TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Asko Vendel 206407IAIB

GPU PARALLEELPROGRAMMEERIMISPLATVORMIDE CUDA JA VULKAN VÕRDLUS

Bakalaureusetöö

Juhendaja: Lembit Jürimägi
Lektor

Kaasjuhendaja: Gert Kanter
Vanemlektor

Tallinn 2025

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: Asko Vendel

04.06.2025

Annotatsioon

GPU paralleelprogrammeerimisplatvormide CUDA ja Vulkan võrdlus

Paralleelprogrammeerimine GPU peal on muutunud oluliseks tänu selle laialdasele kasutusele masinõppes, arvutigraafikas ja videotöötluses. On olemas mitmeid platvorme GPU jõu rakendamiseks, kuid CUDA on esile kerkinud oma jõudluse ja kasutusmugavuse poolest. See on aga toonud kaasa olukorra, kus Nvidia on hakanud tõstma vajaliku riistvara hindu.

Käesoleva lõputöö eesmärk on võrrelda Nvidia platvormi CUDA ühe alternatiivse platvormiga Vulkan. Lahendades mõlemal platvormil pilditöötluse ülesande kasutades selleks GPU paralleelset arhitektuuri. Valitud ülesanne hõlmab FIR filtrite rakendamist pildile, mis sobib hästi GPU peal lahendamiseks.

Töö analüüsib saavutatud tulemusi ning võrdleb platvormide arendusprotsessi keerukust, kasutajatuge ja jõudlust. Võrdluse eesmärk on anda ülevaade, millised on kummagi platvormi tugevused ja piirangud GPU peal põhineva pilditöötluse kontekstis.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 30 leheküljel, 6 peatükki, 16 joonist ja 4 tabelit.

Abstract

GPU-based parallel programming has become increasingly important due to its widespread use in machine learning, computer graphics, and video processing. There are several platforms available for harnessing the computational power of GPUs, but CUDA has emerged as a leading solution due to its performance and ease of use. However, this has led to a situation where Nvidia has started raising the prices of the necessary hardware.

The aim of this thesis is to compare Nvidia's CUDA platform with an alternative solution — Vulkan. A typical image processing task will be implemented on both platforms, utilizing the GPU's parallel architecture. The chosen task involves applying a FIR filter to an image, which is well-suited for GPU-based computation.

The thesis analyzes the results and compares the platforms in terms of development complexity, user support, and performance. The goal of the comparison is to provide an overview of the strengths and limitations of both platforms in the context of GPU-based image processing.

The thesis is written in Estonian and is 30 pages long, including 6 chapters, 16 figures and 4 tables.

Lühendite ja terminite loetelu

AI	*Artificial Intelligence*, Tehisintellekt
API	*Application Programming Interface*, Rakendusliides
CPU	*Central Processing Unit*, Keskprotsessor
CUDA	*Compute Unified Device Architecture*, Nvidia graafikakaardi paralleelprogrammeerimiskeskond ja API
FIR (filter)	*Finite Impulse Response (filter)*, Lõpliku siirdega filter
GPU	*Graphics Processing Unit*, graafikaprotsessor(graafikakaart)
GLSL	*OpenGL Shading Language*, Programmeerimiskeel *shader* failide kirjutamiseks
HIP	*Heterogeneous-computing Interface for Portability*, Paralleelprogrammeerimisplatvorm
HLSL	*High Level Shader Language*, Programmeerimiskeel *shader* failide kirjutamiseks
IDE	*Integrated Development Environment*, programmeerimiskeskond
NVCC	*Nvidia CUDA Compiler*, Kompilaator CUDA koodi jaoks
RGB	*Red Green Blue*, Piksli värvitoonid
RAM	*Random Access Memory*, põhimälu
SDK	*Software Development Kit*, tarkvaraarenduskomplekt
SPIR-V	*Standard Portable Intermediate Representation*
VRAM	*Video Random Access Memory*, graafikakaardi mälu

Sisukord

Joonistе Loetelu

Tabelite Loetelu

1. Sissejuhatus

Paralleelprogrammeerimine või ka paralleelarvutus on programmeerimisviis, mille käigus programm kasutab ülesannete täitmiseks mitut protsessori tuuma või ka ühe protsessori tuuma mitut lõimu. Kuigi tehnoloogia arenguga on CPU-de ehk keskprotsessorite tuumate arv kasvanud, siis kasutatakse paralleelprogrammeerimises tihtipeale peale keskprotsessorite ka graafikakaarte ehk GPU-sid.

1.1 Probleem

Viimastel aastatel on nõudlus graafikakaartidele märgatavalt kasvanud. Selle kasvu põhjuseid on mitmeid, nagu näiteks viimaste aastate AI kiire areng, kuid ka sellele eelnenud krüptorahade kaevandamise populaarsuse kasv. Eriti palju on see mõjutanud graafikakaartide tootja Nvidia kaarte, kuid puutumata ei ole jäänud ka tootjad AMD ja Intel.

Nvidia graafikakaartide suurem hinnakasv võrreldes AMD ja Intel-ga on tingitud Nvidia tehnoloogilistest eelistest, mille tulemusel on Nvidia graafikakaartide rakendamisel paralleelprogrammeerimisel võrreldes konkurentidega parem jõudlus. See parem jõudlus omakorda tuleneb suuresti Nvidia CUDA ökosüsteemi olemasolust. Tänu sellele on tekkinud olukord, kus graafikakaartide resursside rakendamiseks programmeerimisel on Nvidia graafikakaardid tihti kõige effektiivsem valik. See omakorda suurendab Nvidia graafikakaartide turuosa ning nõudlust nende graafikakaartide järgi, mis omakorda suurendab nende hinnakasvu ning teeb nad veelgi raskemini kättesaadavaks.

CUDA on paralleelprogrammeerimisplatvorm Nvidia graafikakaartide rakendamiseks probleemide lahendamisel. Platvormi omanik ja arendaja Nvidia on sinna pikalt investeerinud ning selle tulemusel on CUDA hästi optimiseeritud graafikakaartide riistvara kasutama. Lisaks on platvormil hea tugi ning palju resursse, mis teevad sellel arendamise lihtsamaks. Selle platvormi peamine miinus on see, et see töötab ainutl Nvidia graafikakaartidel.

Paralleelprogrammeerimiseks graafikakaartidel on siiski mitmeid alternatiive CUDA-le, kuigi ükski neist ei ole küll nii küpse ökosüsteemiga. Üks huvitav alternatiiv on platvorm nimega Vulkan. See on küll peamiselt loodud graafika kuvamiseks mängudes, kuid tänu selle modulaarsusele on seda võimalik ka rakendada üldotstarbeliste arvutusprobleemide lahendamiseks. Peamine eelis on see, et Vulkan töötab mitmete tootjate graafikakaartidel. Sellest tulenevalt tekib küsimus, kui sobilik alternatiiv oleks Vulkan CUDA-le?

1.2 Lõputöö ülevaade ja eesmärk

Lõputöö käigus on uuritud erinevaid GPU paralleelprogrammeerimisplatvorme ning probleeme, mida saab effektiivsemalt lahendada kasutades GPU paralleelset riistvara. Platvormidest on välja valitud CUDA ja Vulkan, ning tehtud neist eri aspekte hõlmav võrdlus. Platvormide võrdlemiseks on valitud pilditöötlus valdkonda kuuluv FIR filtrite rakendamine piltidele.

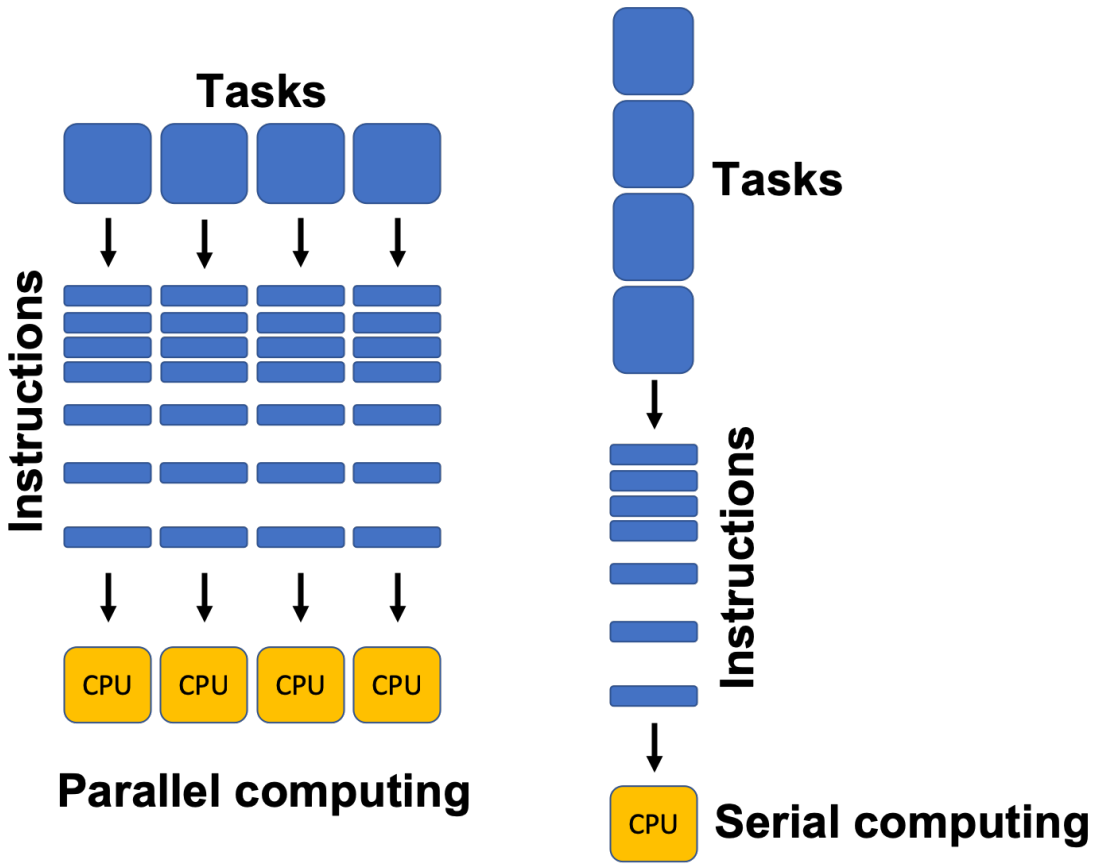
Lõputöö eesmärgiks on anda lugejale ülevaade CUDA ja Vulkan-i erinevustest läbi FIR filtri implementeerimise mõlemal platvormil. Võrreldud on platvorme kasutajatoe, arenduse ja jõudluse poole pealt.

2. Programmeerimismudel

2.1 Klassikaline mudel

Klassikaline programmeerimine, inglise keeles *sequential programming* või *serial programming* on programmeerimisviis, kus programmi kood jookseb ühes lõimus, kasutades ainult protsessori ühte tuuma. See tähendab, et ülesanded täidetakse sammhaaval üksteise järel nii, et enne iga järgmise sammu algust peab eelnev samm olema lõppenud. Näide sellest on joonise [1] parem pool.

