeStruct 7 8
              10:             TypePointer Uniform 9(layout0)
              11:     10(ptr) Variable Uniform
              12:      6(int) Constant 1
              13:             TypeInt 32 0
              14:             TypeVector 13(int) 3
              15:             TypePointer Input 14(ivec3)
16(gl_GlobalInvocationID):     15(ptr) Variable Input
              17:     13(int) Constant 0
              18:             TypePointer Input 13(int)
              21:      6(int) Constant 0
              24:             TypePointer Uniform 6(int)
         4(main):           2 Function None 3
               5:             Label
              19:     18(ptr) AccessChain 16(gl_GlobalInvocationID) 17
              20:     13(int) Load 19
              22:     18(ptr) AccessChain 16(gl_GlobalInvocationID) 17
              23:     13(int) Load 22
              25:     24(ptr) AccessChain 11 21 23
              26:      6(int) Load 25
              27:      6(int) IAdd 26 12
```

```
28:     24(ptr) AccessChain 11 12 20
                Store 28 27
                Return
                FunctionEnd
```

Nagu näha on väljund üsna verboosne, mis teeb temast arusaamise tülikaks. Seetõttu piirdutakse *LINQ* avaldiste tõlkimisel keelega *GLSL* ning kasutatakse binaarkoodi loomisel ära juba olemasolevaid tööriistu. *SPIR-V* loomist saab visualiseerida järgnevalt:



Joonis 7. LINQ avaldise teisendamine formaati SPIR-V

Käesolev alampeatükk kirjeldas ära keele *SPIR-V*, mida kasutatakse graafikariistvara vastu programmeerimisel. Kuna tegu on binaarse keelega, siis *LINQ* päringu tõlkimise ja silumise lihtsustamiseks võetakse veel lisaks kasutusele vahekeel *GLSL*. *GLSL* puhul on tegu inimloetava keelega ja keele struktuur kaardistub hästi keelega *C#*. *GLSL* tõlgitakse hiljem *SPIR-V* formaati läbi tööriista *glslangValidator*.

3.3 Vulkan

Kuna *SPIR-V* puhul on tegu ainult keele ja formaadi kirjeldusega, siis jääb vajadus veel teegi järele, mis suudab selles formaadis loodud programmi käivitada graafikaprotsessoril. Selleks võetakse kasutusele *Vulkan* nimeline teek.

Vulkan pakub programmeerimisliidest graafika- ja arvutusriistvarale. Liides koosneb käskudest, mis lubavad programmeerijal määrata varjutaja programme (*shader program*), arvutustuumasid, objekte ja operatsioone loomaks graafikaprotsessoril kõrgekvaliteedilisi pilte või teostada arvutusi [5].

Vulkan nagu ka *OpenGL* on *Khronos Group*'i poolt loodud programmeerimisliides ning *Vulkan* on mõeldud *OpenGL* järeltulijaks, tulevikus asendades *OpenGL* täielikult. Tegu on nullist üles ehitatud *API*'ga lahendamaks mitmeid fundamentaalseid probleeme, mis on eriti rõhutatud mobiilsetel seadmetel. *Vulkan*'i eesmärgiks on vähendada liidese kasutamisest tekkivat lisakulu (*overhead*), lubada otsesemat ligipääsu

graafikaprotsessorile ja vähendada *CPU* kasutust. Palju vastutust on liigutatud draiverilt rakenduse arendajale, et tagada lihtsam ja stabiilsem draiverite töö olenemata riistvarast. Lisaks on erinevalt *OpenGL*'ile *Vulkan*'i programmeerimisliides täpselt sama olenemata kas luuakse rakendust mobiilseadmele või lauaarvutile [25].

Järgnevalt antakse kiire ülevaade *Vulkan*'i käivitusmudelist. *Vulkan* võimaldab ligipääsu *GPU* füüsilistele seadmetele, kus riistvara toetab vähemalt *Vulkan 1.0* spetsifikatsioonis paika pandud nõudeid ning millele operatsioonisüsteemis leiduvad tootjapoolsed *Vulkan*'i draiverid. Füüsiline seade pakub ühe või rohkem järjekordasid (*queue*), mis võivad omavahel teha tööd asünkroonselt. Hulk järjekordasid on grupeeritud perekondadeks (*queue family*), kus perekond määrab ära järjekordade võimaldatava funktsionaalsuse. Funktsionaalsus jaguneb neljaks: graafika, arvutus, ülekanne (*transfer*) ja hõre mälu (*sparse memory*) [5].

Seadme mälu hallatakse rakenduse poolt. Iga seade võib reklaamida välja ühe või rohkem mälukuhja (*memory heap*), kus iga mälukuhi iseloomustab erinevat mälu piirkonda. Näited võimalike mälu piirkondade kohta:

- Lokaalne seadmele (*device local*) – mälu, mis on füüsiliselt ühenduses graafika seadmega (näiteks videomälu).
- Lokaalne seadmele, nähtav rakendusele (*host visible*) – mälu, mis on füüsiliselt ühenduses graafika seadmega ning kuhu on võimalik programmikoodist otse kirjutada.
- Lokaalne rakendusele (*host local*), nähtav rakendusele – mälu, mis on lokaalne rakendusele (näiteks suvapöördusmälu) ning nähtav nii rakendusele kui ka seadmele.

Vulkan'i rakendus kontrollib seadet läbi käsupuhvrite (*command buffer*) saatmise seadme järjekorda. Käsupuhver võimaldab eelnevalt salvestada ühe või rohkem seadme käsku ning anda need korraga seadmele täitmiseks. Käsupuhvrit võib rakendada seadmel üks või rohkem kordi. Mitut käsupuhvrit võib ehitada paralleelselt, et rakenduse poolt ära kasutada mitut lõime [5].

Käsupuhvrid, mis saadetakse erinevatesse järjekordadesse võivad käivitud paralleelselt ning nende käivitamise järjekord ei ole määratud. Ühte ja samasse järjekorda saadetud käsupuhvrid austavad saatmise järjekorda ning käivituvad etteantud järjekorras. Nii kui käsupuhver on saadetud järjekorda, antakse rakendusele koheselt kontroll tagasi ehk ei oodata käskude täitmist seadmel. Tegevuste korrektsekt sünkroniseerimiseks seadmel pakub liides välja sünkroniseerimisprimitiivid nagu tara (*fence*), semafor (*semaphore*) ja sündmus (*event*) [5].

Antud teema puhul on programmeerimisliidese valimiseks graafikaprotsessoril arvutuste tegemiseks võimalusi mitmeid. Järgnevalt võrreldakse kvalitatiivsete näitajate põhjal levinumaid liideseid *Vulkan*, *CUDA*, *OpenCL*, *OpenGL*, *DirectCompute* [26].

Probleemide skaleeruvus

CUDA, Vulkan ja *OpenCL* soovitavad pärida füüsiliselt seadme võimalusi ja piiranguid ressursside ja funktsionaalsuse osas. *DirectCompute* ja *OpenGL* on aga standardiseeritud vastavalt versiooni järgi. Näiteks jagatud mälu suuruse kohta võimaldab *CUDA* seda täielikult ära kasutada, kuid *DirectCompute* seab piirangu *API*-defineeritud suurusele. Piiratud ressursside eelis on *API* kasutamise lihtsus – ei ole vaja pärida nii palju infot seadmelt, vaid *API* annab garantii muutujate osas. See aga seab piirid ette suuremate andmemahtude korral ning ei luba rakendusel seadme potentsiaalselt võimekust maksimaalselt ära kasutada.

Porditavus

OpenCL ja Vulkan'i võtmeomaduseks on nende kõrge porditavus erinevate platvormide ja erinevate riistvaratootjate seadmete vahel, kusjuures *OpenCL* töötab ka vanemal riistvaral, kus *Vulkan*'i puhul puuduvad tihti draiverid, sest tegu nii uue liidesega või riistvara ei saagi toetada *Vulkan*'it spetsifikatsioonis paika pandud miinimumnõuete tõttu. *DirectCompute* on piiratud *Windows*'i platvormidele ning *CUDA NVIDIA* riistvarale, mis porditavuse koha pealt ei loo neile just kõige ahvatlevamat kuvandit.

Programmeeritavus

CUDA's on kõige lihtsam luua realisatsiooni, sest nõuab kõige vähem koodi seadistamiseks. Lisaks on olemas suur kommuun ja *API* küpsuse tõttu leidub palju

abimaterjale. *OpenCL*'i või *Vulkan*'i implementatsioon ei ole nii lihtne, sest on vaja näiteks käsitsi valida õige füüsiline seade ning luua järjekord käskude saatmiseks.

DirectCompute ja *OpenGL* kannatavad graafika spetsiifiliste abstraktsioonide tõttu. Näiteks on vaja teostada lisasammud, et luua ja kasutada mälu arvutustuumas võrreldes *CUDA* või *OpenCL*'iga.

OpenGL'i suurimaks puudujäägiks on probleem, et rakenduse autoril pole võimalik valida, millist füüsilist graafikaprotsessorit masinas kasutada – selle määrab operatsioonisüsteem. *Windows* 10 puhul tähendab see, et graafikakaart peab olema ühendatud monitoriga ning monitor peab olema määratud kui peamine (*primary*) kuvar.

Antud töös sai valitud liideseks *Vulkan* tema suhteliselt hea porditavuse ning samuti modernsuse tõttu võrreldes *OpenGL*'iga. *Vulkan*'i eelis *OpenCL*'i ees on tema ühine liides nii graafika kui ka arvutusoperatsioonidest, mis võimaldab graafika seadmel paiknevaid ressursse ja mälu otse ka graafilisteks operatsioonideks kasutada (näiteks pilditöötluse puhul arvutustuumal tulemi kuvamiseks ekraanile ilma lisa sünkroniseerimiste või kopeerimisteta).

Käesolevas alampeatükis võrreldi erinevaid teeke, mille kaudu on võimalik suhelda graafikaprotsessoriga. Esile toodi *Vulkan* liides ning anti ülevaade põhikontseptidest ja töövoost.

4 Realisatsioon

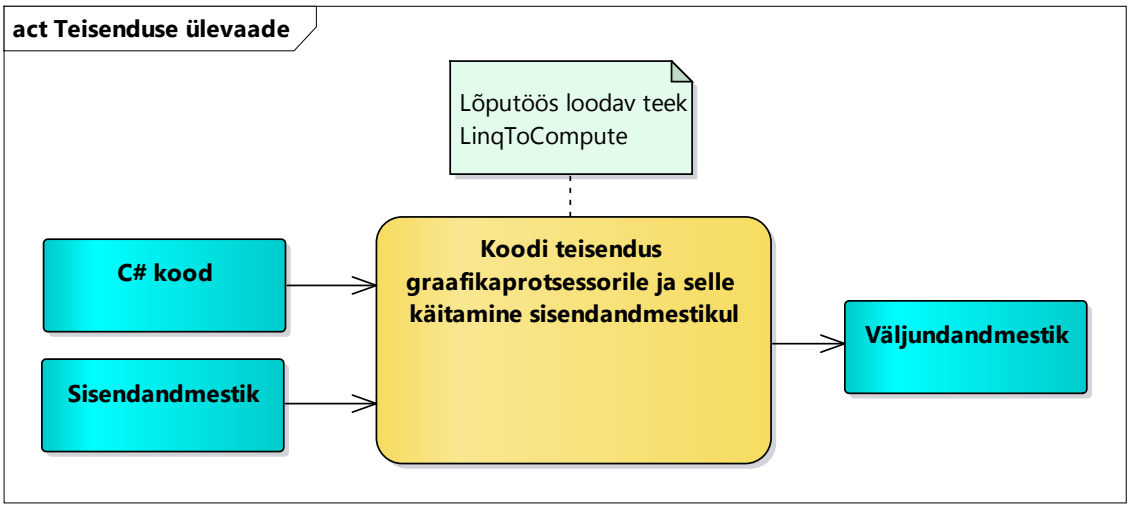
4.1 Ülevaade

Töös loodud teegid on avalikult kättesaadavad *GitHub* keskkonnas [27] [28]. Järgnev tabel annab ülevaate teekide kohta 05.05.2017 seisuga. Statistika failide ja ridade kohta on saadud tööriistaga *cloc* [29]. Statistika ei sisalda andmeid lahenduste test- ega näiteteekidest.

Nimi	*VulkanCore*	*LinqToCompute*
Lühikirjeldus	*.NET* keskkonna liidestus *Vulkan* graafika- ja arvutusteegiga	Võimaldab *.NET LINQ* päringuoperaatorite käitamise graafikakaardil
Kirjeldatud alampeatükkides	4.2	4.3 ja 4.4
Esmane *commit*	28.01.2017	19.03.2017
Allalaadimisi läbi *Nuget* paketihalduri	669	N/A
***Star*'e**	14	0
***Fork*'e**	3	0
***Pull Request*'e**	2	0
Koodifailide arv	81 (osaliselt genereeritud)	17
Koodiridade arv	10342 (osaliselt genereeritud)	1489
Kommentaariridade arv	13301 (osaliselt genereeritud)	85
Litsents	*MIT*	*MIT*

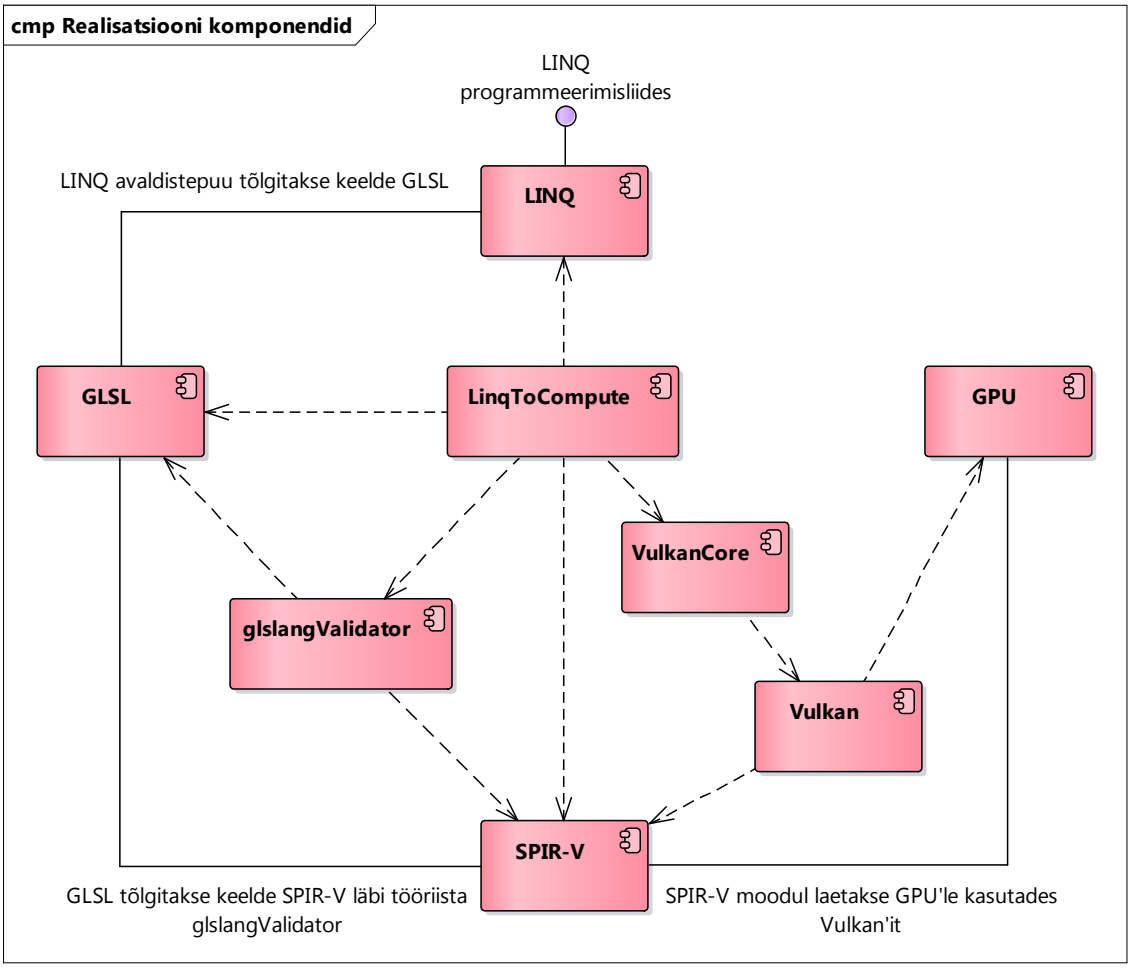
Tabel 2. Töös loodud teekide ülevaade

Kõige kõrgemal tasandil võib realisatsiooni käsitleda kui vahendida, mis suudab käivitada sisendiks antava andmestiku koos sellele rakendatud programmikoodiga *GPU*'l.



Joonis 8. Realisatsiooni ülevaade

Järgnev joonis annab ülevaate, kuidas on eelnevas peatükis kirjeldatud tehnoloogiad seotud loodavate teekidega.



Joonis 9. Realisatsiooni teekide ja tehnoloogiate omavahelised seosed

4.2 Vulkani liidestus .NET'iga

Vulkan on *C* keeles kirjutatud teek ning kompileeritud masinkoodiks igale toetatud platvormile. Juhul, kui *Vulkan*'i käitusteek on seadistatud, peitub ta *Windows*'i platvormidel *vulkan-1.dll* ning *Linux* platvormidel *libvulkan.so* nimelistes failides. Tegu on haldamata (*unmanaged*) koodiga võrreldes *.NET* platvormi keelte kompileeritud hallatud koodiga (*managed*). Keskkondade erinevuste tõttu ei ole hallatud maailmast võimalik otse välja kutsuda haldamata maailma funktsioone. See tähendab, et *.NET* teegis *Vulkan* liidese kasutamiseks on vaja luua sidemed (*bindings*) kasutades *.NET* raamistiku poolt pakutavaid võimalusi.

Vulkan'i puhul on tegu suhteliselt uue teegiga, mille väljalase oli 2016. aasta veebruaris [30]. Seetõttu ei leidunud lõputöö tegemise ajal veel lõplikult valmis sidemeid, mille abil oleks võimalik *.NET* keskkonnast liidest kasutada. Olemasolevad sidemete realisatsioonid ei pakkunud ligipääsu kõikidele *Vulkan*'i käskudele ning nendes esines fundamentaalseid probleeme nagu mälu leke, puuduv koodi dokumentatsioon või ebakorrektne *UTF-8* kodeeringus sõnede käitlemine. Küll aga avatud lähtekoodi kommuuni poolt töö käis selles vallas [31] [32]. Eelnimetatud probleemide tõttu loob käesolev töö oma sidemed *Vulkan* teegiga.

Sidemete kergemaks loomiseks on *Khronos Group* avaldanud *Vulkan*'i programmeerimisliidese registri [33]. See on *XML* formaadis spetsifikatsioon, mis kirjeldab süstemaatiliselt kõik teegis leiduvad käsud, andmestruktuurid, loendid (*enumeration*), konstandid, laiendused jms. Järgnevalt on toodud näide registrist kirjeldatud käsust *vkGetPhysicalDeviceFeatures*:

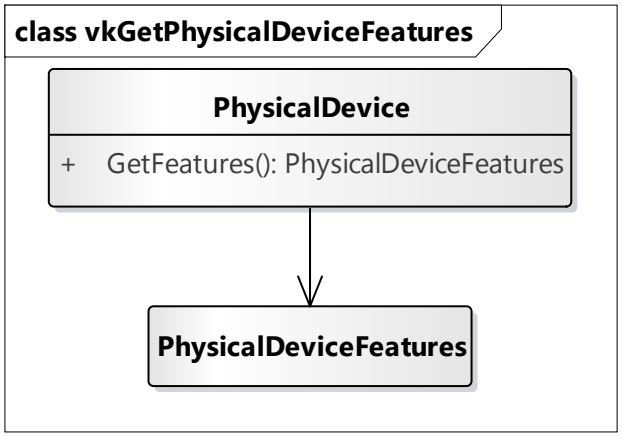
```
<command>
  <proto>
    <type>void</type>
    <name>vkGetPhysicalDeviceFeatures</name>
  </proto>
  <param>
    <type>VkPhysicalDevice</type>
    <name>physicalDevice</name>
  </param>
  <param>
    <type>VkPhysicalDeviceFeatures</type>* <name>pFeatures</name>
  </param>
</command>
```

Lisaks registrist võetavale liidese infole on saadaval käskude ja andmestruktuuride dokumentatsioon [33]. Dokumentatsioon on *asciidoc* vormingus ning võimaldab generaatoril automaatselt lisada ka *C#* koodi dokumentatsiooni. Näiteks eelneva registrikäsu jaoks leidub vastav dokumentatsioon:

