TARTU ÜLIKOOL
Matemaatika-informaatikateaduskond
Arvutiteaduse instituut

Holger Biene

Graafikaprotsessorite kasutamine teadusarvutuste kiirendamisel

Bakalaureusetöö (4AP)

Juhendajad: Eero Vainikko
Mark Tehver

Autor: …………………………………… “…..“ mai 2006

Juhendaja: ………………………………. “…..“ mai 2006

Tartu 2006

Sisukord

1. Sissejuhatus 3
2. Mõistete definitsioonid 5
3. Graafikaprotsessorite arhitektuur 8
   3.1 Graafikaprotsessor ja keskprotsessor 8
   3.2 Graafikaprotsessori arhitektuur ja graafikakonveier 8
4. Arvutusvõimsuse suurendamine 12
   4.1 CPU ja GPU vahelised erinevused 13
5. Voogarvutus 14
   5.1 Voogprogrammeerimise mudel 14
6. Nõuded ja piirangud graafikaprotsessoril töötavatele arvutustele 17
   6.1 Graafikaprotsessoris sisalduvad arvutusressursid 18
7. Keskprotsessori ja graafikaprotsessori programmeerimise analoogiad 20
8. Andmestruktuurid 23
9. Programmeerimiskeeled ja –vahendid 25
   9.1 Cg 25
   9.2 HLSL 26
   9.3 OpenGL Shading Language 26
   9.4 BrookGPU 27
   9.5 Glift 27
Kokkuvõte 29
Usage of graphics processors for speeding up scientific computation 30
Kasutatud kirjandus 31
Lisad 33
   Näidisprogramm 33

1. Sissejuhatus

Arvutiprotsessorite arvutusvõimsuse kiires kasvus on siiani suurimat rolli mänginud protsessorite taktsageduse kiire tõus. Aastal 1999 oli Intel'i protsessorite suurimaks taktsageduseks 500 MHz, 2001. aastal 2 GHz ja 2003. aastal 3,4 GHz. Kuid pärast 2003. aastat on taktsageduse kasvu tempo aeglustunud, 2005. aastal oli suurim taktsagedus 3,8 GHz, taktsageduse edasist suurendamist piiravad üha kasvav protsessori töö käigus eralduv suur soojushulk, tarbitava elektrienergia hulk ning lekkevoolud. Uueks märksõnaks arvutusvõimsuse suurendamisel on saanud paralleelne arvutamine. Protsessori taktsageduse tõstmise asemel keskendutakse nüüd mitmetuumaliste protsessorite arendamisele.

Kui hetkel on tavaarvutitesse jõudnud kahetuumalised protsessorid, siis graafikaprotsessorite näol on tavaarvutid sisaldanud mitmetuumalisi protsessoreid juba pikemat aega. Graafikaprotsessorite põhiülesanne, suure arvu matemaatiliste tehete teostamine ekraanipildi loomiseks, on suures osas paralleliseeritav. Seepärast algas graafikaprotsessorite areng paralleelsuse suunas varem ning sellega on jõutud kaugemale võrreldes arvuti keskprotsessoritega. Uusimad graafikaprotsessorid koosnevad kuni 24 paralleelselt töötavast protsessorist, ületades keskprotsessori arvutusvõimsust (umbes 10 miljardit ujukomatehet sekundis) suurusjärgu võrra (200 miljardit ja enam ujukomatehet sekundis).

Antud bakalaureusetöö eesmärgiks on tutvustada graafikaprotsessorit ja tema potentsiaali keskprotsessori kõrval rakendamiseks mahukate arvutusülesannete lahendamisel. Bakalaureusetöö koosneb üheksast peatükist. Sissejuhatusele järgnev teine peatükk koosneb käesolevas töös kasutatud mõistete ja lühendite definitsioonidest. Kolmas peatükk selgitab graafikaprotsessori ja graafikakonveieri ülesehitust Nvidia GeForce 6800 seeria graafikaprotsessorite alusel. Neljas peatükk kirjeldab riistvaralise disaini eesmärke arvutusvõimsuse suurendamisel ja selgitab keskprotsessori ning graafikaprotsessori disainide erinevusi. Viies peatükk tutvustab voogarvutust ja näitab voogprogrammeerimise mudeli kokkusobivust arvutusvõimsuse suurendamise eesmärkidega.

Kuuendas peatükis on kirjeldatud piiranguid ja nõudeid, mis tekivad graafikaprotsessori kasutamisel arvutusülesannete lahendamiseks. Seitsmes peatükk toob välja kesk–protsessori ja graafikaprotsessori programmeerimisel esinevaid sarnasusi. Kahek–sandas peatükis on kirjeldatud graafikaprotsessorite andmestruktuure koos nende piirangutega ning on toodud näiteid andmestruktuuride teisendamisest graafika–protsessori jaoks sobilikku vormi. Üheksas peatükk pakub ülevaadet eksisteerivatest graafikaprotsessorite kõrgtaseme programmeerimiskeeltest ja –vahenditest. Bakalau–reusetöö lõppu on lisatud graafikaprotsessori näidisprogramm.