Sellisel viisil programmeerimine sobib hästi väiksemate probleemide lahendamiseks. Samuti on tarkvara arendamine ning vigade leidmine lihtsam võrreldes paralleelse programmeerimisega. Kui aga tuleb ette suuremahulisem või keerulisem probleem, siis võib klassikaline programmeerimine osutuda liiga ebaeffektiivseks.



Joonis 1. Paralleelne ja klassikaline programmeerimismudel.[1]

2.2 Paralleelne mudel

Tänapäeva protsessoritel on mitu füüsilist tuuma. Tuumadel, omakorda, võivad protsessid joosta mitmes erienvas lõimus. Paralleelne programmeerimine ehk inglise keeles *parallel programming* või *parallel computing* on programmeerimisviis, mis rakendab rohkem kui ühte protsessori tuuma ja lõimu. Võrreldes klassikalise programmeerimisega, on paralleelne programmeerimine suurema keerukusega. Peamiseks probleemide põhjuseks on töö sünkroniseerimine ja ressursside jagamine.

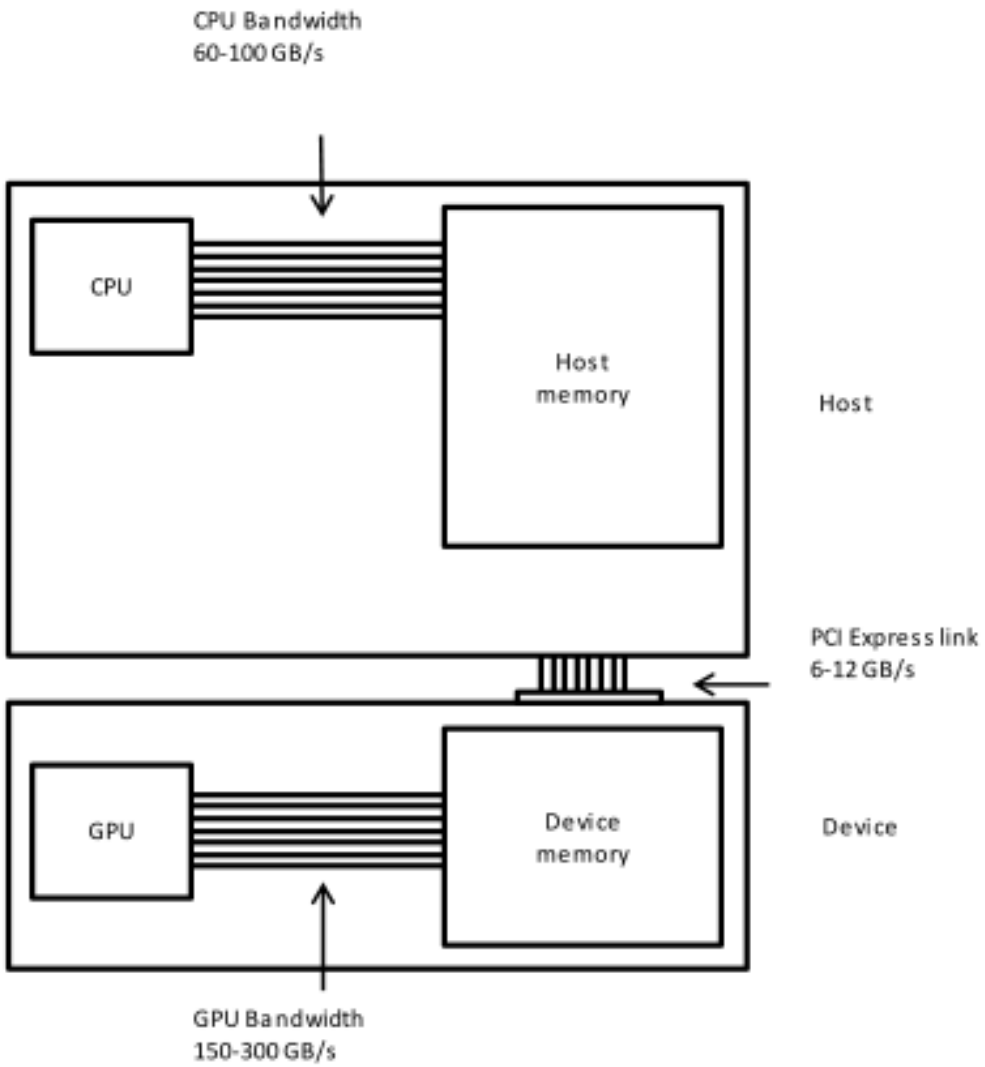
Näiteks võib tekkida probleem race condition[2] siis kui kaks lõimu peavad lugema samalt mäluaadressilt andmeid ning neid õiges järjerkorras muutma, aga ei ole mingit garantiid, et üks lõim jõuab andmed lugeda, muuta ja siis uuesti mällu kirjutada enne kui teine lõim jõuab andmed mällu lugeda. See tulemuseks on lõpuks mälus olevad andmed valed.

Teine suurem problem tekib kui erinevad lõimud jagavad samu ressursse ning sellel probleemil on kaks varianti, livelock ja deadlock[3]. Deadlock tekib kui üks lõim vajab resurssi, mis on teise lõimu kasutusel ning teine lõim vajab ressurssi, mis on esimese lõimu kasutusel. Sellises olukorras jäävad mõlemad lõimud lõputult ootama, et üksteise kasutuses olevad ressursid vabaneksid. livelock on situatsioon kus mõlemad lõimud muudavad oma olekut selleks, et ressurssi vabastada ja deadlock-i vältida, kuid kumbki lõim ei saa teise lõimu oleku tõttu tööga jätkata.

2.3 Paralleelne programmeerimine graafikakaartidel

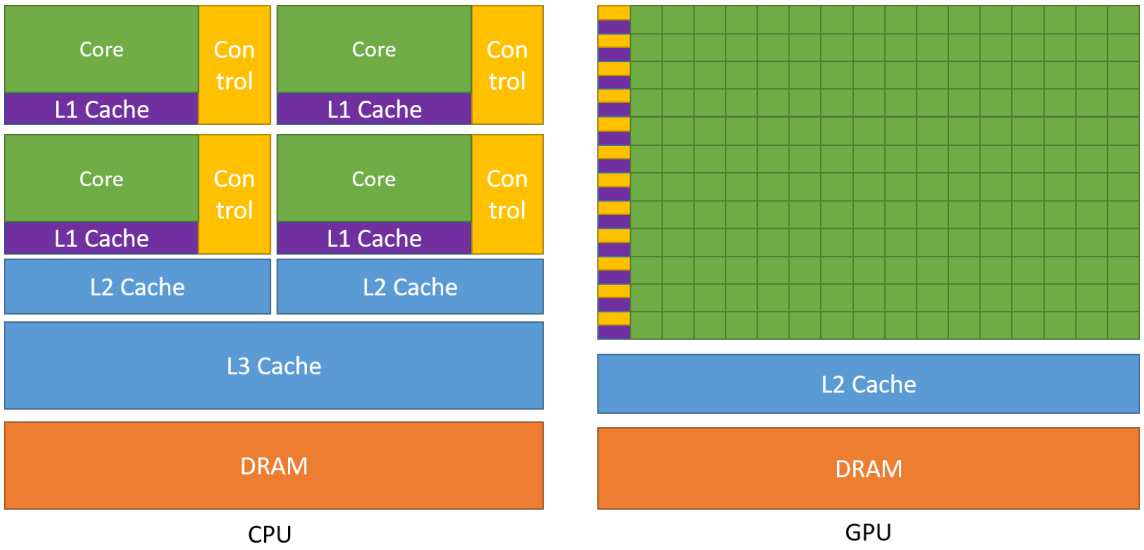
Algselt loodi graafikakaardid reaalajas 3D graafika kuvamise kiirendamiseks. Seega kasutati neid arvutimängude jaoks ja video- ning pilditöötluses. Viimasel ajal on graafikakaartidele leitud veelgi uusi kasutusalasid. Näiteks sobivad graafikakaardid oma paralleeluse tõttu suurepäraselt AI ning eelkõige masinõppe probleemide lahendamiseks.

Graafikakaardid ehk GPU-d on eriotstarbelised protsessorid. Nad jagunevad integreeritud ja diskreetseteks graafikakaartideks. Integreeritud graafikakaardid jagavad süsteemimälu CPU-ga, kuid diskreetsed graafikakaardid on protsessorist täiesti eraldiseisev üksus ning neil on omaenda mälu, mis on üldiselt süsteemi muutmälust kiirem. Seega graafikakaartide rakendamiseks paralleelprogrammeerimises tuleb andmed viia GPU mällu.



Joonis 2. CPU ja GPU lahusus.[4]

Kui võrrelda veel CPU-d ja GPU-d, siis peamine erinevus seisneb tuumades. CPU-l on vähem tuumasid, ligikaudu 4-8 tuuma. GPU-l aga võib olla neid isegi tuhandetes. Lisaks sellele erinevad CPU ja GPU tuumad ehituse poolest. Sellest tulenevalt sobib CPU rohkem keerulistemate, kuid järjestikulist ja GPU lihtsamate, kuid paralleelsete probleemide lahendamiseks. Joonisel 3 on kujutatud CPU ja GPU tuumade arvu erisust.



Joonis 3. CPU ja GPU tuumade võrdlus.[5]

2.3.1 GPU paralleelprogrammeerimise platvormid

Selleks, et rakendada graafikakaartide paralleelset riistvara on tehtud mitmeid platvorme. Mõned neist platvormidest on loodud graafikakaartide tootjate poolt, nagu näiteks Nvidia CUDA, AMD HIP ja Intel oneAPI. Teised platvormid nagu näiteks OpenCL, Vulkan ja SYCL on vabavaralised ja nende arendamine toimub erinevate tarkvara ja tehnoloogia ettevõtete poolt moodustatud *Khronos Group*-i[6] poolt.

Nvidia CUDA platvorm on üks populaarsemaid platvorme graafikakaardi rakendamiseks paralleelprogrammeerimises. See on tuntud hea kasutajatoe ja jõudluse poolest ning see töötab programmeerimiskeeltega nagu C, C++, Fortran ja ka Python läbi PyCuda teegi. Selle peamiseks miinuseks on, et see töötab ainult NVIDIA enda graafikakaartide peal.

AMD HIP on CUDA-ga konkureeriv platvorm, kuid see pole nii laialdaselt kasutuses. On peamiselt suunatud AMD graafikakaartidele, kuid töötab ka Nvidia graafikakaartidel. Võimaldab lihtsalt CUDA-le kirjutatud koodi ümber konverteerida nii, et see töötaks ka HIP-l.

Vulkan on suhteliselt uus platvorm ning OpenGL-i järglane. Selle kasutamine on mõnevõrra keerulisem, kuid võrreldes OpenGL-ga, pakub see paremat jõudlust. Peamised eelised on avatud lähtekood ja kasutatavus kõikide tootjate graafikakaartidega.

2.3.2 Platvormide valik

GPU paralleelprogrammeerimise platvorme on mitmeid, kuid CUDA on kõige laialdasemalt kasutusel hea jõudluse ja kasutajatoe tõttu. Seega Peaks CUDA olema üks platvormidest võrdluses. Teisest küljest sobiv võrdlusesse kõige paremini Vulkan, sest selle kasutamine ei ole piiratud riistvarast, ehk teisisõnu see töötab kõikide tootjate graafikakaartidel ning see on vabavaraline. On ka teisi platvorme, millel on need eelised, kui võrreldes Vulkan-ga, ei ole nad nii moodsad ja jäävad nad ka jõudluse poolest alla.