```
// refBegin vkGetPhysicalDeviceFeatures Reports capabilities of a physical
device

  * pname:physicalDevice is the physical device from which to query the
    supported features.
  * pname:pFeatures is a pointer to a slink:VkPhysicalDeviceFeatures
    structure in which the physical device features are returned.
    For each feature, a value of ename:VK_TRUE indicates that the feature is
    supported on this physical device, and ename:VK_FALSE indicates that the
    feature is not supported.

// refEnd vkGetPhysicalDeviceFeatures
```

Vulkan'i puhul on tegu protseduurilise ja mitte objektorienteeritud liidesega. Ta koosneb hulgast andmestruktuuridest, mis ei oma mingisugust käitumist ning hulgast funktsioonidest (*Vulkan*'i terminite järgi käsud või *commands*), mis on otse väljakutsutavad. Kuna *.NET* arendajad on enamjaolt harjunud objektorienteeritud paradigmaga töötama, kaardistab ka käesolev töö *Vulkan*'i sidemed *C#* objektorienteeritud liideseks teatud kaardistusreeglite järgi. Graafikaprotsessori ressursside eluea haldamiseks rakendatakse *dispose* mustrit [34].

Vulkan esitab oma olemeid (näiteks füüsiline seade, järjekord, semafor) läbi pidemete (*handle*) ehk viitade, mis registris on kirjeldatud lihtsalt kui tüüpdefinitsioonid (*typedef*) viit-tüüpi muutujatele. Need samad pidemed võetakse ka aluseks sidemete kaardistamisel objektorienteeritud maailma, kus iga pideme jaoks luuakse klass. Klassi meetodid defineeritakse *Vulkan* käskude järgi nii, et käsu parameetrite loendi viimane (vasakult poolt lugedes) pide-tüüpi sisendmuutuja määrab ära klassi, mille külge käsk kuulub. Näiteks eeltoodud *XML* lõigu korral käsu *vkGetPhysicalDeviceFeatures* parameeter *VkPhysicalDevice* on pide-tüüpi, kuid *VkPhysicalDeviceFeatures* on tavaline andmestruktuur. See tähendab, et objektorienteeritud maailmas luuakse klass *PhysicalDevice*, millel on meetod *GetFeatures*. Nimetuste puhul järgitakse sihtkeskkonna, milleks antud juhul on *C#*, nimetustavasid [35].



Joonis 10. Vulkan käsu vkGetPhysicalDeviceFeatures kaardistamine objektorienteeritud maailma

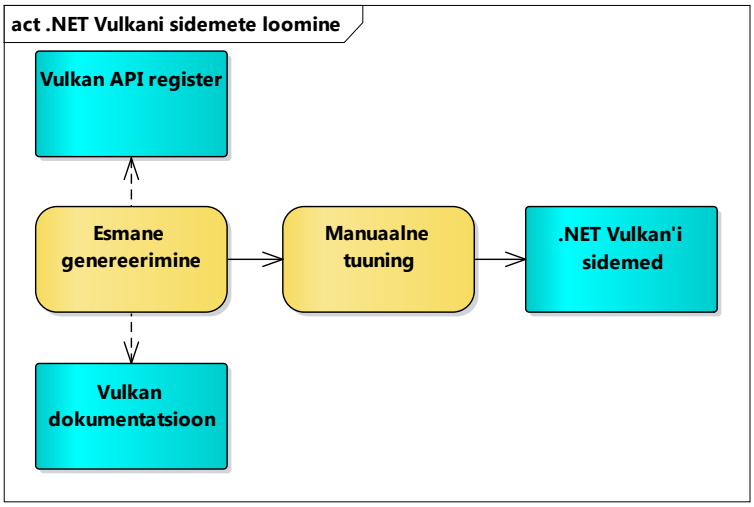
Vulkan'i programmeerimisliidese register loob tugeva aluse koodigeneraatori loomiseks, mis oleks võimeline automaatselt registri info põhjal looma *C#* sidemed. Seda varianti kasutavad kõik autori poolt teadaolevad alternatiivsed sidemed. Käesolev töö kasutab samuti koodigeneraatorit, sest vastasel juhul oleks manuaalset tööd liiga palju (lõputöö kirjutamise hetkel oli *Vulkan*'i registris ligi 250 kaardistamist vajavat käsku). Küll aga valitakse hübriidsem variant, kus generaatorit kasutatakse ainult esmaste sidemete loomiseks ning edasised muudatused genereeritud koodi peal on juba manuaalsed. Selle lähenemise eelised:

- Täielik kontroll koodi üle, mis lubab spetsiifilisemaid optimeerimisi sisse viia ning võimaldada grammatiliselt korrektset koodi dokumenteerimist.
- Kuna tegu on avatud lähtekoodiga projektida [27], teeb generaatori mittekasutamine võõrastel projekti panustamise tunduvalt lihtsamaks – vajalik on arusaamine ainult *Vulkan*'i liidesest ja mitte koodigeneraatorist.

Suurimad miinused:

- Nõuab manuaalseid muudatusi iga kord, kui *Vulkan*'i spetsifikatsiooni uuendatakse.
- Kuna manuaalsed muudatused on tehtud, siis ei saa lihtsalt kõiki sidemeid nullist uuesti genereerida – see teeb fundamentaalsete muudatuste tegemise keeruliseks.

Suures plaanis võib seega *Vulkan*'i *.NET* sidemete loomist ette kujutada järgneva joonise järgi:



Joonis 11. Vulkan'i .NET sidemete loomine

Masinkoodi väljakutsumiseks näiteks *.dll* failist pakub *.NET* erinevaid võimalusi. Järgnevalt võrreldakse kolme varianti. Oluline on kaaluda erinevaid variante, sest igal meetodil on omad head ja omad vead ning tuleb arvestada jõudluse erinevusi ning samuti kas meetod on toetatud kõikidel platvormidel. Jõudlus mängib olulist rolli, sest antud sidemete puhul on tegu õhukese kihiga ümber *Vulkan* liidese, mille ainus eesmärk on võimaldada ligipääs sellele liidesele ehk kogu töö seisneb masinkoodist käskude väljakutsumisel.

Esimene viis on *Platform Invoke* [36]. Tegu on *CLR* teenusega, mis lubab välja kutsuda funktsioone *CLR*'i poolt haldamata teekidest, mis sisaldavad masinkoodi, erinevalt hallatud teekidest, mille kood on vaheformaadi kujul *MSIL* [37]. Arendaja vaatevinklist kirjeldatakse meetodi signatuur kasutades võtmesõnu *static* ja *extern* ning dekoreeritakse meetod atribuudiga *DllImport*, milles on võimalik defineerida väljakutsutava teegi faili nimi ning väljakutsutava funktsiooni nimi:

```
[DllImport("vulkan-1.dll", EntryPoint = "vkGetPhysicalDeviceFeatures")]
static extern void GetFeatures(IntPtr physicalDevice, PhysicalDeviceFeatures*
features);
```

Üheks suureks miinuseks *Platform Invoke* juures on asjaolu, et laetav teek ei pruugi olla sama nimega kõikidel platvormidel. Näiteks *Vulkan*'i puhul on *Windows* platvormil teegi nimeks *vulkan-1.dll*, kuid *Ubuntu* või *Androidi* peal *libvulkan.so*. *DllImport* ei võimalda defineerida rohkem kui ühte faili nime, mis tähendab, et mitme platvormi toetamiseks on vaja *extern* meetodeid dubleerida:

```
[DllImport("vulkan-1.dll", EntryPoint = "vkGetPhysicalDeviceFeatures")]
```

```
static extern void GetFeaturesWin32(IntPtr physicalDevice,
PhysicalDeviceFeatures* features);

[DllImport("libvulkan.so", EntryPoint = "vkGetPhysicalDeviceFeatures")]
static extern void GetFeaturesLinux(IntPtr physicalDevice,
PhysicalDeviceFeatures* features);
```

Käitamisajal tuleb vastavalt platvormile välja kutsuda vastav meetod:

```
if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
    GetFeaturesWin32(physicalDevice, features);
else
    GetFeaturesLinux(physicalDevice, features);
```

See lisab lähtekoodi palju trafaretset (*boilerplate*) koodi ning nõuab hargnevusinstruktsiooni iga *Vulkan* käsu väljakutsel.

Teine variant on kasutada ära operatsioonisüsteemi teeke *kernel32.dll Windows*'il ja *libdl.so Linux*'il ning nende abil laadida sisse *Vulkan*'i teek ja pärida sealt viited käskudele. Eelnimetatud operatsioonisüsteemi teekide laadimine käiks endiselt läbi *Platform Invoke* teenuste, kuid edasi on võimalik pakkuda abstraktsiooni, mis võimaldab kõiki *Vulkan*'i käske välja kutsuda ühest punktist.