3. Ülevaade võrreldatavatest platvormidest

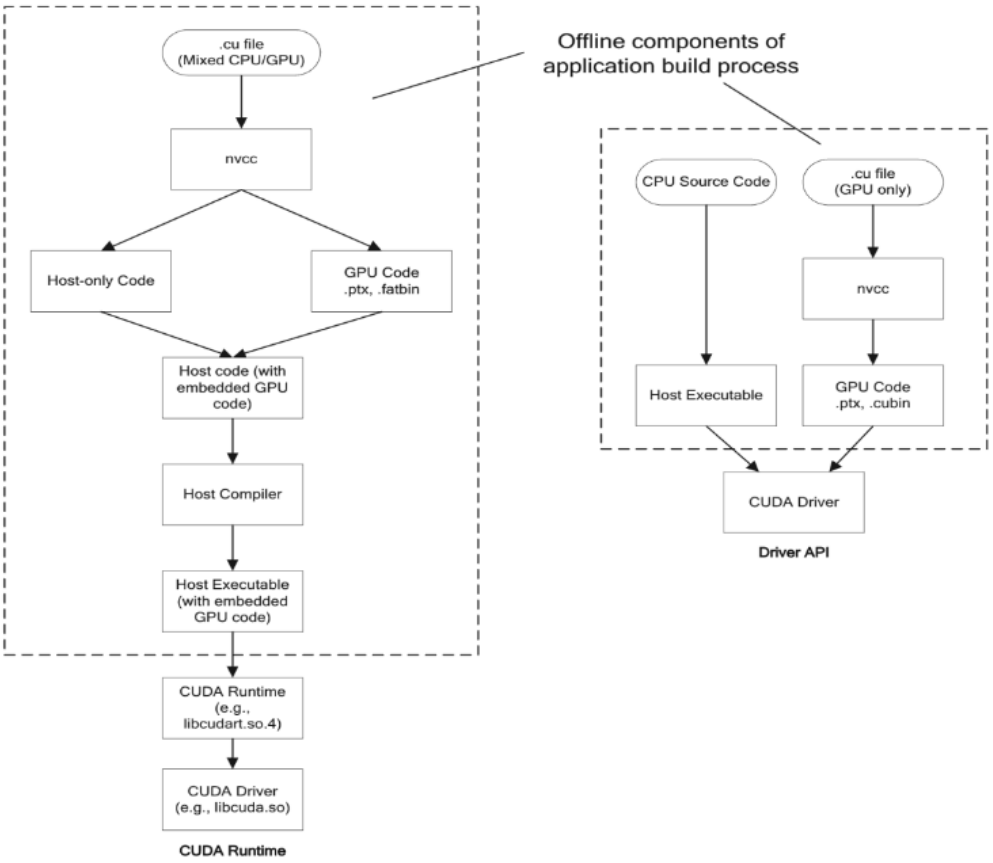
Lõputöö käigus on kasutatud Nvidia CUDA ja Khronos Vulkan GPU paralleelprogrammeerimisplatvorme. Siin peatükis on kirjas nendest tehnoloogiatest täpsemalt. Lisaks on veel kirjas lahendatavas pilditöötluse probleemist ning alternatiivsetes probleemidest mis olid kaalumisel.

3.1 CUDA

CUDA on Nvidia paralleelprogrammeerimis platvorm ning API graafikakaardiga suhtlemiseks. Selle kasutamiseks on vajalik Nvidia graafikakaart, mis toetab CUDA-t. Selleks tuleb alla laadida ja installeerida Nvidia CUDA toolkit. CUDA toolkit koosneb *nvcc* kompilaatorist, erinevatest arendamiseks kasutatavatest teekidest, dokumentatsioonist, IDE-de laiendustest ning graafikakaardi draiveritest.

CUDA toetab nii Microsoft Windows kui ka Linux operatsioonisüsteemi. CUDA-s kirjutatud rakendused on keeles CUDA-C, millel on suur sarnasus keelega C++. Siiski, saab CUDA-t kasutada ka teiste keeletega nagu Python, läbi PyCuda teegi.

Rakendused on kirjutatud .cu faili, mis CUDA-C ja C++ koodist, kuid üldjuhul on C++ koodi rohkem kui CUDA-C koodi. CUDA rakenduste loomine käib läbi *nvcc* kompilaatori. *nvcc* võtab CUDA-C koodi ning saadab ülejäänud koodi C++ kompilaatorile. Läbi *nvcc* saab ka täpsustada millisele tasemele koodi compileerida. Joonisel 4 on näha erinvad sammud, mille kood läbib kompileerimisel. Rakendused saab kirjutada kasutades *CUDA runtime API*-d või *driver API*-d.



Joonis 4. nvcc töövoog.[7]

Runtime API on üks võimalik CUDA kasutajaliides graafikakaardi ressursside rakendamiseks. See lihtsustab märgatavalt CUDA rakenduste kirjutamist, automatiseerides teatud sammud nagu näiteks seadme initsialiseerimine ja mäluhaldus. Seda kasutades on võimalik kiiremini ja lihtsamini kirjutada CUDA rakendusi jõudluse arvelt.

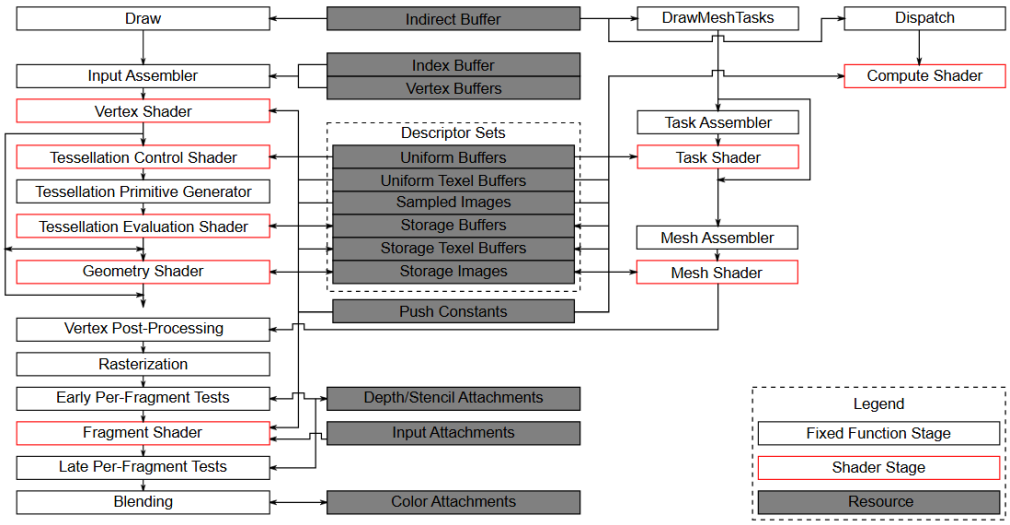
Driver API kasutajaliides pakub kasutajale suuremat kontrolli graafikakaardi ressursside üle. Nagu eelpool mainitud siis, võrreldes Runtime API-ga peab kasutaja ise seadmed initsialiseerima. Selle tõttu on rakenduse kood mahukam, kuid võimaldab näiteks kasutata mitut seadet arvutuste tegemiseks.

3.2 Vulkan

Vulkan on *Khronos Group*-i poolt loodud API graafikakaardi riistvara rakendamiseks. Selle on suureks eeliseks on see, et selle kasutamine ei ole piiratud ühe GPU tootja riistvarale ning Vulkan-i SDK on avatud lähtekoodiga.*Vulkan*-i ülesseadmine on suhteliselt lihtne. Vajalik on alla laadida ning installeerida Vulkan SDK.

Vulkan-i rakendused kirjutatakse tavalises C/C++ keeles, kuid rakendustega kaasneb vajadus luua eraldi shader failid, mis on kirjutatud kas HLSL või GLSL keeles. Mõlemal keelel on suur sarnasus C/C++-ga ning seetõttu ei ole shader-te loomine väga keeruline ülesanne.

Joonisel 5 on kirjeldatud erinevaid Vulkan-i *pipeline*-e ning nendevahelisi seoseid. *Pipeline* saab kategoriseerida vastavalt graafikaoperatsioonidele suunatud ja arvutusoperatsioonidele suunatud osadeks.



Joonis 5. Vulkan-i *pipeline*-d

4. Lahendus

FIR filter on digitaalne filter, mida kasutatakse singaalitöötluses signaali töötlemiseks ja parandamiseks. Peamiselt kasutatakse FIR filtrit heli- ja pildisiginaali töötlemiseks. Kaks enim tuntud FIR filtrit on kõrgpääsfilter(*Highpass filter*) ja madalpääsfilter(*Lowpass filter)*. Need blokeerivad vastavalt madala ja kõrge sagedusega signaali. Kõrgpääsfiltri rakendamine pildile toob pildi servad välja, seda kasutatakse häguse pildi teravdamiseks. Madalpääsfilter vastupidiselt hägustab pilti, sellest on eelkõige kasu pildilt "müra" eemaldamiseks.

Lõputöö käigus on teostatud kõrgpääs- ja madalpääsfilter kasutades CPU-d ja GPU-d. Kuigi lõputöö eesmärgiks on võrrelda GPU paralleelprogrameerimisplatvorme, siis veendumaks, et see on sobiv probleem mida lahendada ning ka selle testimiseks, on filtrid implementeeritud ka veel täielikult CPU-d kasutades.

Teostus on jaotatud kolmeks suuremaks peatükiks. Esimeses peatükis on kirjeldatud filtrite teostust CPU-d kasutades. Selles peatükis on kirjeldatud täpsemalt kuidas FIR filter töötab ning kuidas nende loogika on implementeeritud. GPU rakendamiseks on peamised vajaminevad sammud GPU mälu eraldamine, andmete transportimine GPU mällu ning viimaks programmi loogika teostamine kasutades GPU paralleelseid tuumasid. Seetõttu keskenduvad mõlemad CUDA ja Vulkan peatükkid nendele sammudele, kuid kirjeldades kuidas mõlemal platvormil nende sammude läbiviimine toimub.

4.1 FIR filtrite teostus CPU-l

Enne filtrite loomist on vaja laadida pildi andmed arvuti põhimällu ning veenduda, et nad on kujul, mis on sobiv edasiseks töötlemiseks. Need kaks sammu on vajalikud ka CUDA ja Vulkaniga töötamisel, aga kuna isegi GPU-d kasutades on need sammud teostatud puhtalt CPU peal, siis on need sammu kirjeldatud selles peatükis.

4.1.1 Pildi laadimine mällu

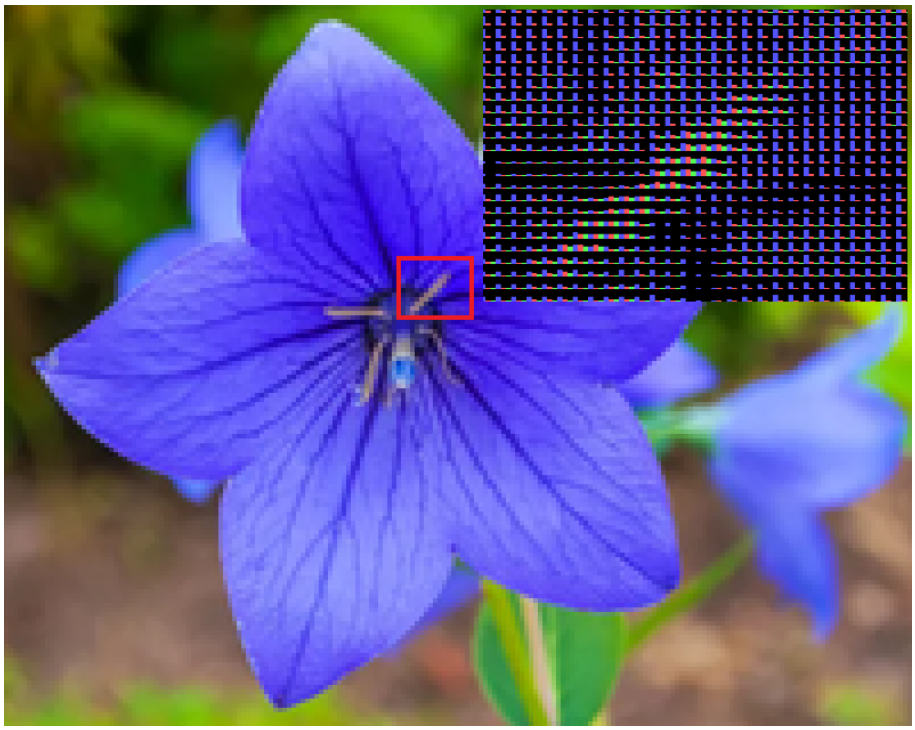
Selleks, et pilti töödelda, on vaja pilt laadida andmete kujul arvutimällu. Selleks saavutamiseks on mitu erinevat varianti. Üks variant on kirjutada ise kood, mis arvestades pildi formaati loeks pildi mällu. Kuna pilditöötlus ise ei ole lõputöö fookuses, siis oleks sobilikum teine variant, mis eeldaks mõne juba valmiskirjutatud tööriista kasutamist, mis

pakuks pildi kodeerimise ja dekodeerimise võimalust.

Mõned sellist võimalust pakkuvad tööriistad on tarkvarapakkett *ImageMagick*[8] ning teegid *LodePNG*[9] ja *stb*[10]. *ImageMagick* põhjalikke võimalusi pilditöötluseks ning selle kasutamine on suhteliselt keeruline. Lisaks kuna meil on vaja ainult pilt laadida mällu, siis sobiksid eelnimetatud teegid paremini. *LodePNG* ja *stb* kasutamine on suhteliselt sarnane. Nende peamine erinevus seisneb selles, et *LodePNG* töötab ainult PNG formaadis piltidel, sama kui *stb* töötab mitme erineva formaadiga. Seetõttu langes otsus *stb* kasuks.

4.1.2 Andmed mälus

Peale pildi laadimist mällu, paiknevad andmed *array* ehk massiiv andmetüübis järjestikuliselt nii, et iga liige tähistab pildi ühe piksli R, G ja B väärtust. Need tähistavad vastavalt piksli punast, rohelist ja sinist värvitooni. Mõnel juhul on ka veel väärtus A ehk *alpha channel*, mis on piksli läbipaistvus, kuid see sõltub pildifaili formaadist. Iga elemendi väärtus on vahemikus 0 kuni 255 ning see tähistab värvitooni tugevust. Selle näiteks on toodud pilt 6, millest on punase ristkülikuga tähistatud ala värvitoonide tugevused välja toodud. Iga tähistatud ala pikslile vastavad väljalõikes kolm sammast, mis tähistavad selle piksli RGB värvitoonide tugevust.



Joonis 6. piksli RGB väärtuste näide.

Kuna andmed paiknevad massiivis järjest ilma ühegi eraldajata, siis on neid raske töödelda,

sest me ei tea milline väärtus vastab millisele pikslile. Selleks tuleb andmed ümber tõlgendada sellisele kujule, mis annaks meile aimu milline väärtus vastab millisele pikslile. Selleks tõstame andmed ümber nii, et massiivis on pildi laiuse ehk *width* arv elemente ja iga element on omakorda massiiv, milles on pildi kõrguse, ehk *height* arv elemente. Viimaks iga element selles massiivis on samuti massiiv, mis koosneb kolmest elemendist mis tähistavad eelnevalt nimetatud RGB värvitoonide tugevust. Sellisel kujul sarnanevad andmed maatriksile, mille iga element on piksli RGB väärtuste kolmik.

4.1.3 FIR filtrid

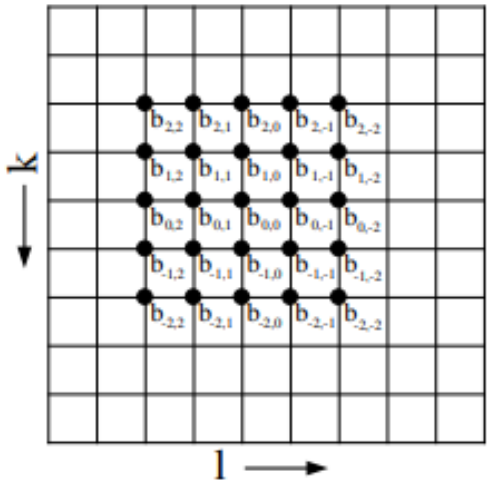
Lühidalt seletades on FIR filter sellist tüüpi filter, mis töötleb sisendsignaali nii, et väljundsignaaliks on sisendsignaali ning sellele eelnevate signaalide summa, kus iga signaal on korrutatud läbi mingi valitud koefitsendiga. Seda saab esitada valemiga (4.1)[11]. Selles valemis $y[n]$ tähistab väljundsignaali, $x[n]$ sisendsignaali, b_k koefitsenti ja M filtri järku, ehk mitu eelnevat signaali mõjutavad väljundsignaali.

$$y[n] = \sum_{k=0}^{M} b_k \cdot x[n-k] = b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] + \cdots + b_M x[n-M] \quad (4.1)$$

Selliselt kujul sobib see filter ühe dimensionaalse signaali, nagu näiteks helisignaali töötlemiseks. Selleks, et kahe dimensionaalset signaali, nagu pikslite maatriksi filtreerida, saab seda valemit (4.1) natukene muuta, ning tulemuseks on valem (4.2)[11]

$$y[m,n] = \sum_{k=-M_1}^{M_2} \sum_{l=-N_1}^{N_1} b_{k,l} \cdot x[m-k, n-l] \quad (4.2)$$

Seda võib visuaalselt ette kujutada kui kasti, mille keskpunktiks on pildi piksel $b_{0,0}$ (joonis 7). See on sisendsignaal ning seda ümbritsevad pikslid on signaalid, mis mõjutavad sisendsignaali väärtust. Sellist filtrit tuleks rakendada igale pildi pikslile. Siiski tuleks meeles pidada, et igal filtrisse jääval $b_{m,n}$ koefitsiendil on erinev väärtus, mis sõltub vastavalt sellele, kuidas pilti tahetakse töödelda.



Joonis 7. filtri kahe dimensionaalne representatsioon.[11]

Filtri teostuses on kõigepealt loodud funktsioon, mis loob joonis 7 peal kujutatud sarnase $N \cdot N$ suuruse filtri kasti ning täidab selle koefitsientidega. Selline lahendus võimaldab luua erineva suurusega filtreid, mis omakorda võimaldab rakendada GPU-d paremini, kuna filtri suuruse kasvades, suureneb märgatavalt arvutuste arv, mida tuleb iga piksli filtreerides teha.

Koodis piisab selle filtri "kasti" kujutamiseks lihtsalt massiivi loomisest, mis hoiab koefitsientide väärtusi. madalpääsfiltri puhul on koefitsientide jaoks mitmeid variante, antud töös on kasutatud keskmistamist, see tähendab et iga koefitsiendi võib väärtustada 1-ga, sest see filter võtab kõikide võrreldavate pikslite keskmise. Sellest tulenevalt ongi madalpääsfiltril pildile hägustav effekt. Lisaks on teisi variante nagu näiteks Gauss-i[12] filter. Siin töös Gaussi filtrit ei ole rakendatud, kuna teatud liiki müra vähendamiseks ta ei ole nii sobilik. Lisaks hägustamise saavutamiseks peab olema filter suurem kui keskmistamise puhul. Kõrgpääsfiltri puhul on sarnane lähenemine, kuid koefitsientide väärtused on erinevad. Üks näide nendest koefitsientidest on valemis (4.3)[11], kuid seda filtrit on raske suuremaks teha, sest sobivad koefitsiendid tuleb arvutamise teel leida.

$$\begin{aligned} y[m,n] = \frac{1}{4}x[m+1,n+1] \quad &- x[m+1,n] \quad + \frac{1}{4}x[m+1,n-1] \\ - x[m,n+1] \quad &+ 3x[m,n] \quad - x[m,n-1] \\ \frac{1}{4}x[m-1,n+1] \quad &- x[m-1,n] \quad + \frac{1}{4}x[m-1,n-1] \end{aligned} \tag{4.3}$$

joonisel 8 on näha osa pildist, millele on rakendatud mõlemat FIR filtrit. Pilt 8a on originaalne pilt, millel on palju müra. Pilt 8b on originaalne pilt pärast madalpääsfiltri rakendamise, mis on hägustamise teel müra vähendanud ning pildile 8c on rakendatud kõrgpääsfiltrit, et seda uuesti teravdada.

(a) originaalne pilt

(b) madalpääsfiltri rakendamine originaalsele pildile

(c) kõrgpääsfiltri rakendamine hägustatud pildile

Joonis 8. madal- ja kõrgpääsfiltri effekt pildile.

4.2 Lahendus CUDA platvormil

Suur osa CUDA lahenduse koodist ühtib eelmises sektsioonis kirjeldatud lahenduse koodiga. Kood on kirjutatud CUDA-C keeles, mis on tegelikult C++ laiendus. Seega saab eelmises sektsioonis kirjutatud koodi võtta aluseks ning teha vajalikud muudatused. Lahendus on kirjutatud kasutades runtime API-d, kuna puudub vajadus driver API poolt pakutavale kontrollile rakenduse üle ning sellega väldime ka liigset keerukust, mis kaasneks driver API kasutamisega.

Vajalikud muudatuse saab kokku võtta lühidalt. Esiteks on vaja eraldada graafikakaardi mälu kasutatavate andmete jaoks. Teiseks, tuleb transportida andmed graafikakaardi mällu ehk VRAM-i. Kolmandaks, on vaja muuta rakenduse loogikat nii, et see kasutaks CUDA tuumasid. Viimaks tuleb andmed uuesti transportida VRAM-st tagasi põhimällu.