```
[DllImport("kernel32", EntryPoint = "LoadLibrary")]
static extern IntPtr Kernel32LoadLibrary(string fileName);

[DllImport("kernel32", EntryPoint = "GetProcAddress")]
static extern IntPtr Kernel32GetProcAddress(IntPtr module, string procName);

[DllImport("libdl.so", EntryPoint = "dlopen")]
static extern IntPtr LibDLLoadLibrary(string fileName, int flags);

[DllImport("libdl.so", EntryPoint = "dlsym")]
static extern IntPtr LibDLGetProcAddress(IntPtr handle, string name);
```

Laetud *Vulkan*'i käsuviitele deklareeritakse käsule vastavat tüüpi delegaat ning defineeritakse delegaadi põhjal muutuja, mida on võimalik rakendusel välja kutsuda.

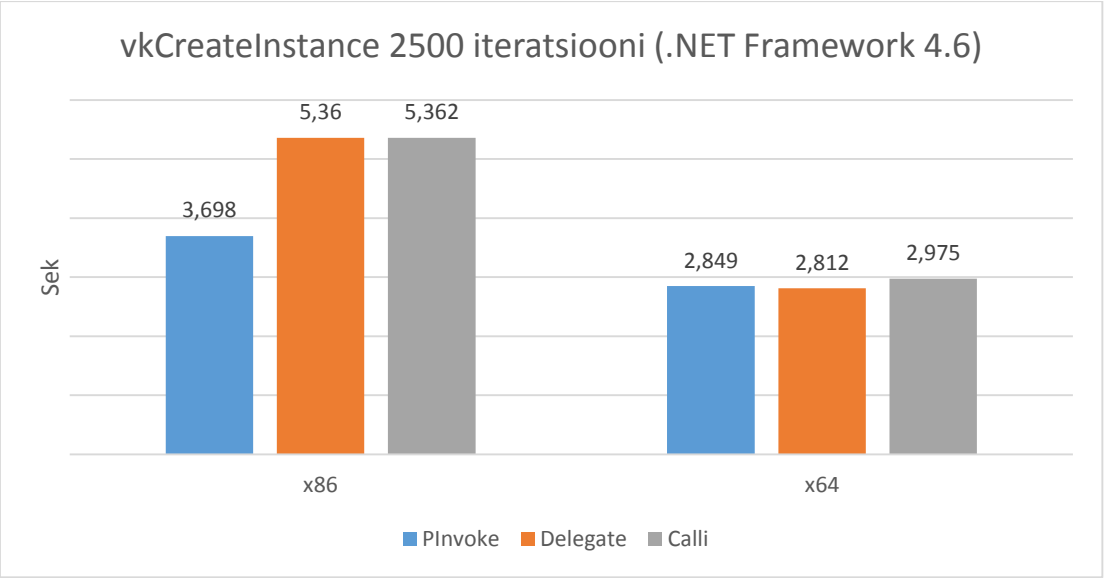
```
delegate void vkGetPhysicalDeviceFeaturesDelegate(IntPtr physicalDevice,
PhysicalDeviceFeatures* features);

Marshal.GetDelegateForFunctionPointer<vkGetPhysicalDeviceFeaturesDelegate>(fu
nctionPointer);
```

Marshal.GetDelegateForFunctionPointer on *.NET* raamistiku abimeetod, mis oskab funktsiooni viitest luua tugevalt tüpiseeritud delegaatmuutuja, mille abil on võimalik turvaliselt funktsiooni välja kutsuda.

Selle variandi eelis esimese ees on, et kaob vajadus duplitseerida koodi iga platvormi kohta.

Kolmas variant on alternatiiv teisest ja rõhub põhiliselt jõudluse parendamisele. See on meetod, mis on kasutusel mitmetes tuntumates *.NET* graafikaliidestest [38] [39]. Väidetavalt on *Marshal.GetDelegateForFunctionPointer* meetodi abil loodud delegaatmuutuja väljakutsumine aeglane, sest lisaks originaalfunktsiooni väljakutsele teostatakse ka rida turvakontrolle, mis keelavad ära väärkasutusjuhud masinkoodi väljakutsel. Rõhk on sõnal „väidetavalt“, sest lõputöö käigus teostatud jõudlustest (sellest allpool) seda väidet ei kinnita, mis võib olla tingitud *.NET* raamistiku versioonide eripärast. Selle variandi puhul jäetakse ära *Marshal.GetDelegateForFunctionPointer* ning luuakse hoopis tühi meetod, mille kehand implementeeritakse *post-process* sammuna peale kompileerimist. Idee on kasutada *MSIL* instruktsiooni *calli* [40], mis kutsub funktsiooniviite välja otse ilma lisategevusteta. Kahjuks *C#* programmeerimiskeel ei võimalda seda instruktsiooni väljendada, mistõttu langetakse *post-process* sammule. Selle sammu käigus laetakse kompileeritud teek ning valitud kohtadesse lisatakse operatsioonikood *calli*.

Aitamaks valida eeltoodud kolme variandi vahel, sai teostatud jõudlustest võttes aluseks *Vulkan*'i käsk *vkCreateInstance* [41]. Test viidi läbi *Windows 10* platvormil klassikalisel *.NET Framework*'il (alternatiividena on *cross-platform* implementatsioonid *Mono* või *.NET Core*). Teostati 2500 iteratsiooni iga variandiga nii 32- kui ka 64-bitise rakenduse korral.



Joonis 12. Erinevate masinkoodi väljakutsumise viiside jõudlustesti tulemused

Testist järeldub, et 64-bitise rakenduse puhul on masinkoodi väljakutsumine märkimisväärselt kiirem ning kõik variandid on jõudluse puhul võrdsed (teatud vea piires). 32-bitise rakenduse korral oli kiireim *Platform Invoke*. Huvitav on, et kolmas variant, mis peaks teoorias olema kõige optimaalsem, sai sama tulemuse teisega. Arvatavasti on tegu kas kompilaatori või *CLR*'i poolse optimeerimisega. Täpsemaks infoks peaks mõõtmisi teostama veel teiste *.NET Framework* versioonidega. Käesolev töö valib teise delegaadipõhise variandi, sest on *calli* instruktsiooni rakendamisega võrreldes lihtsam ning lähtekood jääb puhtam võrreldes *Platform Invoke* lähenemisega.

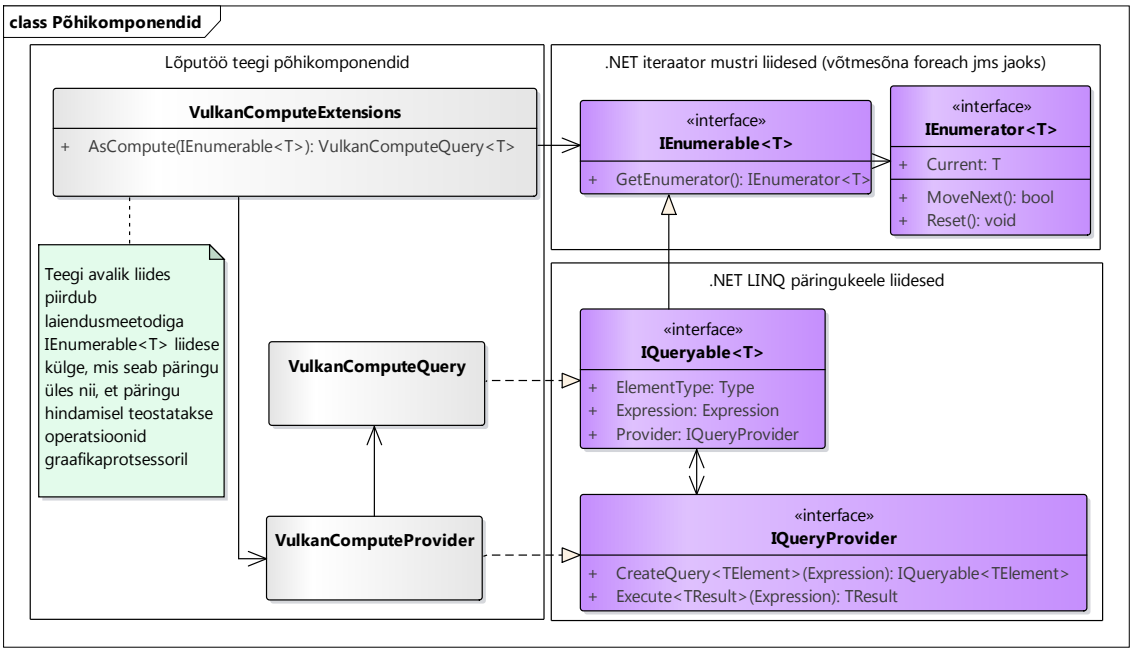
Selleks, et garanteerida sidemete teatud funktsionaalne korrektsus, luuakse testid eesmärgiga saavutada täieliku lauseadekvaatsuse kriteeriumid *Vulkan*'i tuumkäskudel. Kuna *Vulkan*'i puhul on tegu kolmandate osapoolte poolt laiendatava teegiga, on lisaks tuumkäskudele registris kirjeldatud ka terve hulk laiendustest tulenevaid käske. Nende käskude testimine jääb skoobist välja, sest nõuab tihti spetsialiseeritud riistvara ja/või tarkvara olemasolu. Näiteks *AMD* spetsiifilisi laienduskäske ei ole võimalik testida *NVIDIA* riistvaral.

Käesolev alampeatükk tõi välja probleemi, et *Vulkan*'i puhul on tegu *C* keeles kirjutatud teegiga, mistõttu ei ole teegi kasutamine hallatud *.NET* keskkonnast triviaalne. Toodi välja, kuidas see probleem lahendati ning kuidas protseduurilisest liidesest objektorienteeritud liides loodi.

4.3 LINQ avaldise teisendamine keelde SPIR-V

Käesolevas alampeatükis kirjeldatakse ära teisendusprotsess *LINQ* avaldisest *SPIR-V* mooduliks. Alampeatükis 3.2 kirjeldatud põhjustel ei teisendata *LINQ* avaldist otse sihtformaati, vaid esmalt keelde *GLSL* ning seejärel läbi tööriista *glslangValidator* [22] formaati *SPIR-V*. Seega piirdub kirjeldus enamjaolt teisendusega vaheetappi.

Suur eelis on asjaolu, et terve teegi avalik liides piirdub üheainsa laiendusmeetodiga *AsCompute* .*NET* raamistiku baasteegi liidese *IEnumerable* külge. Enne detailsemat kirjeldust antakse joonise põhjal ülevaade, kuidas teegi põhiklassid on seotud raamistiku liidestega. *IEnumerable* ja *IQueryable* on võtmeliidesed ning kirjeldatud alampeatükis 3.1.



Joonis 13. Töös loodavate põhikomponentide liidestus

Andmestik, mille kallal soovitakse arvutusi läbi viia, saab alguse alati rakenduse poole pealt. Pea kõik .*NET* raamistiku tüübid, mis iseloomustavad andmehulka, implementeerivad *IEnumerable* liidest, mis garanteerib, et tüüp tagab iteratsiooni funktsionaalsuse üle andmehulga. See on sobilik liides, mida aluseks võtta, sest katab ära enamikud andmehulgad, mille peal võib kasutajal soovi olla riistvara poolt kiirendatud arvutusi teostada. *AsCompute* ülesanne on triviaalne: võtab sisendandmejada, loob uue instantsi klassist *VulkanComputeProvider* (teisendused ja päringu käivitamine toimub seal) ja mähib need uude *VulkanComputeQuery* instantsi. *VulkanComputeQuery*

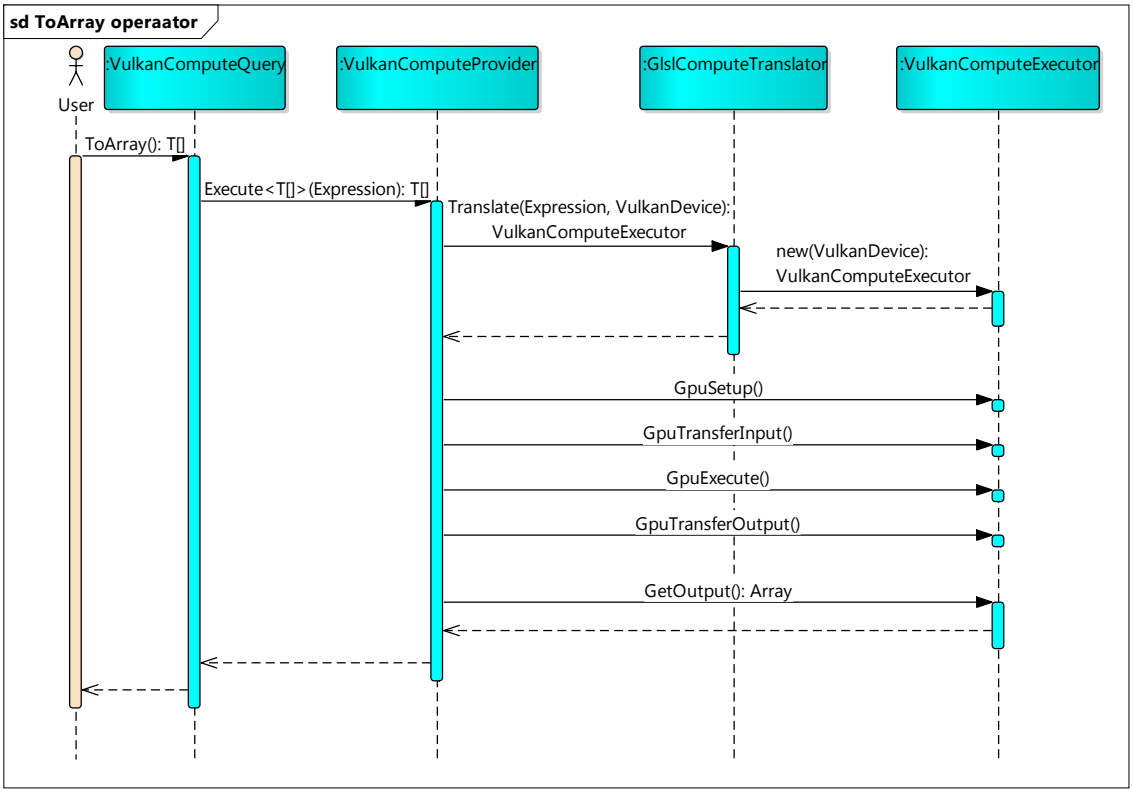
iseloomustab päringut, mida soovitakse kiirendada ning kannab endaga kaasas vajalikku informatsiooni päringute teisendamiseks ja arvutuse läbiviimiseks *VulkanComputeProvider* klassi poolt.