4.2.1 Mälu eraldamine ning andmete kopeerimine

Graafikakaardi mälu eraldamine toimub sarnaselt tavamälu eraldamisele C++ keeles. Selleks on *cudaMalloc* funktsioon, mis sarnaneb C++ funktsiooniga *Malloc*. Selle argumentideks tuleb anda muutuja, mis viitab eraldatavale mälule ning eraldatava mälu suuruse baitides. Hiljem saab *cudaFree* funktsiooni abil eraldatud mälu uuesti vabastada.

Andmete transportimiseks on funktsioon *cudaMemcpy*. See võtab argumentideks muutuja, mis viitab andmetele GPU mälus, muutuja mis viitab kopeeritavatele andmetele, andmete mahu baitides ning väärtuse, mis tähistab kopeerimise suunda. Andmete transportimiseks graafikakaardi mällu on selleks väärtuseks *cudaMemcpyHostToDevice*. Peale andmete töötlemist saab selle funktsiooni abil andmed uuesti tagasi viia *RAM*-i, kasutades väärtust *cudaMemcpyDeviceToHost*.

4.2.2 Kernel-i loomine

Suur osa arvutustest, mida tuleb ainult ühe korra arvutada jääb siiski CPU kanda. Need on näiteks pildifaili põhimällu laadimine ning filtri "kast"-i loomine. Need I ole nii mahukad arvutused, seega nende tegemine kaasutades GPU-d ei annaks erilist ajalist võitu. Filtri saab üle kanda videomällu ning see lihtsalt kernelile kaasa anda.

kernel on eraldi funktsioon ning platvormil CUDA tähistatakse see funktsioon vastava spetsifikaatoriga. Näiteks spetsifikaator *__global__* näitab, et funktsiooni saab välja kutsuda CPU, kuid see täidetakse GPU-l. Selles funktsioonis saab olema kogu see töö, mis on vaja lahendada kasutades GPU tuumasid, ehk teisisõnu filtrite rakendamine igale pildi pikslile.

CUDA-l on sisseehitatud objektid *blockIdx*, *threadIdx* ja *blockDim*, mis tähistavad vastavalt ploki indeksit, lõimu indeksid ja ploki lõimude koguarvu. Neid objekte kasutades on võimalik igale pikslile määrata lõim, seega iga lõim saab näiteks ühe pildi piksli, mille väärtus uuesti arvutada.

4.3 Lahendus Vulkan platvormil

Pilditöötlus probleemi lahendamine platvormil Vulkan on mõnevõrra keerulisem võrreldes CUDA-ga. Kuigi Läbitavad sammud on suures pildis samad. Tuleb eraldada GPU mälu ja andmed sinna kopeerida, kirjutada GPU riistvara kasutav kernel(Vulkan-i puhul shader) ning viimaks uued andmed tagasi põhimällu transportida, siis Vulkan-i puhul on koodi palju rohkem.

4.3.1 Initsialiseerimine

Vulkan-i rakenduse loomine algab *instance*-i loomisega. Selleks on vaja luua *InstanceCreateInfo* objekt ning, mis hoiab endas Vulkan-i rakendusega seotud informatsiooni nagu näiteks rakenduse nimi, API-i versioon ja laiendused. *instance*-i luues tuleb see objekt kaasa anda.

Järgmiseks tuleb luua *PhysicalDevice* objekt, mis esindab GPU-d mida kasutame andmete töötlemiseks. *PhysicalDevice* objekt esindab füüsilist seadet, kui selle kasutamiseks on meile vaja loogilist seadet. Seega on vaja luua *PhysicalDevice* objekti kasutades *Device* objekt, kuid selleks on vaja luua *Queue* objekt, mis hoiaks andmeid, mida seade ehk *GPU* hakkaks töötlema. Oluline on selle juures, et oleks valitud selline *Queue*, mis sobiks *Vulkan Compute-ga*, kuna Vulkan toetab erinevaid tüüpi *Queue*-d ja kõik ei oli sobilikud iga *Vulkan*-i osaga. Peale *Device* loomist on eeltöö tehtud.

4.3.2 Andmete kopeerimine mällu

Andmete seadme mällu kopeerimine Vulkan-s koosneb mitmest sammust. Kõigepealt tuleb luua mälupuhver, siis tuleb eraldada puhvrile mälu ning viimaks tuleb puhver ja mälu omavahel siduda. See teeb andmete kopeerimise mällu mahukaks ülesandeks, kui teisest küljest annab see programmeerijale suuremat kontrolli rakenduse mälu kasutuse üle.

Mälupuhvri loomiseks tuleb kõigepealt luua *BufferCreateInfo* objekt, mis hoiab endas puhvri loomiseks vajalikku informatsiooni. Sealhulgas on informatsioon nagu puhvri suurus baitides, puhvri kasutust kirjeldav informatsioon. Näiteks tuleb märkida *BufferUsageFlagBits::eTransferSrc*- ja *BufferUsageFlagBits::eStorageBuffer*-ga, et puhvrit kasutatakse nii andmete liigutamiseks süsteemimälust seadmemällu ja ka andmete hoidmiseks.

Mälu eraldamiseks on kõigepealt vaja leida millist mälu on vaja. Selleks on funktsioon *getBufferMemoryRequirements*, mis tagastab vajalikud nõuded mälule. Seda informatsiooni

kasutades saab API-lt küsida vajaliku mälu, mida näeb ka CPU ehk Host. Kogu eelneva informatsiooni põhjal saab funktsiooni *allocateMemory* kaudu eraldada seadmes mälu, mis hoiaks pildi andmeid.

4.3.3 Shader-i loomine

GPU peal tehtava töö jaoks on vaja luua eraldi shader fail, seda võib mingil määral vaadelda kui faili, mis hoiab endas *kernel*-t. See on kas HLSL või GLSL laiendiga fail, mis on kirjutatud samanimelises keeles, mis sarnaneb C-le. See fail koosneb koodist mida GPU peal paralleliseerida, ehk siis filtri rakendamine igale pildi pikslile. Saranaselt CUDA-ga on vaja lõimud indekseerida ning ka see on selles failis. Lisaks programmi loogikale ja lõimude indekseerimisele on vaja lisada HLSL või GLSL faili veel konstantide definitsioonid, mida kasutatakse piksli koordinaatide ümberavutamisel ning lõimu grupi suurus.

Esialgselt oli shader-ks HLSL fail, mis sai valitud kuna HLSL keel tundus loogilisem ja paremini struktueeritud võrreldes GLSL-ga. Lahenduse teostuse käigus tuli HLSL puhul välja üks märgatav puudujääk. HLSL ei toeta *uint8_t* andmetüüpi ning kuna pildifaili pikslid on mälus *unsigned char* andmetüübina, mis on funktsionaalselt sama *uint8_t*-ga, siis oleks shader-i loomiseks tulnud teha suuri muutusi kogu rakenduse töös. Seetõttu sai mindud üle GLSL shader-le, millel on *uint8_t* tugi olemas.

Vulkan ei suuda lugeda HLSL või GLSL formaadis shadere faili[13]. Seega tuleb see fail kompileerida SPIR-V baitkoodi. Selleks on kasutatud glslc compilaatorit, mis loob GLSL failist spv ehk SPIR-V faili. HLSL puhul loob DirectX Shader Compiler spv faili.

4.3.4 Pipeline-i loomine

Pipeline loomine koosneb samuti mitmest sammust. Alustuseks tuleb luua Vulkan *Shader-Module* object, mis loeb eelnevast punktis loodud SPIR-V faili.

Kuigi konstantide laadimine *GPU* mällu on suhteliselt lihtne. Piisab sellest, et need konstandid lihtsalt defineerida Shader failis ja siis väärtustada *cpp* failis. Siis selleks, et saaks kasutada keerulisemaid andmetüüpe nagu *Array*, tuleb luua *DescriptorSet* objekt. Seoses sellega on vajalik ka *DescriptorSetLayout* objekt, mis kirjeldab andmete paigutust.

5. Platvormide võrdlus

Platvormide võrdlus koosneb pilditöötluse probleemi lahenduse ning kasutajatoe võrdlusest. Näiteks lahenduse osas saab võrrelda kui lihtne on ühel platvormil mälu andmete kopeerimine GPU mällu. Kasutajatoe puhul saab võrrelda kui lihtne oli leida platvormi puhul juhendmaterjali või kui palju lisateadmisi oli vaja, et lahendus platvormil tööle saada. Kuna arendus CUDA-l toimus kasutades *runtime API-d*, siis on eelkõige võrreldud pilditöötlus probleemi lahenduse implementatsiooni *runtime API-d* ja *Vulkan*- vahel.

5.1 Kasutajatugi

Vulkan-l kui ka CUDA-l on saadaval repositoorium näidistest[14][15], mis koosneb tüüpilistest probleemidest GPU-l lahendada. Nendega tutvumine on hea viis platvormidega alustamiseks.

Võrreldes Vulkan-ga on CUDA-l ametlik integratsioon erinevate IDE-de. Näiteks saab CUDA-t integreerida *Visual Studio IDE*-ga nii, et läbi integreeritud programmeerimiskeskkonna on näha väga põhjalikku rakenduse mälukasutust. Samast tööriistast on ka käsuliini versioon, mille näidet on näha joonisel 9 Lisaks sellele on läbi selle integratsiooni väga lihtne luua uusi CUDA projekte. Vulkan-i puhul on näiteks vaja teha lisatööd, et siduda IDE SDK-ga või siis kasutada projektide konfigureerimise tööriista nagu *CMake*.

```
OPT   Compute is more heavily utilized than Memory: Look at the Compute Workload Analysis section to see what the
      compute pipelines are spending their time doing. Also, consider whether any computation is redundant and
      could be reduced or moved to look-up tables.

Section: Launch Statistics
-------------------------------- --------------- ---------------
Metric Name                          Metric Unit    Metric Value
-------------------------------- --------------- ---------------
Block Size                                                   256
Function Cache Configuration                    CachePreferNone
Grid Size                                                 37,668
Registers Per Thread             register/thread              50
Shared Memory Configuration Size           Kbyte            8.19
Driver Shared Memory Per Block       Kbyte/block            1.02
Dynamic Shared Memory Per Block       byte/block               0
Static Shared Memory Per Block        byte/block               0
# SMs                                         SM              16
Threads                                   thread       9,643,008
Uses Green Context                                           0
Waves Per SM                                            588.56
-------------------------------- --------------- ---------------
```

Joonis 9. Näide Nvidia Compute tööriista poolt pakutavast informatsioonist.

5.2 Arendus

Mõlema platvormi puhul on suurem osa koodist kirjutatud keeltes C/C++ ning ülejäänud kood C/C++-le sarnases keeles. CUDA puhul on selleks keeleks CUDA-C, millele üle

minek C++ pealt on väga lihtne. Peamiseks erinevuseks on funktsioonide spetsifikaatorid, millega kirjeldada, millisel seadmel kood jookseb. Kuigi platvormil Vulkan toimub koodi kirjutamine samuti keeles C/C++, siis on eraldiseisvad shader failid, mis on nii HLSL kui ka GLSL puhul samuti väga sarnased C/C++-ga. Siiski, eeldab see, et programmeerija oskab seda faili iseseisvalt kompileerida kasutades mõnda shader-i kompileerijat. CUDA puhul on kogu programmi kood ühesugue formaadiga failides ning piisab sellest, et see anda nvcc kompilaatorile kompileerida.

Võrreldes CUDA-ga on Vulkan-s paljude etappide läbiviimine koodi kirjutamise poolest suhteliselt mahuks. Näiteks eelnevalt nimetatud andmete kopeerimine põhimälust GPU mällu. CUDA puhul piisab ainult kahest funktsioonist, mälu eraldamiseks on *CudaMalloc* ning andmete transportimiseks on *CudaMemcpy*. Platvormil Vulkan tuleb sama ülesande täitmiseks luua mälupuhver, küsida rakendusliideselt nõuded mälu jaoks, eraldada mälu, siduda mälu puhvriga ning viimaks kopeerida andmed. See annab küll programmeerijale parema kontrolli mälu kasutuse üle, kuid teisalt teeb see programmi haldamise raskemaks ning jällegi nõuab, et programmeerijal oleks platvormi kasutamisest parem arusaam.

Joonisel 10 on näha kuidas toimub andmete transportimine platvormil CUDA. Piisab ainult mälu eraldamisest ning peale seda on võimalik kohe kopeerida. joonisel 11 on andmete transportimiskes vajalik tegevus platvormil Vulkan. See teeb andmete transportimise isegi väiksemate rakenduste puhul koodi poolest mahukamaks ning raskemini loetavaks.

```
// Allocate Data memory on device
cudaError_t errDataMalloc = cudaMalloc((void**)&d_data, (sizeof(unsigned char) * y * x * CHANNELS));
if (errDataMalloc != cudaSuccess) {
    std::cerr << "CUDA DataMalloc failed: " << cudaGetErrorString(errDataMalloc) << std::endl;
}

// Data to device memory
cudaError_t errDataMemCpy = cudaMemcpy(d_data, data, (sizeof(unsigned char) * y * x * CHANNELS), cudaMemcpyHostToDevice);
if (errDataMemCpy != cudaSuccess) {
    std::cerr << "CUDA DataMemCpy failed: " << cudaGetErrorString(errDataMemCpy) << std::endl;
}

// Allocate c1 memory on device
cudaError_t errC1Malloc = cudaMalloc((void**)&d_c1, (sizeof(float) * (n + 1) * (n + 1)));
if (errC1Malloc != cudaSuccess) {
    std::cerr << "CUDA DataMalloc failed: " << cudaGetErrorString(errDataMalloc) << std::endl;
}

// c1 to device memory
cudaError_t errC1MemCpy = cudaMemcpy(d_c1, c1, (sizeof(float) * (n + 1) * (n + 1)), cudaMemcpyHostToDevice);
if (errC1MemCpy != cudaSuccess) {
    std::cerr << "CUDA DataMemCpy failed: " << cudaGetErrorString(errDataMemCpy) << std::endl;
}

// Allocate out1 memory on device
cudaError_t errOut1Malloc = cudaMalloc((void**)&d_out1, (sizeof(unsigned char) * y * x * CHANNELS));
if (errOut1Malloc != cudaSuccess) {
    std::cerr << "CUDA DataMalloc failed: " << cudaGetErrorString(errDataMalloc) << std::endl;
}

// out1 to device memory
cudaError_t errOut1MemCpy = cudaMemcpy(d_out1, out1, (sizeof(unsigned char) * y * x * CHANNELS), cudaMemcpyHostToDevice);
if (errOut1MemCpy != cudaSuccess) {
    std::cerr << "CUDA DataMemCpy failed: " << cudaGetErrorString(errDataMemCpy) << std::endl;
}
```

Joonis 10. CUDA sammud andmete videomällu transportimiseks.

```
//Buffer creation
vk::Buffer dataBuffer = Device.createBuffer(bufferCreateInfo1);
vk::Buffer outBuffer = Device.createBuffer(bufferCreateInfo2);
vk::Buffer CoeffBuffer = Device.createBuffer(bufferCreateInfoCoeff);
vk::Buffer C4Buffer = Device.createBuffer(bufferCreateInfoC4);

//Buffer memory reqs
vk::MemoryRequirements dataMemReqs = Device.getBufferMemoryRequirements(dataBuffer);
vk::MemoryRequirements outMemReqs = Device.getBufferMemoryRequirements(outBuffer);
vk::MemoryRequirements CoeffMemReqs = Device.getBufferMemoryRequirements(CoeffBuffer);
vk::MemoryRequirements C4MemReqs = Device.getBufferMemoryRequirements(C4Buffer);

if (BufferSize1 > dataMemReqs.size || BufferSize2 > outMemReqs.size || BufferSizeCoeff > CoeffMemReqs.size || BufferSizeC4 > C4MemReqs.size) {
    throw std::runtime_error("Buffer size exceeds allocated memory size!");
}

//Finding memory types
auto findMemoryType = [&](uint32_t typeFilter, vk::MemoryPropertyFlags properties) -> uint32_t {
    vk::PhysicalDeviceMemoryProperties memProps = PhysicalDevice.getMemoryProperties();
    for (uint32_t i = 0; i < memProps.memoryTypeCount; i++) {
        if ((typeFilter & (1 << i)) && (memProps.memoryTypes[i].propertyFlags & properties) == properties) {
            return i;
        }
    }
    throw std::runtime_error("Failed to find suitable memory type!");
    };

//Memory allocation to buffers
vk::MemoryAllocateInfo dataAllocInfo{
    dataMemReqs.size,
    findMemoryType(dataMemReqs.memoryTypeBits, vk::MemoryPropertyFlagBits::eHostVisible | vk::MemoryPropertyFlagBits::eHostCoherent)
};
vk::DeviceMemory dataMemory = Device.allocateMemory(dataAllocInfo);

vk::MemoryAllocateInfo outAllocInfo{
    outMemReqs.size,
    findMemoryType(outMemReqs.memoryTypeBits, vk::MemoryPropertyFlagBits::eHostVisible | vk::MemoryPropertyFlagBits::eHostCoherent)
};
vk::DeviceMemory outMemory = Device.allocateMemory(outAllocInfo);

vk::MemoryAllocateInfo CoeffAllocInfo{
CoeffMemReqs.size,
findMemoryType(CoeffMemReqs.memoryTypeBits, vk::MemoryPropertyFlagBits::eHostVisible | vk::MemoryPropertyFlagBits::eHostCoherent)
};
vk::DeviceMemory CoeffMemory = Device.allocateMemory(CoeffAllocInfo);

vk::MemoryAllocateInfo C4AllocInfo{
C4MemReqs.size,
findMemoryType(C4MemReqs.memoryTypeBits, vk::MemoryPropertyFlagBits::eHostVisible | vk::MemoryPropertyFlagBits::eHostCoherent)
};
vk::DeviceMemory C4Memory = Device.allocateMemory(C4AllocInfo);

//Buffer binding to memory
Device.bindBufferMemory(dataBuffer, dataMemory, 0);
Device.bindBufferMemory(outBuffer, outMemory, 0);
Device.bindBufferMemory(CoeffBuffer, CoeffMemory, 0);
Device.bindBufferMemory(C4Buffer, C4Memory, 0);

//Mapping and copying data to memory
void* mappedData = Device.mapMemory(dataMemory, 0, BufferSize1);
if (!mappedData) {
    throw std::runtime_error("Failed to map data memory!");
}
memcpy(mappedData, data, BufferSize1);
Device.unmapMemory(dataMemory);

void* mappedOut = Device.mapMemory(outMemory, 0, BufferSize2);
if (!mappedOut) {
    throw std::runtime_error("Failed to map zeros memory!");
}
memcpy(mappedOut, out1, BufferSize2);
//Device.unmapMemory(outMemory);

void* mappedCoeff = Device.mapMemory(CoeffMemory, 0, BufferSizeCoeff);
if (!mappedCoeff) {
    throw std::runtime_error("Failed to map Coeff memory!");
}
memcpy(mappedCoeff, c1, BufferSizeCoeff);
//Device.unmapMemory(outMemory);

void* mappedC4 = Device.mapMemory(C4Memory, 0, BufferSizeC4);
if (!mappedC4) {
    throw std::runtime_error("Failed to map Coeff memory!");
}
memcpy(mappedC4, c4, BufferSizeC4);
//Device.unmapMemory(outMemory);

std::cout << "Data, c1, c4 and out1 copied to device memory successfully." << std::endl;
```

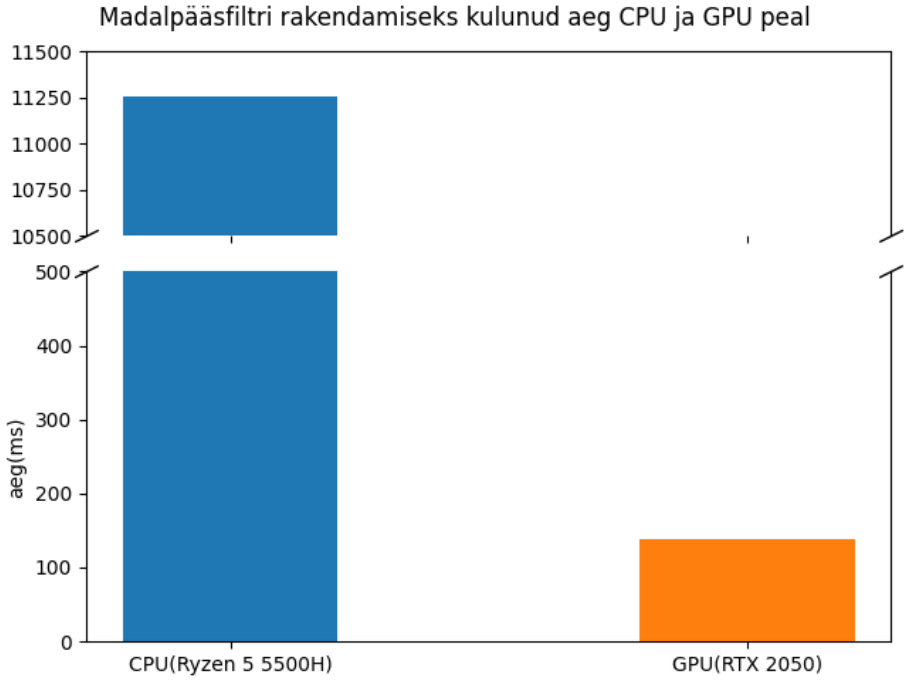
Joonis 11. Vulkan sammud andmete videomällu transportimiseks.

5.3 Jõudlus

Platvorme on võrreldud kolmel erineval Nvidia ja ühel Intel-i graafikakaardil. Võrdlemiseks on kasutatud pilti resolutsiooniga 4128 x 2322. Joonistel, millel ei ole võrreldud erineva suurusega filtreid, on filtri suurus 7x7(koefitsient on 4). Alampeatükkides 5.3.3 ja 5.3.2 on võrdluses mõõdetud aega, mis kulub ainult arvutuste tegemiseks, ehk seal ei kajastu andmete transportimiseks kulunud aeg. Alampeatükis 5.3.4 on mõõdetud aega, mis kulub ainult andmete transportimiseks.