```
public static VulkanComputeQuery<T> AsCompute<T>(
    this IEnumerable<T> source)
{
    return new VulkanComputeProvider(VulkanDevice.Default)
        .CreateQuery<T>(source.AsQueryable().Expression);
}
```

Võtmesõna *this* määrab ära, et tegu on laiendusmeetodiga liidese *IEnumerable* külge. Iga päringu jaoks luuakse uus *VulkanComputeProvider*, mis vastutab päringu loomise eest. Kui arvutis on rohkem kui üks graafikaprotsessor, tekib valikuvõimalus, et millise protsessori peal päringut käivitada. Antud juhul võetakse alati vaikimisi *Vulkan*'i draiveri poolt tagastatud esimene protsessor. Seda väljendab *VulkanDevice.Default*, millest on täpsemalt juttu alampeatükis 0. Võimalik edasiarendus võiks lasta ka kasutajal soovi korral protsessorit määrata. *AsQueryable* mähib *IEnumerable* andmejada tüüpi, mis implementeerib liidest *IQueryable*, mille avaldiste puu (*expression tree*) koosneb ühest konstantavaldisest (*ConstantExpression*), mille väärtus on seesama andmejada. *VulkanComputeProvider.CreateQuery* tagastab uue päringu:

```
VulkanComputeQuery<TResult> CreateQuery<TResult>(Expression expression)
{
    return new VulkanComputeQuery<TResult>(this, expression);
}
```

Edasi on võimalik tagastatud instantsi peal rakendada kõiki *LINQ* päringuoperaatoreid, et ehitada üles graafikaprotsessori päring. Antud töös on realiseeritud päringuoperaatorid *Select* ja *Zip*. Päring on viitkäivitusega (*deferred execution*) ehk päringu teisendus, käivitamine ja väljundi tagastamine käivitub, kui kasutaja alustab päringu itereerimist, rakendab mõnda agregeerimisoperaatorit (näiteks *Count*, *Max*, *Average*) või küsib tagasi väljundi massiivi või nimekirja operaatoritega *ToArray* või *ToList*. Edasi vaadeldakse käivitust operaatori *ToArray* kaudu.

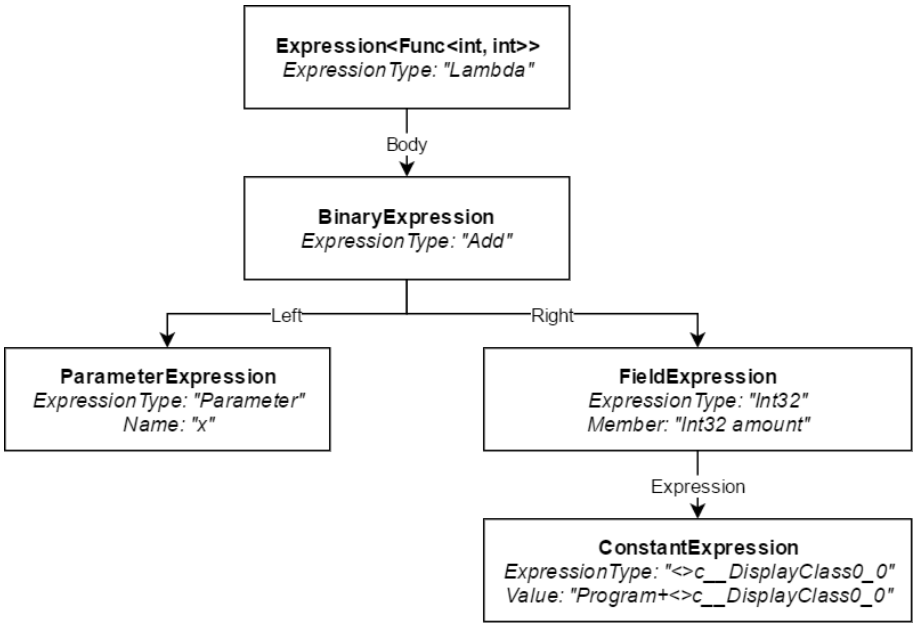


Joonis 14. ToArray operaatori käivitamise jadadiagramm

ToArray väljakutsel antakse kompilaatori poolt loodud *LINQ* avaldiste puu läbi *Expression* tüüpi muutuja ette *VulkanComputeProvider* instantsile. Puu interpreteerimine toimub kahekordse puu läbimisega. Puu läbimine on teostatud läbi *visitor* mustri [42]. Esmase läbimise eesmärk on avaldiste puu lihtsustamine. See lihtsustamine on rakendatav igasugustele probleemidele ja ei oma ühtegi graafikaprotsessorile tõlkimise heuristikat. Teine läbimine on juba spetsiifilisem ning tegeleb konkreetselt *GLSL* arvutustuuma koodi väljastamisega.

Lihtsustamine on vajalik lokaalsete ehk pinus asetsevate muutujate tõlkimiseks. Näiteks järgneva päringu korral on avaldise puu puhul märgata probleem:

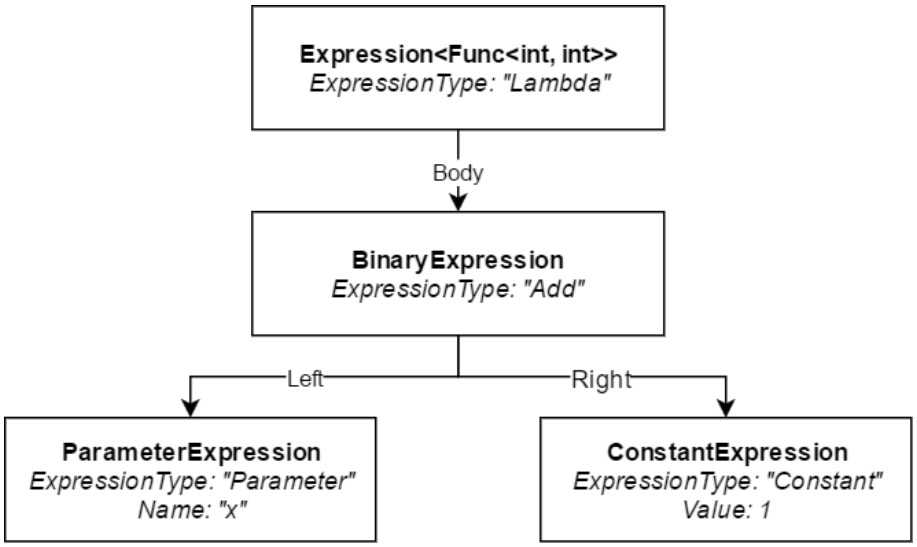
```
int amount = 1;
Expression<Func<int, int>> increment = x => x + amount;
```



Joonis 15. Avaldiste puu lokaalse muutuja korral

Kuna lokaalse muutuja *amount* skoop on piiratud tema plokiga, peab kompilaator looma uue klassi, mille abil muutuja eluiga pikendatakse. See on eelnevalt jooniselt näha kahe parempoolse avaldise kaudu. Selline käitumine on tuntud kui sulund (*closure*). Kahjuks teeb see avaldise interpreteerimise keeruliseks, sest järsku on vajalik interpreteerida anonüümset klassi, mille enda ja tema väljade nimed on automaatselt genereeritud. Genereeritud nimedele ei saa lootma jääda, sest nad pole spetsifikatsioonis paika pandud ning võivad erinevate kompilaatori versioonide vahel erineda [43]. Lisaks teeb see puu raskemini loetavaks.

Lihtsustamiseks on võimalik käia läbi alampuud. Juhul, kui alampuu ei sisalda ühtegi parameeter-tüüpi avaldist (*ParameterExpression*), on võimalik välja arvutada alampuu väärtus ning asendada terve alampuu ühe seda väärtust kandva konstant tüüpi avaldisega (*ConstantExpression*). Peale lihtsustamist näeb eelnev puu välja järgnev:



Joonis 16. Avaldiste puu peale lihtsustamist

Lihtsustatud avaldisele vastaks koodis järgnev lõik:

```
Expression<Func<int, int>> increment = x => x + 1;
```

Peale lihtsustamisi algab puu tõlkimine graafikaprotsessorile mõistetavale kujule, mida realiseerib *GlslComputeTranslator* klass. Siin jätab arhitektuur ruumi edasiarenduseks, kus nimetatud tõlkija asemel oleks võimalik luua tõlkija, mis teisendab puu otse formaati *SPIR-V*. Tõlkimine keelde *GLSL* on suures plaanis üsna otsekohene, sest *GLSL C*-tüüpi süntaks kaardistub väga hästi *C# C*-tüüpi süntaksile.

Pea kõik *C#* unaarsed operaatori on toetatud, välja arvatud *address-of* (&), *dereference* (*, ->) ja *type-cast* (mille asemel kasutatakse konstruktoreid). Samuti on toetatud kõik binaarsed operaatorid. Alates *GLSL* versioonist 1.3 on toetatud loogikaoperaatorid (&, |, ^, ~, <<, >>), *modulo* (%) ja neile vastavad *modify-assign* operaatorid (&=, |=, ^=, <<=, >>=, %=). [23]

Operandide järjekord jääb samaks ühe suure erinevusega, kus korrutamise puhul on järjekord vastupidine. Kuigi mõlema keskkonna puhul on mäluasetus vektoritel ja maatriksitel sama, siis vaadeldakse *GLSL*'is vektoreid kui veeruvektoreid (sarnaselt, nagu on tavaks matemaatikas) ja *C#* baasmatemaatika teegis kui reavektoreid. See tähendab, et maatriksite ja vektorite korrutamisel tuleb sama tulemuse saamiseks teostada kummaski keskkonnas korrutamisi vastupidises järjekorras. Tuleb olla ettevaatlik, sest üldjuhul maatriksite korrutamine ei ole kommutatiivne. [23]

```
// Order of matrix multiplication in C#
Matrix4x4 rotate, scale, translate;
Matrix4x4 transform = rotate * scale * translate;

// Order of matrix multiplication in GLSL
mat4 translate, scale, rotate;
mat4 transform = translate * scale * rotate;
```

Kuna *GLSL*'is puudub *type-cast* operaator, tuleb asendada teisendusel taolised operaatorid konstruktoritega:

```
// Type-cast in C#
float floatValue = 1.0f;
int intValue = (int)floatValue;

// Type-cast substitute in GLSL
float floatValue = 1.0f;
int intValue = int(floatValue);
```

Funktsioonide väljakutsete teisendamisega ilmneb juba rohkem probleeme, sest funktsioonid ei ole enamjaolt üks-ühele ülekantavad. Käesolev töö teisendab ainult piiratud hulga meetodeid ning iga välise meetodi väljakutse puhul visatakse rakenduses erand. Teisendatakse ainult *.NET* baasklassidest matemaatilised funktsioonid, mis leiduvad *System.Math* tüübi ning *System.Numerics* nimeruumis peituvate vektor- ja maatrikstüüpide küljes. Nende teisendus on otsekohene, sest enamus juhtudel leidub vaste *GLSL* sisseehitatud funktsiooni näol.

***.NET* meetod**	***GLSL* funktsioon**
double System.Math.Sin(double)	*float sin(float)*
int System.Math.Abs(int)	*int abs(int)*
float System.Numerics.Vector3.Dot(Vector3, Vector3)	*float dot(vec3, vec3)*

Tabel 3. Näited .NET meetodite teisendusest GLSL funktsioonideks

Esimese näite puhul on näha, et kummagi keskkonna sama operatsiooni signatuurid päris täpselt ei vasta. See nõuab realiseeritava teegipoolset lisatööd garanteerimaks, et *GLSL sin* funktsioon kutsutakse välja tõesti 32-bitise ujukoma arvulise argumendi korral.

.NET'ile leidub arvuliselt populaarseid kolmanda osapoole matemaatika valdkonna teeke, mis kahjuks käesolevas realisatsioonis ei ole toetatud. Üks võimalus oleks rakendada funktsiooniteisendusel *duck typing* lähenemist, kus teisendatav funktsioon ei määrata mitte konkreetse tüübi järgi, vaid tüübi nime järgi, rakendades peegeldust (*reflection*).

See töötaks paljudel juhtudel, sest näiteks ühes teegis realiseeritud skalaarkorrutise meetod nimega *Dot* kannaks teises teegis väga suure tõenäosusega sama nime. Küll aga tekib ka väga suur oht valesti tõlgendamisele.

Isegi kui baasklasside funktsioonid on teisendatavad, tekib küsimus, et kuidas lasta kasutajal suvalist funktsiooni (mitte ajada segamini *LINQ* avaldisega) kasutada. Näiteks puudub *GLSL*'is sisseehitatud funktsioon maatriksi inverteerimiseks. Kerge lahendus oleks lasta kasutajal lisada oma kaardistusi funktsioonide teisendustel ning *GLSL* koodi genereeritavasse arvutustuuma. See aga mängib otse vastu antud töö eesmärgile olla kasutaja jaoks võimalikult lihtne ning võib muuta teegi kasutamise keerulisemaks, kui oleks mõni madalatasemelisem lähenemine. Teine variant oleks lasta kasutajal oma soovitud funktsioon väljendada läbi *LINQ* päringuoperaatorite. Kohati võib olla taoline kood väga kohmakas ja kehvasti loetav (näiteks tsüklite väljendamine läbi päringuoperaatorite) või üldse realiseerimatu. Taolist keerukust ilmselt peeti silmas ka alampeatükis 2.2 kirjeldatud artiklis *LINQ* kasutamise kohta.

Suvaliste *C#* andmestruktuuride lisamine *GLSL* keskkonda on triviaalne, sest peegeldus pakub selleks piisavalt infot. *LINQ* avaldiste tõlkimisel hoitakse nimekirja lisatud andmestruktuuridest ning uue andmestruktuuri kohtamisel käiakse peegelduse üle struktuuri väljadel ning lisatakse analoogne struktuur keelde *GLSL*.

```
// Custom structure in C#
struct MyStruct
{
	public Vector2 Value1;
	public int Value2;
}

// Custom structure in GLSL
struct MyStruct
{
	vec2 Value1;
	int Value2;
};
```

Veidi lisatööd vajab anonüümsete klasside tõlkimine, sest kompilaatori poolt antud klassi ja tema väljade nimed sisaldavad *GLSL*'is lubamatuid sümboleid. Anonüümsed klassid on levinud võimalus *LINQ* avaldises väljendamaks vahetüüpe andmete edasikandmiseks päringuoperaatorite vahel või projektsiooni tulemusteks ilma, et peaks käsitsi uut tüüpi defineerima. Anonüümse klassi tuvastamine käib tõlkimisel peegeldusest saadud tüübi nime järgi, mille korral annab tõlkija vaba oleva spetsiaalse nime tüübile/väljale.

```
// Using an anonymous class in C# LINQ SELECT operator.
int[] input = { 1 };
var output = input.AsCompute().Select(x => new { x }).ToArray();

// Anonymous class translated to GLSL structure.
struct anonymous0
{
    int x;
};
```

Nii sisendandmete kui ka päringuoperaatorites viidatud väliste massiivide jaoks salvestatakse viide ning kirjeldatakse arvutustuumas *buffer* tüüpi muutuja, kuhu hiljem massiiv üle kantakse.

Käesolevas alampeatükis kirjeldati ära tõlkeprotsess *LINQ* päringust keelde *GLSL* ning ilmnenud eripärad. Seda protsessi võib pidada töö tähtsamaiks osaks.

4.4 SPIR-V mooduli käivitamine Vulkan'i kontekstis

Peale arvutustuuma tõlkimist on vaja nii arvutustuum kui ka sisendandmed kopeerida üle graafikaprotsessorile ligipääsetavasse mällu, arvutustuum käivitada ning arvutuste tulemused tagasi kopeerida rakenduse poolt ligipääsetavasse mällu. Selleks on vaja üles seadistada vajalik infrastruktuur.

Esmalt luuakse *Vulkan*'i instants (V*kInstance*). Juhul, kui rakendus on silumisrežiimis, lubatakse silumise lihtsustamiseks kiht *VK_LAYER_LUNARG_standard_validation*. Tegu on metataseme kihiga, mis grupeerib kokku hulga valideerimiskihte. Nende eesmärk on pookida rakendusse valideerimist lubavad koodilõigud, mille abil on võimalik tuvastada *Vulkan* liidese väär- või mitteoptimaalset kasutust. Vaikimisi puudub *Vulkan*'il enamus valideerimisloogikat, et tagada maksimaalne jõudlus. Lisaks lubatakse laiendus *VK_EXT_debug_report*, mis võimaldab rakendusel instantsile ette anda viite funktsioonile, mis kutsutakse välja, kui valideerimiskihis leitakse probleem. Sellise viite registreerimine käib läbi *vkCreateDebugReportCallbackEXT* käsu [5].

Teisalt valitakse füüsiline graafikaprotsessor, millel soovitakse arvutusi teostada. Käesolevas töös võetakse *vkEnumeratePhysicalDevices* käsu poolt tagastatud esimene seade, millel on tugi arvutusjärjekordadele (*compute queue*) ning ei arvestata muid parameetreid (nagu näiteks videomälu, järjekordade arv, vms). Esmane seade üldjuhul sobib, sest enamjaolt avalikustatakse diskreetne graafikakaart enne integreeritud

graafikakaarti. Samas ei ole see spetsifikatsiooni järgi garanteeritud, mis jätab ruumi edasiarenduseks sobiva graafikaprotsessori valikul.

Kolmandaks luuakse füüsilise seadme põhjal loogiline seade läbi *vkCreateDevice* käsu, kus piiratakse ära, millist füüsilise seadme funktsionaalsust hakatakse kasutama. Küsitakse kasutusluba kõikidele arvutusjärjekordadele, et teegi mitmelõimelise kasutuskorra puhul oleks võimalik ära kasutada seadmel asünkroonset arvutamist (*async compute*).

Eelmainitud infrastruktuur on skoobilt globaalne ehk kõikide päringute puhul kasutatakse samu objekte. Edasine *Vulkan*'i objektide käsitlus on skoobilt piiratud iga konkreetse päringuga. Päringujärgset tegevust vaadeldakse neljas etapis: seadistus, andmete kirjutamine, arvutustuuma käivitus, andmete lugemine. Taolist segmenteerimist kasutatakse ka peatükis 5.1, kus jõudluse analüüsimisel on kasulik eristada igal etapil kulunud aega.

Päringu käivitamisel seadistatakse esmalt arvutustuuma poolt kasutatavad andmepuhvrid läbi *vkCreateBuffer* käsu. Kõik loodavad puhvrid on *storage buffer* tüüpi. Olenevalt kas tegu on puhvriga, kuhu rakendus kirjutab või kust rakendus loeb, määratakse puhvrile erinevat tüüpi mäluosa. Mälu, kuhu rakendus kirjutab, võiks olla *device local* ja *host visible* (mälu tüübid on kirjeldatud alampeatükis 0). See on ideaalne juhuks, kus *CPU* saab kirjutada otse *GPU* mällu, mida *GPU* saab lugeda ilma *PCIe* siini kasutamata. Juhul, kui seade ei toeta sellise kombinatsiooniga mälu, taganetakse lihtsalt *host visible* tüüpi mälule. Mälu, kust rakendus loeb, võiks olla *host visible*, *host coherent* ja *host cached* omaduste kombinatsiooniga. See on ainuke mälutüüp, mis lubab vahemälu põhist lugemist (*cached read*). [44]

Järgnevalt luuakse sünkroniseerimisprimitiivina tara (*fence*), arvutustuuma poolt vajalikud deskriptorid (*descriptor*), arvutuskonveier (*compute pipeline*) ning salvestatakse käsupuhvrisse (*command buffer*) arvutustuuma käivitamiseks vajalikud käsud.

Tara loomine käib läbi *vkCreateFence* käsu. Tara on vajalik, et arvutustuuma käivitamisel oodata tara taga, kuni on teada, et arvutus on lõppenud.

Deskriptorid kirjeldavad ära arvutuskonveieri poolt kasutatavad ressursid. Käesoleva töö päringute puhul on iga sisend- ja väljundpuhver kirjeldatud deskriptoriga, mis määrab ära, et puhvrit kasutatakse arvutustuumas ning et tegu on *storage buffer* tüüpi puhvriga. Deskriptorite paigutus määratakse ära läbi *vkCreateDescriptorSetLayout* käsu ja deskriptorid luuakse läbi *vkCreateDescriptorPool* ja *vkAllocateDescriptorSets* käskude. Igale deskriptorile seatakse vastavusse andmepuhver läbi *vkUpdateDescriptorSets* käsu.

Arvutuskonveier kapseldab kogu arvutuse teostamiseks vajaliku seisundi graafikaprotsessoril. Konveieri paigutus luuakse läbi *vkCreatePipelineLayout* käsu ning konveier ise läbi *vkCreateComputePipeline* käsu, kus argumentidena antakse ette vastavalt deskriptorite paigutus ja arvutustuum.

Viimaks salvestatakse arvutustuuma käivitamiseks vajalikud graafikaprotsessori käsud:

```
int groupCountX = (int)Math.Ceiling(input.Length / 256.0);

cmdBuffer.Begin(
    new CommandBufferBeginInfo(CommandBufferUsages.OneTimeSubmit));
cmdBuffer.CmdBindPipeline(PipelineBindPoint.Compute, pipeline);
cmdBuffer.CmdBindDescriptorSet(
    PipelineBindPoint.Compute, pipelineLayout, descriptorSet);
cmdBuffer.CmdDispatch(groupCountX, 1, 1);
cmdBuffer.End();
```

Kuna sisendandmeid vaadeldakse 1-mõõtmelise jadana, määratakse *vkCmdDispatch* käsu puhul töörühmade arvuks dimensioonides y ja z üks (töörühmad on kirjeldatud alampeatükis 3.3). X-dimensioonis valitakse suurus sisendi pikkuse *input.Length* järgi. Lokaalse töörühma pikkuseks on määratud 256, sest see on toetatud kõikide *Vulkan*'iga töötavate seadmete puhul [5]. Võimalik edasiarendus valiks selle suuruse konkreetse seadme näitajate põhjal. Jagades sisendi pikkus lokaalse töörühma suurusega ning võttes sellest täisarvuline lagi, saadakse arvutuseks vajalike töörühmade arv.

Andmete kopeerimine igasse sisendpuhvrisse toimib järgneva koodilõigu alusel:

```
void* dstPtr = deviceMemory.Map(0, hostMemorySize);
GCHandle hostMemoryHandle = GCHandle.Alloc(hostMemory, GCHandleType.Pinned);
void* srcPtr = hostMemoryHandle.AddrOfPinnedObject();

Buffer.MemoryCopy(srcPtr, dstPtr, deviceMemorySize, hostMemorySize);

hostMemoryHandle.Free();
deviceMemory.Unmap();
```

Esmalt kaardistatakse seadme mälu läbi *vkMapMemory* käsu, millega saadakse sinna viide. Kuna rakenduse puhul on tegu hallatud keskkonnaga, tuleb otse viitelt viitele mälu kopeerimise puhul tagada, et keskkonna mäluhaldur hallatud mälupiirkonda ei liigutaks ega muudaks. Selleks on vajalik *GCHandle.Alloc* väljakutse, mis keelab mäluhalduril mälupiirkonnaga opereerimise. Kopeerimine teostatakse läbi *Buffer.MemoryCopy* väljakutse. *hostMemoryHandle.Free* annab mäluhaldurile tagasi loa opereerida mälupiirkonnaga. *deviceMemory.Unmap* annab *Vulkan*'ile teada, et rakendus on lõpetanud seadme mälupiirkonna kasutamise.

Andmete kopeerimine seadme väljundpuhvrist tagasi rakenduse mällu toimib täpselt samamoodi nagu eelnevas lõigus kirjeldatud kopeerimise korral. *Buffer.MemoryCopy* argumendid antakse lihtsalt teises järjekorras ette.

Lõpuks, eelnevalt salvestatud käsupuhver tuleb käskude täitmiseks saata seadme arvutusjärjekorda. Rakendus ootab tara taga, kuni käsud on seadmel täidetud.

```
queue.Submit(new SubmitInfo(commandBuffers: new[] { cmdBuffer }), fence);
fence.Wait();
```

Kui eelneva alampeatüki järel on olemas protsess, kuidas algoritm tõlkida graafikaprotsessorile arusaadavale kujule, siis käesolevas alampeatükis vaadeldi, kuidas tõlgitud programm ja sisendandmed *GPU*'le kättesaadavaks teha, käivitada ja tulemused sealt tagasi lugeda.

5 Analüüs

5.1 Jõudluse võrdlus CPU'l vs GPU'l paralleliseeritud ülesande korral

Jõudluse võrdluseks on aluseks võetud algoritm kiire *Fourier'* teisendus (*FFT* ehk *Fast Fourier Transform*) [45].

Fourier' teisendus on signaalitöötlusest tuntud algoritm, mis teisendab funktsiooni selle sagedusspektrit iseloomustavaks funktsiooniks. Diskreetne *Fourier*' teisendus on pideva *Fourier*' teisenduse vaste ajas ja nivoos diskreeditud funktsioonide ja signaalide jaoks. Signaalitöötluses on signaal funktsioon, millest võetakse proovid üle piiratud ajavahemiku (näiteks raadiosignaal või temperatuurinäidud). Teisendus muudab võrdsete sammudega proovide jada sagedusdomeeni, mis näitab ära signaalis esinevad sagedused.

Diskreetse *Fourier*' teisenduse asümptootiline keerukus on $O(n^2)$. Selle keerukuse vähendamiseks on leitud optimeerimisvõimalus, mis toob keerukuse $O(n \log n)$ gruppi. See optimeeritud variant kannab nime kiire *Fourier*' teisendus. Algoritmi realisatsiooni keeles *C#* on analoogne alampeatükis 2.3 kirjeldatud allikas välja toodud realisatsioonile [14].

Esmalt vaadeldakse kogu ülesande arvutusaega kolme erineva meetodiga: *LINQ* [13], *PLINQ* [19] ning käesoleva töö graafikaprotsessori põhise realisatsiooniga. Seejärel uuritakse viimase meetodi korral kulunud aega detailsemalt, kus eristatakse *LINQ* avaldise tõlkimisele, graafikaseadme seadistamisele, andmete kirjutamisele videomällu, graafikaprotsessoril käivitatud ülesandele ning andmete lugemisele videomälust kulunud aega. Kõiki kolme meetodi tulemused valideeritakse võrreldes nende omavahelist võrdväärsust.

Enne *Fourier*' teisenduse koodilõigu näitamist, tuuakse veidi lihtsam näide, kus on selgelt eristatud liidese kasutuse erinevus *CPU* ühelõimelise, *CPU* mitmelõimelise ja *GPU* põhiste realisatsioonide vahel. Antud näites luuakse 200 miljonit täisarvu ning suurendatakse iga arvu ühe võrra.