5.3.1 CPU ja GPU jõudluse võrdlus

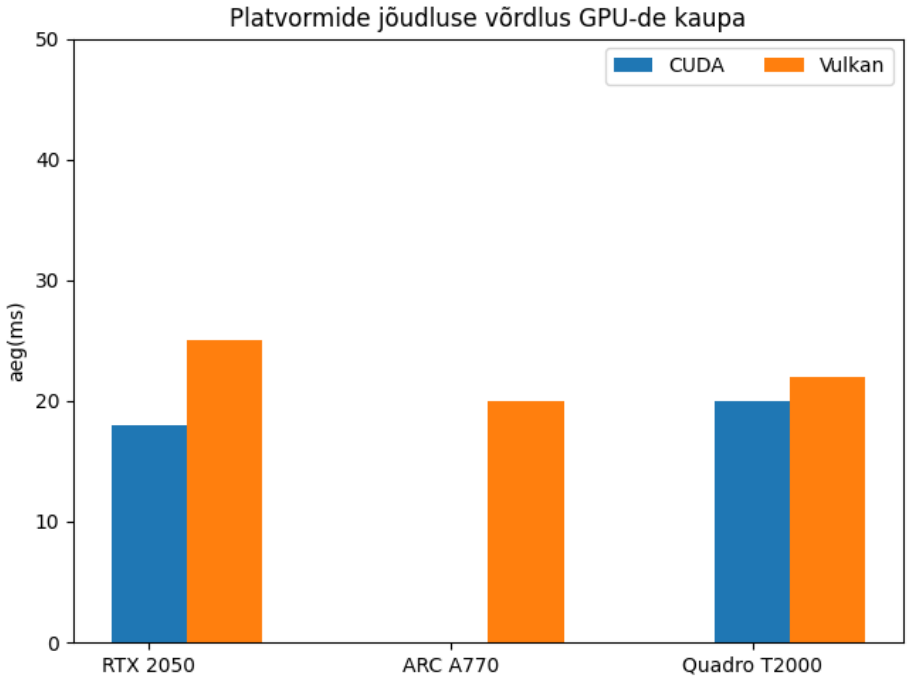
Kuna lõputöö käigus said teostatud FIR filtrid ka CPU peal, siis on välja toodud madalpääs-filtri rakendamiseks pildile kulunud aeg ühe CPU ja GPU peal. Joonisel 12 on näha, et sama filtri rakendamine pildile kasutades GPU-d on ligikaudu 80 korda kiirem kui kasutada CPU-d. Võrdluses välja toodud GPU teostus on tehtud CUDA-l ning sisaldab GPU puhul lisaks arvutuste jaoks kulunud ajale ka andmete transpordiks kulunud aega.



Joonis 12. Madalpääsfiltri rakendamiseks kulunud aeg CPU ja GPU peal.

5.3.2 Platvormide jõudlus erinevatel graafikakaartidel

Joonisel 13 on võrreldud kolme graafikakaarti mõlemal platvormil. RTX 2050 on madala taseme graafikakaart, Quadro on sülearvuti tööjaama kaart ning ARC A770 on Intel-i graafikakaart, mis on neist graafikakaartidest suurema võimsusega, kuid erinevalt T2000-st on A770 suunatud rohkem tavaotstarbeliseks kasutamiseks(näiteks arvutimängud). Kuna CUDA töötab ainult Nvidia graafikakaartidel, siis A770 selle jõudlust võrrelda ei saa.



Joonis 13. Platvormide jõudluse võrdlus GPU-de kaupa.

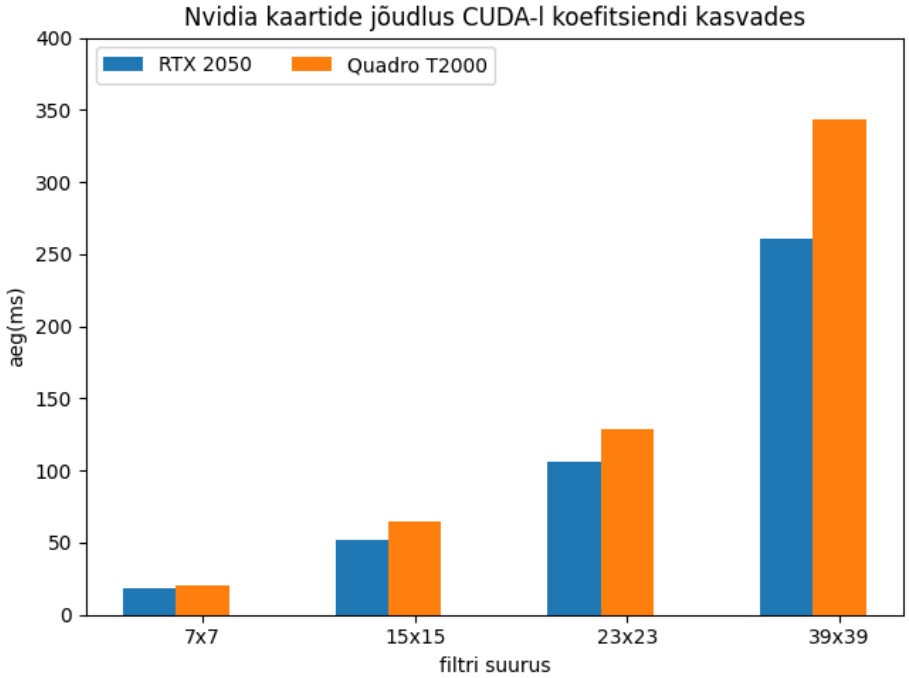
Esialgsete tulemuste põhjal oli RTX 2050 peal CUDA palju aeglasem Vulkan-st. See oli suhteliselt ootamatu tulemus, kuna teiste testitud Nvidia graafikakaartide peal on CUDA kiirem kui Vulkan. Sai tehtud mõned oletused, mis võisid olla sellise tulemuse põhjuseks. Näiteks võimalik, et FIR filtrite implementatsioon ei ole nii effektiivne kui võimalik ning seetõttu jääb RTX 2050 nõrgem võimekus suuremaks takistuseks kui teiste Nvidia graafikakaartide puhul. Hiljem siiski tuli välja selle põhjus ning RTX 2050 osutus isegi Quadro T2000-st kiiremaks(joonis 13).

5.3.3 Jõudlus erinevate suurustega filtritel

Filtri suurus mõjutab oluliselt arvutuste arvu, mida GPU peab tegema, sest iga filtrisse jääv piksel mõjutab filtri keskpunktis oleva piksli väärtust. Kui filter muutub suuremaks, siis on rohkem piksleid mis mõjutavad keskpunkt piksli väärstu, seega rohkem tööd GPU-le. Seega on võrreldud ka GPU-de jõudlust erinevatel platvormidel FIR filtri suuruse kasvades.

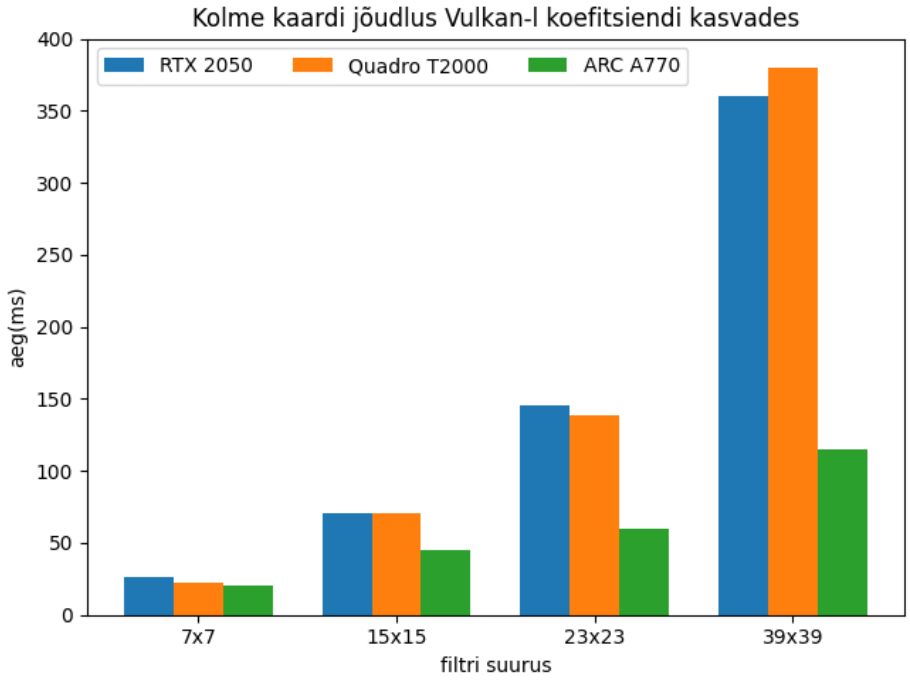
Kuigi Quadro T2000 on tööjama GPU ja RTX 2050 on tavaotstarbeline GPU, siis teoorias peaksid nad olema jõudluse poolest suhteliselt sarnased. Neil on mõlemal 4GB videomälu, kuid RTX 2050 mälu on kiirem (GDDR6 vs GDDR5 T2000 puhul). RTX 2050 on rohkem CUDA tuumasid, kui T2000-l, mis peaks RTX 2050-le andma eelise suure koguste arvutuste tegemisel. Nagu näha joonisel 14, väiksema filtri korral on kiirused peaaegu samad, aga filtri suuruse kasvades suureneb ka arvutuste arv ning kasvab RTX 2050 eelis T2000 ees.

Eelmises peatükis sai mainitud, et algsete tulemuste põhjal oliplatvormil CUDA RTX 2050 väga palju aeglasem T2000-st. Näiteks 7x7 suurusega filtri korral kulus RTX 2050-l 152ms arvutuste tegemiseks, kuid T2000-s ainult 26ms. Sellise tulemuse põhjuseks oli see, et RTX 2050 arvutil testiti kernelit kasutades Visual Studio IDE-d ning mingisugusel põhjusel tegi see kerneli palju aeglasemaks. Kui testida täpselt samasugust kernelit kasutades kompileerimiseks käsurida, siis olid tulemused palju loogilisemad.



Joonis 14. Nvidia kaardid platvormil CUDA koefitsiendi kaupa.

Joonisel 15 on samasugune võrdlus platvormil Vulkan. Siin on näha, et RTX 2050 ja Quadro T2000 tööaeg on väga sarnane, mis on oodatud tulemus kuna need kaks GPU-d on riistvaravõimekuse poolest suhteliselt sarnased. A770-l on väikese filtri korral minimaalne jõudluse eelis, kuid filtri suuruse kasvades kasvab ka A770 eelis. See on samuti loogiline kuna A770 on teistest võrreldud kaartidest palju võimsam.