```
int[] input = Enumerable.Range(0, 200_000_000).ToArray();
```

```
input              .Select(x => x + 1).ToArray();
input.AsParallel().Select(x => x + 1).ToArray();
input.AsCompute()  .Select(x => x + 1).ToArray();
```
Sarnaselt käivitatakse ka *Fourier*' teisendus kolme erineva viisiga.

Signaali sisend näite jaoks genereeritakse pseudo-juhuarvude generaatori abil, kus iga signaali proov on ujukomaarv ühtlase jaotusega vahemikus 0-1. Päringuoperaatoreid ei rakendata mitte signaali sisendile, vaid abijadale, läbi mille muutujate indekseeritakse sisendisse. Tegu on iteratiivse, mitte rekursiivse realisatsiooniga.

```
int fftSize = 2;
int[] xSequence = Generate(size, i => i);
Vector2[] input = Generate(size, _ => new Vector2(random.Next(), 0.0f));

int numIterations = Log(size, 2.0);
for (int i = 0; i < numIterations; i++)
{
    output = (from x in xSequence
        let angle = -2 * PI * (x / fftSize)
        let t = new Vector2(Cos(angle), Sin(angle))
        let x0 = Floor(x / fftSize) * (fftSize / 2) + x % (fftSize / 2)
        let x1 = x0 + size / 2
        let val0 = input[x0]
        let val1 = input[x1]
        select new Vector2(
            val0.X + t.X * val1.X - t.Y * val1.Y,
            val0.Y + t.Y * val1.X - t.X * val1.Y))
        .ToArray();

    fftSize *= 2;

    Swap(ref input, ref output);
}
```

Sama päringut käivitatakse mitme *CPU* protsessori peal asendades *xSequence* avaldise avaldisega *xSequence.AsParallel()* ning *GPU* protsessori peal asendades avaldisega *xSequence.AsCompute()*.

Väiksema vea tagamiseks mõõtmistulemustes rakendatakse paari koodipõhist nippi [46]:

- Määratakse testi operatsiooniprotsessi prioriteediklass kõrgeks: `Process.GetCurrentProcess().PriorityClass = ProcessPriorityClass.High;`
- Iga testalgoritm käivitatakse enne mõõtmist minimaalse sisendandmestiku korral, et tagada, et *just-in-time* kompilaator on kompileerinud vahekeele *msil* masinkeeleks.

- Enne iga mõõtmist käivitatakse manuaalselt prügikoristus ning oodatakse, et see oleks oma töö lõpetanud:

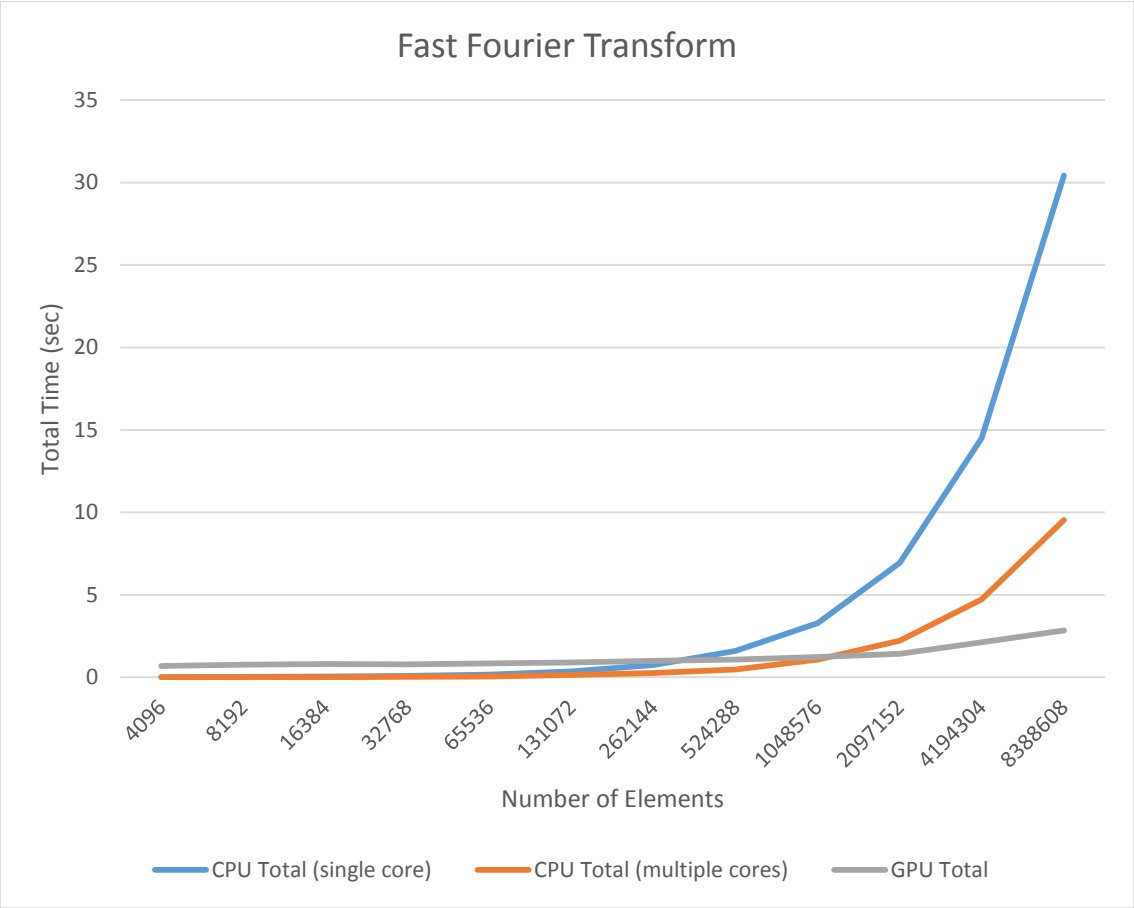
```
GC.Collect();
GC.WaitForPendingFinalizers();
```

Mõõtmised on teostatud järgneva riist- ja tarkvaralise konfiguratsiooniga:

Operatsioonisüsteem	*Windows 10 Professional x64*
Protsessi režiim	*x86*
.*NET* versioon	*.NET Framework 4.6.1*
Protsessor	*Intel Core i7-2600K 3.40GHz (4 physical * 2 hardware threads)*
Muutmälu	*Kingston 16GB DDR3 1333MHz*
Graafikaprotsessor	*NVIDIA GeForce GTX970 1050MHz (1664 CUDA cores)*

Tabel 4. Konfiguratsioon Fourier' teisenduse mõõtmisel

Järgnevalt on esitatud mõõtmise tulemused nii graafiliselt kui ka tabelkujul. Graafiline kujutus annab hea ülevaate jõudluse skaleeruvusest andmemahtude kasvamise korral. Tabelkujul on täpselt ära näha graafikaprotsessori rakendamise lisakulu väiksemate andmemahtude korral ning proovid, mis hetkest hakkab mitmelõimeline *CPU* ja mis hetkest *GPU* variant ennast ära tasuma.

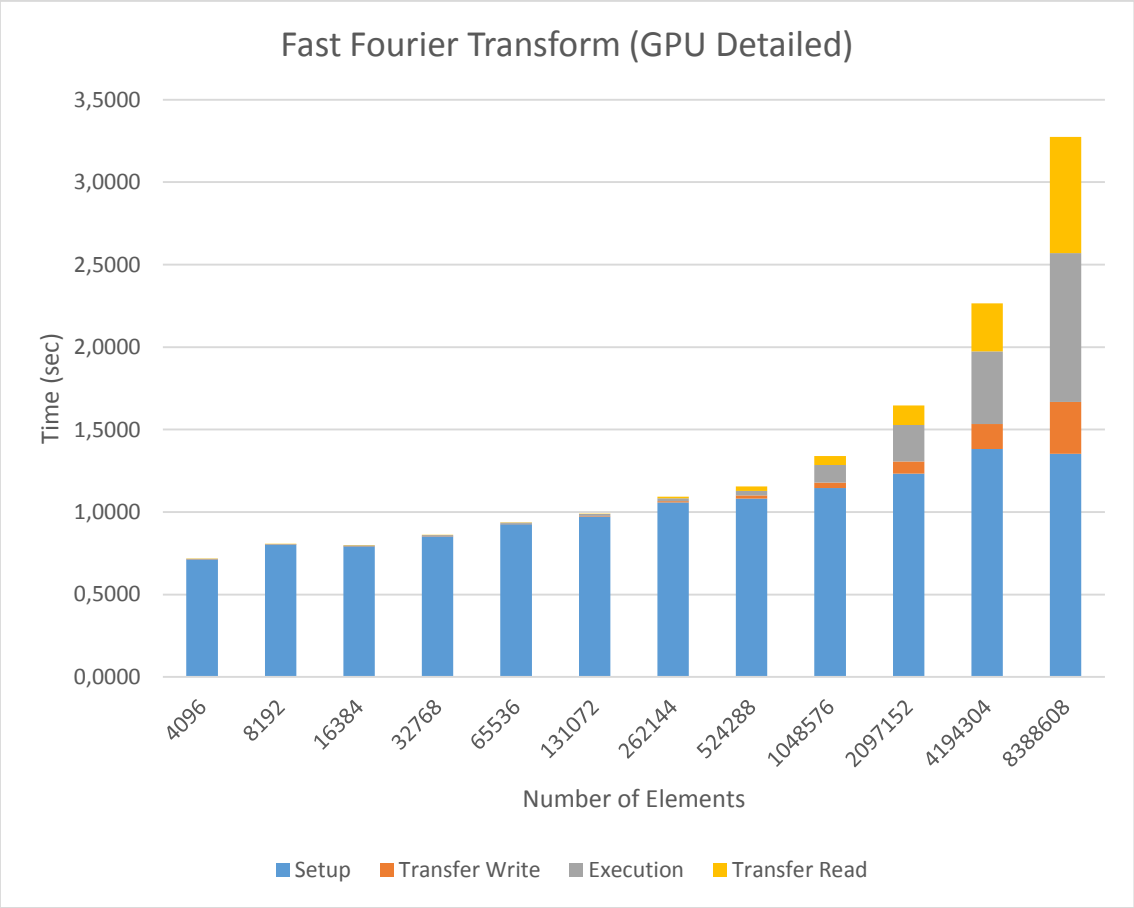


Joonis 17. Kiire Fourier' teisenduse jõudluse võrdlus erinevate LINQ-põhiste meetodite vahel

Elementide arv	***CPU* (üks tuum) kulunud aeg (sek)**	***CPU* (mitu tuuma) kulunud aeg (sek)**	***GPU* kulunud aeg (sek)**
4096	0,0088	0,0038	0,6897
8192	0,0215	0,0077	0,7609
16384	0,0395	0,0147	0,8093
32768	0,0781	0,0252	0,7946
65536	0,1657	0,0545	0,8424
131072	0,3603	0,1362	0,8954
262144	0,7511	0,2556	0,9979
524288	1,5987	0,4813	1,0784
1048576	3,2972	1,0736	1,2368
2097152	6,9371	2,2210	1,4306
4194304	14,5069	4,7191	2,1289
8388608	30,4193	9,5437	2,8427

Tabel 5. Kiire Fourier' teisenduse jõudluse võrdlus erinevate LINQ-põhiste meetodite vahel

Graafiku järgi vastavad mõõtmistulemused ootustele, kus *CPU* põhine lähenemine on väiksemate andmemahtude korral *GPU* põhisest lähenemisest tunduvalt tõhusam. Kõikide andmemahtude juures on protsessori mitme tuuma kasutamine tõhusam ühe-tuumalisest variandist. Väiksemate andmemahtude juures hakkaks lõimede haldamisele kuluv aeg üle kaaluma saadavat ajavõitu. Alles ~2 miljoni elemendi juures hakkab graafikaprotsessori kasutamine ennast ära tasuma. Edasi on detailsemalt välja toodud samade andmehulkade juures, mis etappidele kulub viimases variandis kõige rohkem aega:



Joonis 18. Kiire Fourier' teisenduse graafikaprotsessori meetodile kulunud aeg detailsemalt

Elementide arv	**Seadistamise aeg (sek)**	**Andmete kopeerimisel videomällu kulunud aeg (sek)**	**Arvutustele kulunud aeg (sek)**	**Andmete kopeerimisel videomälust kulunud aeg (sek)**
4096	0,7134	0,0002	0,0025	0,0001
8192	0,8018	0,0003	0,0045	0,0002
16384	0,7929	0,0005	0,0052	0,0003

32768	0,8531	0,0009	0,0065	0,0007
65536	0,9265	0,0017	0,0087	0,0019
131072	0,9730	0,0037	0,0112	0,0033
262144	1,0586	0,0083	0,0169	0,0089
524288	1,0823	0,0178	0,0285	0,0268
1048576	1,1448	0,0344	0,1061	0,0549
2097152	1,2335	0,0735	0,2209	0,1193
4194304	1,3820	0,1517	0,4411	0,2896
8388608	1,3532	0,3151	0,9025	0,7036

Tabel 6. Kiire Fourier' teisenduse graafikaprotsessori meetodile kulunud aeg detailsemalt tabelkujul

Seadistamise aeg sisaldab endas nii *LINQ* avaldiste tõlkimist keelde *GLSL* kui ka sealt edasi *SPIR-V* mooduliks. Lisaks läheb selle aja alla tööks vajalike *Vulkan*'i objektide initsialiseerimine. Viimane on arvestatud seadistuse aega sisse, kuna oli märkimisväärselt väike osa võrreldes avaldiste tõlkimisega ning kasvab sarnase logaritmilise kurvi alusel. Seega võib seadistuse all ette kujutada ainult tõlkimisaega.

Seadistuse aeg on varieeruv ja mitte püsiv, sest tegu on iteratiivse algoritmiga, kus vastavalt suuremate andmehulkade korral teostatakse ka rohkem iteratsioone ehk rohkem kordi on vaja läbida seadistusetappi. Kuna iteratsioonide vahel avaldised ei muutu, vaid ainult sisendandmestik, siis teoorias oleks võimalik mingi kriteeriumi põhjal tõlgitud arvutustuum hoiustada vahemällu ning taaskasutada seda. Kuidas seda kõige mõistlikum teostada oleks, jääb võimalikuks edasiarenduseks.

Rakenduse poolt andmete kirjutamine videomällu on kiirem kui sealt lugemine, sest tegu väga levinud kasutusjuhuga ning seetõttu ka agressiivsemalt optimeeritud [47].

5.2 Autoripoolne hinnang

Olles realiseerinud alampeatükis 2.1 kirjeldatud magistritöös välja pakutud lahenduse, võib väita, et tegu on praktiliselt töötava meetodiga teostada arvutusi graafikakaardil.

Käesoleva töö osaline realisatsioon ei ole väga keeruline just seetõttu, et *LINQ* päringute tõlkimine toimub keelde *GLSL*, mille süntaktiline struktuur on sarnane keele *C#* struktuurile, lubades mitmel pool avaldiste üks-ühele teisenduse ilma keerukat transformatsiooni läbides. Kui optimeerida realisatsiooni otse tõlkima *LINQ* avaldisi

keelde *SPIR-V*, kasvab lähtekoodi keerukus tunduvalt, sest nimetatud binaarkeele struktuur erineb märkimisväärselt ning nõuab arendajapoolset tähelepanu palju madalama taseme aspektidele [21].

Üheks suureks probleemiks on asjaolu, et *LINQ* liidesele omane deklaratiivne väljendusviis ei pruugi olla alati arendajale intuitiivsem kui imperatiivne lähenemine. See tähendab, et algoritmi kirjeldus on raskemini mõistetav ning läheb tugevalt vastuollu käesoleva teegi põhieesmärgiga pakkuda lihtsat programmeerimisliidest. Heaks näiteks on 2-mõõtmeline massiiv. Kahjuks *LINQ* ei paku päringuoperaatoreid 2- või rohkema mõõtmeliste andmehulkade jaoks, mis tähendab, et sisendandmeid tuleb kujutada läbi 1-mõõtmelise vormi. Selline andmestruktuur sobib küll funktsionaalselt aga semantiliselt ei tundu õige. Kui taolisel andmehulgal soovida leida iga andmeelemendi jaoks tema naaberelementidega keskmine väärtus, võib *LINQ*-põhine realisatsioon kohmakas tunduda:

```
// For each element, find an average value for it and its' neighbors.
// NB! Does not take out of bounds indices into account for simplicity.

const int size = 256;

// Regular imperative approach.

var input = new int[size, size];
// Initialize the input matrix with some data.

var output = new int[size, size];
for (int x = 0; x < size; x++)
for (int y = 0; y < size; y++)
{
    for (int i = -1; i <= 1; i++)
    for (int j = -1; j <= 1; j++)
    {
        output[x, y] += input[x + i, y + j];
    }
    output[x, y] /= 9;
}


// LINQ-based approach.

var input = new int[size * size];
// Initialize the input matrix with some data.

int[] output = Enumerable.Range(0, input.Length).Select(i => (
    input[i-size-1] + input[i-size] + input[i-size+1] +
    input[i-1]      + input[i]      + input[i+1] +
    input[i+size+1] + input[i+size] + input[i+size+1]
) / 9).ToArray();
```

Kuigi koodi mõttes tundub *LINQ*-põhine lähenemine kompaktne, siis probleemiks on skaleeruvus, kus 5x5 suuruse tuuma puhul läheb käsitsi liitmine tülikaks ning tuleks välja mõelda keerukam ja skaleeruvam lahendus, mille puhul loetavus väheneks drastiliselt. Ka alampeatükis 5.1 välja toodud *Fourier'* teisenduse algoritm oleks läbi klassikaliste tsüklite arusaadavam.

Pilditöötlus on väga levinud operatsioon, kus kasutatakse riistvara kiirendust. Piltide käsitlemiseks pakuvad nii *GLSL* kui ka *SPIR-V* spetsialiseeritud funktsionaalsust ja andmetüüpe. Näiteks, läbi *LINQ* liidese ei ole elegantset viisi, kuidas defineerida pildist proovi võtmise meetodit (*sampling*). Olgu selleks siis *clamping*, *wrapping* või *mirroring*. Et lubada kasutajal sellist infot edastada, tuleb laiendada teegi avalikku liidest uute tüüpidega ning kannatab taaskord lihtsus. Seega, taoline teek ei sobi pilditöötluseks või nõuaks äärmiselt läbimõeldud liidest, et oleks tasuv võrreldes madalama taseme alternatiiviga.

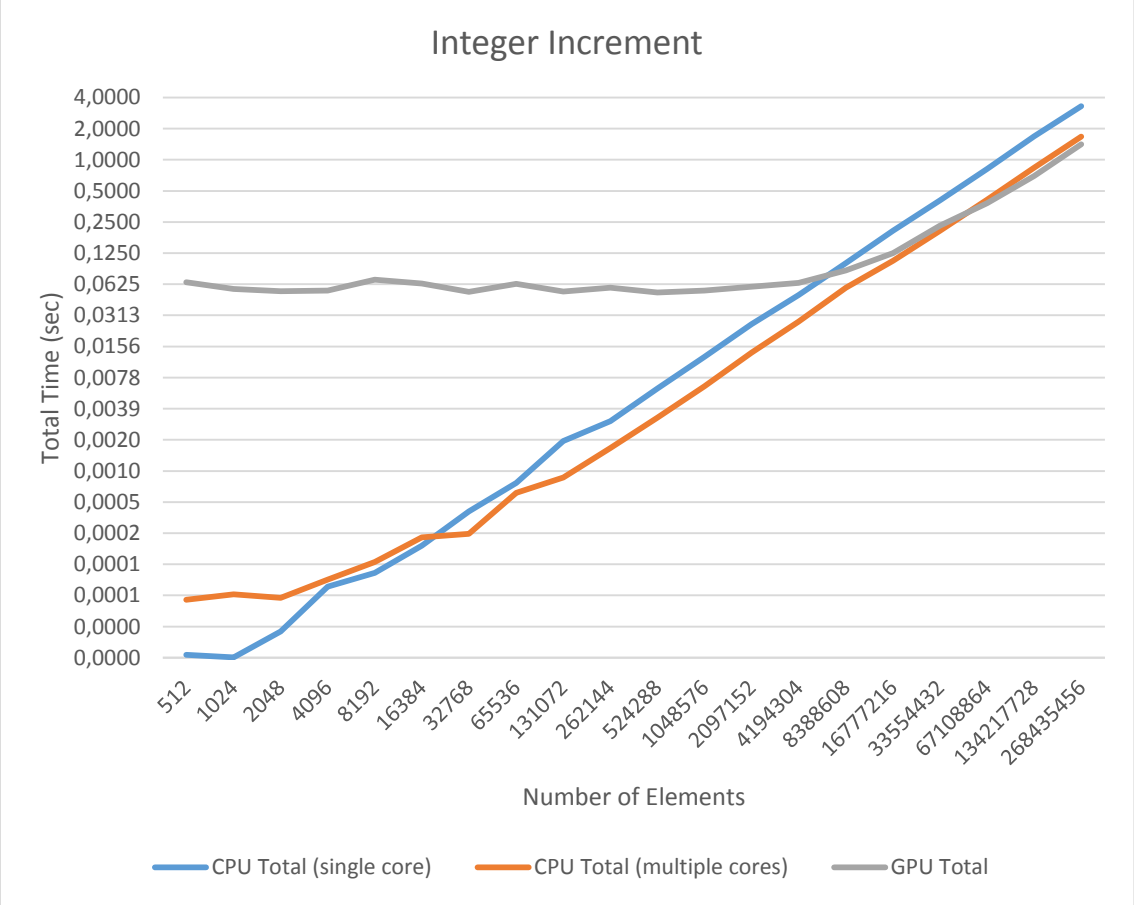
Kasutaja või kolmanda osapoolte funktsioone on keeruline väljendada. Puudub automaatne mehhanism, mis oskaks suvalist funktsiooni teisendada arvutustuuma keelde. Mõneti võiks leevendada probleemi variant, kui kasutada toetatud meetodite tuvastamiseks *duck typing*'ut, mis lubaks vähemalt kasutada kolmanda osapoole matemaatikateeke, kus operatsioonide nimed kattuvad *.NET* baasteegis toetatud operatsioonide nimedega. Lahenduseks oleks laiendada teegi avalikku liidest, et lubada kasutajal lisada loodavasse arvutustuuma mistahes funktsioone otse läbi *GLSL* süntaksi. Madalama taseme alternatiiv antud teegile tundub mõistlikum.

Kokkuvõtteks järeldab autor oma realisatsioonist saadud kogemuste põhjal, et käesolev meetod arvutuste kiirendamiseks oleks sobilik ainult väikese hulga probleemide lahendamiseks, kusjuures sinna hulka ei kuulu pilditöötlus. Lisaks sobib see viis pigem prototüüpimiseks, et aimu saada potentsiaalsest jõudluse kasust graafikaprotsessoril kiirendamisel. Tootekoodi puhul soovitab autor leida ressursid, et panustada arendaja koolitusse, et selgeks teha mõni madalama taseme teek ja printsiibid. Seda just fikseeritud jõudluse lisakulu tõttu, mis on tingitud *LINQ* avaldiste tõlkimisest *SPIR-V* mooduliks.

5.3 Võimalikud edasiarendused

Võimalusi edasiarendusteks on mitmeid ning need tuuakse käesolevas alampeatükis välja. Teemasid käsitletakse pinnapealselt ehk ühegi teema kohta ei ole tehtud põhjalikku taustauuringut.

Nagu oli näha alampeatükist 5.1, oleneb optimaalse arvutusmeetodi valik suuresti andmehulgast. Väikese andmehulga puhul on mõistlik teostada arvutusi ühe *CPU* tuuma peal, suurema andmehulga korral juba rakendada mitut *CPU* tuuma ning veel suuremate korral viia ülesande täitmine üldse *GPU* peale. Huvitav oleks aga analüüsida, mis näidikute põhjal ja kuidas leida algoritmi korral need andmehulga suurused, kus üleminek ühe tuuma pealt mitme peale või *CPU* pealt *GPU* peale hakkab ennast ajaliselt ära tasuma nagu on näha järgneva joonise joonte ristumiskohtade pealt.



Joonis 19. Jõudluse eelise üleminekud erinevate meetodite vahel

Kui leida taoline meetod, oleks võimalik liidest kasutaja jaoks riistvara suhtes veelgi abstraktsemaks muuta. Taoline lihtsustuse tagaks olukorra, kus kasutaja ei pea enam mõtlema, mil moel tema arvutusi teostatakse.

Maksimeerimaks võimalikku jõudlust, oleks teoorias võimalik kasutada kogu arvuti poolt kasutatavat riistvara arvutustöö täitmiseks. See kujutaks endast ette paralleeltöötlust nii *CPU* kui ka kõikide masinas leiduvate *GPU*'de peal. Põhiprobleemiks jääb küsimus, mis andmehulkade juurest hakkab selline töö jaotus ennast ära tasuma ning kuidas tööd kõige optimaalsemalt jaotada – iga protsessor on erineva võimekusega.

Eelnevas alampeatükis kirjeldatud probleemi lahendamiseks, et kasutajal oleks võimalik suvalist funktsiooni kasutada *LINQ* avaldises, tuleks laiendada teegi avalikku liidest ja pakkuda võimalus kasutajal arvutustuuma laiendada.

GPU-põhine käivitus on hetkel *CPU* mõistes sünkroonne. Peale käivituskäsu *vkQueueSubmit* saatmist graafikaprotsessorile *CPU* ootab, kuni töö on lõppenud läbi *vkWaitForFences* käsu. See ootamine raiskab väärtuslikku *CPU* aega. Kahjuks ei paku *Vulkan* asünkroonset liidest selle probleemi lahendamiseks. Lahenduse saaks ise ehitada läbi käsu *vkGetFenceStatus*, mille kaudu on võimalik pärida, kas arvutus on lõppenud. Välja võiks see näha perioodilise pärimisena (*polling*), kus päringute vahel - juhul kui päring annab eitava vastuse – saab protsessor töötada teiste ülesannete kallal.

Üheks optimeeringuks oleks kaotada vajadus kasutada tööriista *glslangValidator*, mis tähendaks *LINQ* avaldiste puu tõlkimist otse binaarkeelde *SPIR-V*. Selle lähenemise miinuseks on raskem silumine ja väljundi mõistmine, kuid ideaalis võimaldaks teek mõlemat tõlkimisvarianti.

6 Kokkuvõte

Käesoleva töö eesmärgiks oli luua realisatsioon, mis lihtsustaks riistvara kiirenduse kasutust suuremahulisteks arvutusteks läbi tuttava kõrgetasemelise programmeerimisliidese. Madalatasemeliste teekide kasutamise probleemiks on kõrge õppimisele ja utiliseerimisele minev ajaline kulu. Teemakohased materjalid olid välja pakkunud idee ehitada *.NET LINQ* liidese taha kompilaator, mis tõlgib standardsed päringuoperaatorid graafikaprotsessorile arusaadavaks koodiks. See võimaldaks kasutajale läbinähtavalt kiirendada suuremahulisi arvutustöid käivitades neid *GPU*'l. Antud töö on loogiline jätk teoreetilistele alustele, kinnitades seal püstitatud väiteid, tuues välja võimaliku realisatsiooni kirjelduse ning autoripoolse hinnangu.

Töö olulisemaks tulemuseks on selle raames valminud kaks teeki, mis on avalikult kättesaadavad *GitHub* keskkonnas ja omavad paindlikku *MIT* litsentsi. Esimene neist on üldotstarbelisem teek, läbi mille on võimalik *.NET* keskkonnas programmeerida *Vulkan* graafika- ja arvutusteegi vastu. Teiseks on töö teemat käsitlenud prototüüp, mille põhiülesandeks on orkestreerida andmeid ja andmete peal rakendatavaid algoritme protsessori ja graafikaprotsessori vahel. Lisaks analüüsiti jõudlust signaalitöötluses levinud kiire *Fourier*' teisenduse põhjal, mis annab head aimu meetodi skaleeruvusest suuremate andmemahtude korral. Samas toob see ka välja suure püsikulu, mis muudab meetodi kasutuse väiksemate andmemahtude korral kõlbmatuks.

Kuigi päringute tõlketeegi toodangukõlbuliku versioonini jõudmiseks tuleks veel palju tööd teha ja meetodi kasulikkus on üleüldse kaheldav, siis kirjeldatud töö peaks olema heaks pidepunktiks huvilistele, kes analoogset probleemi soovivad lahendada. Käsitletud meetodi suurimaks puuduseks on *LINQ* päringuoperaatorite piiratus ning kohati ebaintuitiivne süntaks vajalik tavapäraste algoritmi konstruktsioonide kirjeldamiseks. See läheb otseselt vastuollu seatud kvaliteedile olla kasutaja jaoks võimalikult lihtne. Küll aga jääb ruumi veel paljudeks huvitavateks edasiarendusteks / alternatiivideks.

Kasutatud kirjandus

[1] Intel, „Intel Core i7-7700 Processor Specifications,“ Intel, [Võrgumaterjal]. Kättesaadav: https://ark.intel.com/products/97128/Intel-Core-i7-7700-Processor-8M-Cache-up-to-4_20-GHz. [Kasutatud 26 2 2017].