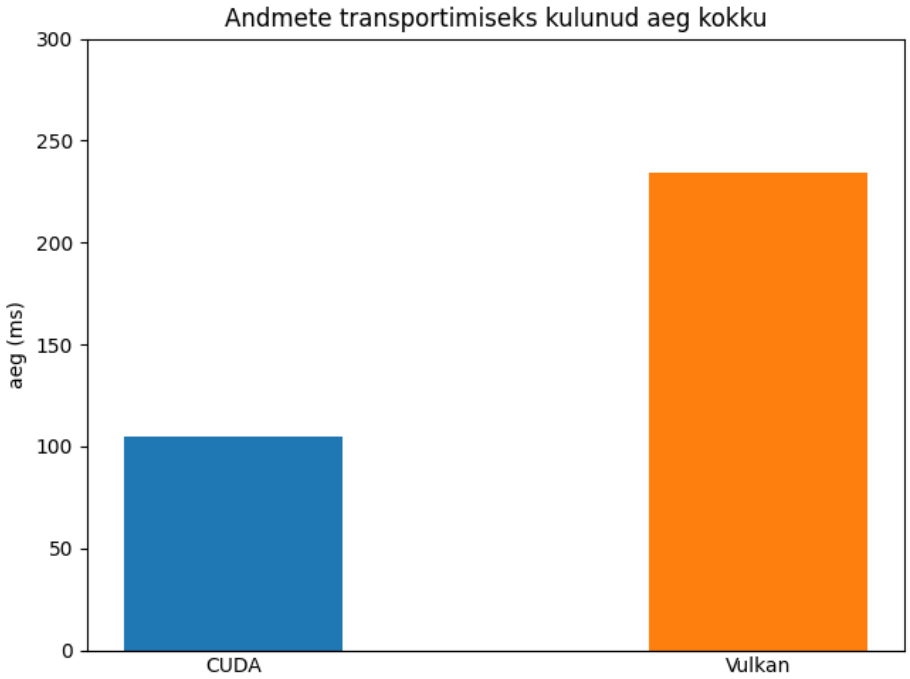


Joonis 15. Kaardid Vulkan-l koefitsiendi kaupa.

5.3.4 Jõudlus andmete transportimisel

Joonis 16 näitab kogu aega, mis on kulunud andmete transportimiseks põhimälust videomällu ja tagasi. See aeg sõltub pigem pildi suurusest, mitte arvutustest, seega filtri suuruse muutudes ei muutu transportimiseks kulunud aeg. Samuti mängib siin suurt rolli peale GPU-le ka CPU ja põhimälu kiirus. Joonisel olevad andmed on saadud arvutil, millel on GPU RTX 2050, CPU R5 5500H ja põhimälu kiirusega 3200 MHz.

Veel oli tulemustest näha, et platvormil CUDA esialgne andmete transport põhimälust videomällu kiirem kui platvormil Vulkan. Vulkan-l hoopis vastupidi, andmete transport peale töötlemist videomälust tagasi põhimällu palju aeglasem. Seda on ka näha järgmise peatüki tabelites 3 ja 4. Kokkuvõttes ei saa väga põhjapanevaid järeldusi andmete transprortimise kiirusest teha, kuna seda oleks vajalik testida mitmete erinevate CPU, RAM, GPU ja VRAM kombinatsioonidega.



Joonis 16. Kogu kulunud aeg andmete transportimiseks ühel testitud arvutil.

5.3.5 Kõik testitulemused

Kõik kerneli arvutusajad millisekundites on tabelites 1 ja 2. Samuti on nendes tabelites madalpääsfiltri rakendamiseks kulunud aeg filtri suuruste 27x27, 31x31 ja 35x35 korral. Tabelites 3 ja 4 on andmete transportimiseks kulunud ajad. Lisaks RTX 2050, Quadro T2000 ja Arc A770-le on tabelites toodud välja testitulemused RTX 3090 peal, mis on kõikidest testitud graafikakaartidest kõige suurema võimsusega.

Tulemustest on näha, et üldjoontes on CUDA kiirem kui Vulkan nii arvutuste tegemises kui ka andmete transportimises. Intel-i GPU puhul võtab mälu kaardistamise samm märgatavalt aega, kuid Nvidia graafikakaartide puhul on selle sammu aeg praktiliselt 0 millisekundit. Platovrmil Vulkan on veel samme mis tuleb läbida andmete transportmiseks, kui need on samuti igal testitud platvormil <0.1ms, seega neid tabelis välja ei ole toodud. Peab ka meeles pidama, et andmete transportimise puhul mängib suurt rolli ka CPU ja põhimälu kiirus, mitte ainult GPU.

Tabel 1. CUDA testitulemused

CUDA			
Filtri suurus	RTX 2050	Quadro T2000	RTX 3090
7x7(N = 4)	18	20	3
15x15(N = 8)	52	65	9
23x23(N = 12)	106	129	18
27x27(N = 14)	140	172	24
31x31(N = 16)	178	209	30
35x35(N = 18)	221	278	38
39x39(N = 20)	261	343	47

Tabel 2. Vulkan testitulemused

Vulkan				
Filtri suurus	RTX 2050	Quadro T2000	RTX 3090	Arc A770
7x7(N = 4)	26	22	23	20
15x15(N = 8)	70	70	27	45
23x23(N = 12)	145	138	27	60
27x27(N = 14)	193	188	34	73
31x31(N = 16)	246	243	44	83
35x35(N = 18)	302	307	54	97
39x39(N = 20)	360	380	66	115

Tabel 3. CUDA andmete transpordi testitulemused

CUDA			
Transpordi samm	RTX 2050 R5 5500H	Quadro T2000 i7-9850H	RTX 3090 i9-10850K
GPU mälu eraldamine	78	36	120
andmed GPU mällu	16	12	8
tagasi põhimällu	11	6	4

Tabel 4. Vulkan andmete transpordi testitulemused

Vulkan				
Transpordi samm	RTX 2050 R5 5500H	Quadro T2000 i7-9850H	RTX 3090 i9-10850K	Arc A770 R7 3700X
Puhvermälu eraldamine	5	16	16	5
mälu kaardistamine	0	0	0	4
andmed GPU mällu	9	6	7	5
tagasi põhimällu	220	122	136	190

6. Kokkuvõte

Lõputöö eesmärgiks oli luua võrdlus GPU paralleelprogrammeerimise platvormidest CUDA ja Vulkan. Võrdluse eesmärk on anda lugejale arusaam, kuidas mõlemal platvormil rakenduse arendus toimub, millised sammud tuleb rakenduse arendamisel läbida, millised abitööristad ja materjalid on mõlemal platvormil abiks ning viimaks kui effektiivselt mõlemad platvormid on probleemi lahendamisel.

Lõputöös on antud lühitutvustus klassikalisest programmeerimisest ja paralleelprogrammeerimisest. Kirjeldatud parallelprogrammeerimist graafikakaartidel ning antud lühike ülevaade mõndadest eksisteerivated platvormidest. Täpsem ülevaade on antud võrreldatavates platvormidest. On kirjeldatud kuidas nad töötavad ning mis tuleb teha, et saaks alustada nende kasutamist.

Platvormide võrdluse võimaldamiseks on implementeeritud pilditöötlusprobleemi lahendus, mis kasutab kahte FIR filtrit madalpääsfilter ja kõrgpääsfilter, et pildilt müra eemaldada ning pilti teravdada. Kõigepealt on see teostatud kasutades CPU-d ning hiljem mõlemal võrreldavalt platvormil kasutades GPU-d.

Platvorme sai võrreldud kasutajatoe, arenduse ning jõudluse poole pealt. Kasutajatoe osas leidub mõlemal platvormil küllaltki palju erinevat materjali, kuid abistavate tööriistade poolest on CUDA-l eelis. Arenduse külje pealt tuli välja, et Vulkan-i rakendused on koodi poolest CUDA rakendustest mahukamad. Viimaks on võrreldud platvormide jõudlust neljal erineval graafikakaardil nii arvutuste tegemise kiiruse kui ka andmete transportimise kiiruse poole pealt. Andmetest on näha, et üldjuhul on CUDA kiirem kui Vulkan.

Kasutatud kirjandus

[1] *Python Programming And Numerical Methods: A Guide For Engineers And Scientists*. [Kasutatud: 03-05-2025]. URL: https://pythonnumericalmethods.studentorg.berkeley.edu/notebooks/chapter13.01-Parallel-Computing-Basics.html.

[2] Yury Lebedev. *Race Condition: Analysis and Prevention Methods*. [Kasutatud: 06-12-2024]. URL: https://medium.com/@quasaryy/race-condition-analysis-and-prevention-methods-0c0b434ea332.

[3] Somnath Musib. *Deadlock, Livelock and Starvation*. [Kasutatud: 06-12-2024]. URL: https://www.baeldung.com/cs/deadlock-livelock-starvation.

[4] *Using your GPU with CuPy*. [Kasutatud: 08-12-2024]. URL: https://carpentries-incubator.github.io/lesson-gpu-programming/cupy.html.

[5] *The GPU Devotes More Transistors to Data Processing*. [Kasutatud: 07-12-2024]. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model.

[6] [Kasutatud: 10-12-2024]. URL: https://www.khronos.org/.

[7] Nicholas Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. [Kasutatud: 19-05-2024].

[8] *ImageMagick software suite*. [Kasutatud: 28-11-2024]. URL: https://imagemagick.org/index.php.

[9] *LodePNG PNG image decoder and encoder*. [Kasutatud: 28-11-2024]. URL: https://lodev.org/lodepng/.

[10] *stb libraries*. [Kasutatud: 28-11-2024]. URL: https://github.com/nothings/stb.

[11] *FIR Filtering and Image Processing*. [Kasutatud: 14-05-2025]. URL: https://www.eecs.umich.edu/courses/eecs206/archive/spring02/lab.dir/Lab6/lab6_v3_0_release.pdf.

[12] *A Class of Fast Gaussian Binomial Filters for Speech and Image Processing*. [Kasutatud: 19-05-2024]. URL: https://web.njit.edu/~akansu/PAPERS/Haddad-AkansuFastGaussianBinomialFiltersIEEE-TSP-March1991.pdf.

[13] *HLSL in Vulkan*. [Kasutatud: 31-12-2024]. URL: https://docs.vulkan.org/guide/latest/hlsl.html.

[14] *CUDA Samples*. [Kasutatud: 31-12-2024]. URL: https://github.com/NVIDIA/cuda-samples.

[15] *Vulkan Samples*. [Kasutatud: 31-12-2024]. URL: https://github.com/KhronosGroup/Vulkan-Samples.

Lisa 1– Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks[1]

Mina, Asko Vendel

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose "GPU paralleelprogrammeerimisplatvormide CUDA ja Vulkan võrdlus", supervised by Lembit Jürimägi and Gert Kanter
   1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
   1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.
3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

04.06.2025

[1]Lihtlitsents ei kehti juurdepääsupiirangu kehtivuse ajal vastavalt üliõpilase taotlusele lõputööle juurdepääsupiirangu kehtestamiseks, mis on allkirjastatud teaduskonna dekaani poolt, välja arvatud ülikooli õigus lõputööd reprodutseerida üksnes säilitamise eesmärgil. Kui lõputöö on loonud kaks või enam isikut oma ühise loomingulise tegevusega ning lõputöö kaas- või ühisautor(id) ei ole andnud lõputööd kaitsvale üliõpilasele kindlaksmääratud tähtajaks nõusolekut lõputöö reprodutseerimiseks ja avalikustamiseks vastavalt lihtlitsentsi punktidele 1.1. ja 1.2, siis lihtlitsents nimetatud tähtaja jooksul ei kehti.

Lisa 2 - Lähtekood

Kogu lõputöö lähtekood on kättesaadaval GitHub-i repositooriumis:
`https://github.com/AskoVendel/cuda-vulkan-comparison`