[2] NVIDIA, „GeForce GTX 1080 Graphics Card,“ NVIDIA, [Võrgumaterjal]. Kättesaadav: https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080/. [Kasutatud 26 2 2017].

[3] NVIDIA, „Parallel Programming and Computing Platform | CUDA | NVIDIA,“ NVIDIA, [Võrgumaterjal]. Kättesaadav: http://www.nvidia.com/object/cuda_home_new.html. [Kasutatud 26 2 2017].

[4] Khronos Group, „OpenCL - The open standard for parallel programming of heterogeneous systems,“ [Võrgumaterjal]. Kättesaadav: https://www.khronos.org/opencl/. [Kasutatud 26 2 2017].

[5] Khronos Group, „Vulkan 1.0.48 - A Specification,“ Veebruar 2017. [Võrgumaterjal]. Kättesaadav: https://www.khronos.org/registry/vulkan/specs/1.0/xhtml/vkspec.html. [Kasutatud 30 4 2017].

[6] Microsoft Corporation, „Direct3D,“ [Võrgumaterjal]. Kättesaadav: https://msdn.microsoft.com/en-us/library/windows/desktop/hh309466(v=vs.85).aspx. [Kasutatud 26 2 2017].

[7] Apple Inc., „Metal for Developers - Apple Developer,“ [Võrgumaterjal]. Kättesaadav: https://developer.apple.com/metal/. [Kasutatud 26 2 2017].

[8] L. Bläser, D. Egloff, O. Knobel, P. Kramer, X. Zhang ja D. Fabian, „Alea Reactive Dataflow: GPU Parallelization Made Simple,“ 2014. [Võrgumaterjal]. Kättesaadav: http://www.concurrency.ch/Content/publications/Blaeser_etal_Alea_Reactive_Dataflow_REBLS_2014.pdf. [Kasutatud 26 2 2017].

[9] C. Rossbach, J. Currey, M. Silberstein, B. Ray ja E. Witchel, „PTask: Operating System Abstractions To Manage GPUs,“ 23 10 2011. [Võrgumaterjal]. Kättesaadav: https://www.cs.utexas.edu/users/witchel/pubs/sosp11rossbach-ptask.pdf. [Kasutatud 26 2 2017].

[10] Oracle Corporation, „java.com: Java + You,“ [Võrgumaterjal]. Kättesaadav: https://www.java.com/en/. [Kasutatud 26 2 2017].

[11] Microsoft Corporation, „About .NET,“ 10 2016. [Võrgumaterjal]. Kättesaadav: https://docs.microsoft.com/en-us/dotnet/articles/standard/. [Kasutatud 26 2 2017].

[12] P. Agrawal, „Parallelizing LINQ Program for GPGPU,“ 6 2012. [Võrgumaterjal]. Kättesaadav: https://pdfs.semanticscholar.org/ae51/f0cf614d63134c290d3d3585423c2ab09aac.pdf. [Kasutatud 26 2 2017].

[13] Microsoft Corporation, „LINQ: .NET Language-Integrated Query,“ Veebruar 2007. [Võrgumaterjal]. Kättesaadav: https://msdn.microsoft.com/en-us/library/bb308959.aspx. [Kasutatud 26 2 2017].

[14] N. Palladinos, „GpuLinq: Democratizing GPGPU programming through OpenCL and LINQ,“ 6 4 2015. [Võrgumaterjal]. Kättesaadav: https://github.com/nessos/GpuLinq. [Kasutatud 26 2 2017].

[15] P. Eeles, „Capturing Architectural Requirements,“ 11 2005. [Võrgumaterjal]. Kättesaadav: https://www.ibm.com/developerworks/rational/library/4706.html#N100A7. [Kasutatud 7 4 2017].

[16] L. W. Paul Vick, „The Microsoft Visual Basic Language Specification 11.0,“ 2017. [Võrgumaterjal]. Kättesaadav: https://msdn.microsoft.com/en-us/library/ms234437.aspx. [Kasutatud 26 2 2017].

[17] I. Landwerth, „Introducing .NET Standard,“ 26 9 2016. [Võrgumaterjal]. Kättesaadav: https://blogs.msdn.microsoft.com/dotnet/2016/09/26/introducing-net-standard/. [Kasutatud 7 4 2017].

[18] H. Vallaste, „e-Teatmik: IT ja sidetehnika seletav sõnaraamat,“ 2017. [Võrgumaterjal]. Kättesaadav: http://www.vallaste.ee/. [Kasutatud 30 4 2017].

[19] Microsoft Corporation, „Parallel LINQ (PLINQ),“ [Võrgumaterjal]. Kättesaadav: https://msdn.microsoft.com/en-us/library/dd460688%28v=vs.110%29.aspx?f=255&MSPPError=-2147217396. [Kasutatud 26 3 2017].

[20] D. Box ja A. Hejlsberg, „LINQ: .NET Language-Integrated Query,“ Microsoft Corporation, 2 2007. [Võrgumaterjal]. Kättesaadav: https://msdn.microsoft.com/en-us/library/bb308959.aspx. [Kasutatud 17 3 2017].

[21] Khronos Group, „SPIR-V Specification,“ 26 2 2017. [Võrgumaterjal]. Kättesaadav: https://www.khronos.org/registry/spir-v/specs/1.0/SPIRV.pdf. [Kasutatud 18 3 2017].

[22] Khronos Group, „OpenGL Reference Compiler,“ [Võrgumaterjal]. Kättesaadav: https://www.khronos.org/opengles/sdk/tools/Reference-Compiler/. [Kasutatud 19 3 2017].

[23] Khronos Group, „The OpenGL Shading Language 4.5,“ 14 4 2016. [Võrgumaterjal]. Kättesaadav: https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.50.pdf. [Kasutatud 22 3 2017].

[24] Khronos Group, „Shader Storage Buffer Object - OpenGL Wiki,“ [Võrgumaterjal]. Kättesaadav: https://www.khronos.org/opengl/wiki/Shader_Storage_Buffer_Object. [Kasutatud 30 4 2017].

[25] N. Hajdarbegovic, „Introduction To Vulkan API,“ [Võrgumaterjal]. Kättesaadav: https://www.toptal.com/api-developers/a-brief-overview-of-vulkan-api. [Kasutatud 22 3 2017].

[26] T. Sörman, „Linköpings Universitet,“ 2016. [Võrgumaterjal]. Kättesaadav: https://liu.diva-portal.org/smash/get/diva2:909410/FULLTEXT01.pdf. [Kasutatud 22 3 2017].

[27] J. Varus, „Vulkan bindings for .NET Standard,“ [Võrgumaterjal]. Kättesaadav: https://github.com/discosultan/VulkanCore. [Kasutatud 24 3 2017].

[28] J. Varus, „Experimental Vulkan compute provider for .NET LINQ,“ [Võrgumaterjal]. Kättesaadav: https://github.com/discosultan/LinqToCompute. [Kasutatud 26 3 2017].

[29] A. Danial, „cloc,“ [Võrgumaterjal]. Kättesaadav: https://github.com/AlDanial/cloc. [Kasutatud 30 4 2017].

[30] Khronos Group, „Vulkan - Industry Forged,“ [Võrgumaterjal]. Kättesaadav: https://www.khronos.org/vulkan/. [Kasutatud 7 4 2017].

[31] A. Armstrong, „C#/.NET Bindings for the Vulkan API & SPIR-V,“ [Võrgumaterjal]. Kättesaadav: https://github.com/FacticiusVir/SharpVk. [Kasutatud 24 3 2017].

[32] Mono, „Open source .NET binding for the Vulkan API,“ [Võrgumaterjal]. Kättesaadav: https://github.com/mono/VulkanSharp. [Kasutatud 24 3 2017].

[33] Khronos Group, „The Vulkan API Specification and related tools,“ [Võrgumaterjal]. Kättesaadav: https://raw.githubusercontent.com/KhronosGroup/Vulkan-Docs/1.0. [Kasutatud 24 3 2017].

[34] Microsoft Corporation, „Dispose Pattern,“ [Võrgumaterjal]. Kättesaadav: https://msdn.microsoft.com/en-us/library/b1yfkh5e(v=vs.110).aspx. [Kasutatud 1 5 2017].

[35] Microsoft Corporation, „General Naming Conventions,“ [Võrgumaterjal]. Kättesaadav: https://msdn.microsoft.com/en-us/library/ms229045(v=vs.110).aspx. [Kasutatud 24 3 2017].

[36] Microsoft Corporation, „Platform Invoke Tutorial,“ 2003. [Võrgumaterjal]. Kättesaadav: https://msdn.microsoft.com/en-us/library/aa288468(v=vs.71).aspx. [Kasutatud 24 4 2017].

[37] Microsoft Corporation, „Common Language Runtime (CLR),“ [Võrgumaterjal]. Kättesaadav: https://msdn.microsoft.com/en-us/library/8bs2ecf4(v=vs.110).aspx. [Kasutatud 26 2 2017].

[38] A. Mutel, „SharpDX GitHub Repository,“ [Võrgumaterjal]. Kättesaadav: https://github.com/sharpdx/SharpDX. [Kasutatud 24 3 2017].

[39] OpenTK, „This Open Toolkit library is a fast, low-level C# wrapper for OpenGL and OpenAL,“ [Võrgumaterjal]. Kättesaadav: https://github.com/opentk/opentk. [Kasutatud 24 3 2017].

[40] Microsoft Corporation, „OpCodes.Calli Field,“ [Võrgumaterjal]. Kättesaadav: https://msdn.microsoft.com/en-us/library/system.reflection.emit.opcodes.calli(v=vs.110).aspx. [Kasutatud 24 3 2017].

[41] Khronos Group, „vkCreateInstance(3) Manual Page,“ [Võrgumaterjal]. Kättesaadav: https://www.khronos.org/registry/vulkan/specs/1.0/man/html/vkCreateInstance.html. [Kasutatud 24 3 2017].

[42] A. Shvets, „Visitor Design Pattern,“ [Võrgumaterjal]. Kättesaadav: https://sourcemaking.com/design_patterns/visitor. [Kasutatud 1 5 2017].

[43] M. Warren, „LINQ: Building an IQueryable Provider – Part III,“ 8 2007. [Võrgumaterjal]. Kättesaadav: https://blogs.msdn.microsoft.com/mattwar/2007/08/01/linq-building-an-iqueryable-provider-part-iii/. [Kasutatud 28 3 2017].

[44] T. Lottes, „Vulkan Device Memory,“ 8 8 2016. [Võrgumaterjal]. Kättesaadav: http://gpuopen.com/vulkan-device-memory/. [Kasutatud 5 4 2017].

[45] T. Ruuben, „Digisignaalide Töötlemine,“ [Võrgumaterjal]. Kättesaadav: https://lr.ttu.ee/digisignaalid/Materjalid/DIGISIGNAALID_4.pdf. [Kasutatud 26 3 2017].

[46] B. Watson, Writing High-Performance .NET Code, 2014.

[47] Y. Fujii, T. Azumi, N. Nishio, K. Shinpei ja M. Edahiro, „Data Transfer Matters for GPU Computing,“ [Võrgumaterjal]. Kättesaadav: http://www.ertl.jp/~shinpei/papers/icpads13.pdf. [Kasutatud 30 4 2017].