2. Mõistete definitsioonid

Antud peatükis on toodud bakalaureusetöös kasutatud lühendite ja mõistete definitsioonid.

Alfasujutamine – *alpha blending*, alfasujutamise käigus kombineeritakse läbipaistvale objektile kuuluvad pikslid läbipaistva objekti taga asuvate objektide pikslitega selliselt, et lõplikult visualiseeritud kaadris on läbipaistev objekt läbinähtav [1]. Näiteks klaasaken või vesi.

CPU – *Central Processing Unit*, arvuti keskprotsessor.

Diskreetimine – *rasterization*, protsess, mille käigus primitiivide andmete põhjal koostatakse fragmentide massiiv. Esiteks luuakse visualiseeritava kaadri mõõtmete suurune võre, seejärel määratakse kindlaks, milliseid võre punkte primitiivid katavad. Tipuandmete põhjal määratakse primitiivide poolt kaetud võre punktide värvus, läbipaistvus ja sügavus [1].

Fragment – diskreetimise käigus tekkiv visualiseeritava kaadri osake. Kannab endas andmeid värvuse, läbipaistvuse ja sügavuse kohta. Graafikakonveieri edukal läbimisel, kui fragmenti ei eemaldata teostatavate testide käigus fragmentide massiivist, kirjutatakse fragmendi andmed kaadripuhvrisse. Fragment on potentsiaalne tulevane piksel kaadripuhvris [1].

GPGPU – *General-Purpose Computing on Graphics Processing Units*, graafika–protsessorite kasutamine arvutuste tegemiseks keskprotsessori asemel.

GPU - *Graphics Processor Unit*, graafikaprotsessor.

Kaadripuhver – graafikaprotsessori mälupiirkond, milles hoitakse täielikult visualiseeritud kaadrit [1].

Kernel – kernel sisaldab endas arvutusoperatsioone, mida rakendatakse kõigile andmevoo elementidele.

Malli operatsioonid – *stencil operations*, malli operatsioonid on sarnased pikslite sügavuste võrdlemisega. Iga piksli malli väärtust võrreldakse etteantud testväärtusega, kui piksli malli väärtus ei ole sobiv, praagitakse piksel välja ning teda ei visualiseerita. Pikslite malli väärtused asuvad malli puhvris. Malli operatsioone kasutatakse pikslite valikuliseks visualiseerimiseks. Näide tavaelust: pihustades värvi pinnale kinnitatud ringikujulise auguga šabloonile, jääb pärast šablooni eemaldamist pinnale ringikujuline värvilaik, ülejäänud värv jäi šablooni peale ning pinnale ei jõudnud. Etteantud malli testväärtused toimivad šabloonina, lastes läbi vaid soovitud väärtusi.

MIMD – *Multiple Instruction, Multiple Data*, arvutiarhitektuur, kus mitu arvutusüksust töötlevad andmeid üksteisest sõltumatult ja ei pruugi teostada samaaegselt samu arvutusoperatsioone [2].

Octree – puukujuline andmestruktuur kolmemõõtmelise ruumi puhul kasutamiseks, vaata *quadtree*. Igal tipul on kaheksa alluvat.

Primitiiv – objekti nimetus arvutigraafikas, primitiiv võib olla punkt, joon, kolmnurk või nelinurk. Primitiivid tekivad tipuandmete töötlemisel [1].

Quad – neljast tipust koosnev primitiiv.

Quadtree – puukujuline andmestruktuur, kus iga tipp esitab kahemõõtmelise ruumi alamruumi. Igal tipul võib olla kuni 4 otsest alluvat.

SIMD – *Single-Instruction, Multiple-Data*, arvutiarhitektuur, kus arvutusüksused rakendavad sama arvutusoperatsiooni üheaegselt hulgale erinevatele andmetele [2].

Sujutamine – *blending*, mitme väärtuse, näiteks pikslite värvide, kombineerimine üheks uueks väärtuseks. Kasutatakse näiteks objektide läbipaistvuse saavutamiseks või valgusefektide tekitamisel [1].

Z-puhver – graafikaprotsessori mälupiirkond, kus hoitakse informatsiooni iga piksli ruumilise sügavuse kohta. Ruumilise sügavuse informatsiooni kasutatakse juhtudel, kui ühes ning samas kohas visualiseeritavas kaadris asuvad kahe või rohkema objekti pikslid. Sellisel juhul võrreldakse pikslite sügavuste väärtusi ning valitakse väiksema sügavusega piksel. Valitud piksli sügavuse väärtus kirjutatakse Z-puhvrisse. Z-puhvri

abil loob graafikaprotsessor ruumilise, sügavustajuga pildi: lähemal asuvad objektid katavad kaugemal asuvaid objekte [1].

Tipp – *vertex*, kolmemõõtmelise graafika baaselement. Tipp kirjeldab punkti ruumis, tipu kirjeldus sisaldab alati punkti ruumilisi koordinaate (x, y, z), lisaks sõltuvalt rakendusest seotakse tipuga ka lisaatribuudid nagu värv ja normaalvektor [3].

Visualiseerimine – *rendering*, graafikaprotsessorile edastatud andmete põhjal arvutiekraanile väljundpildi loomine [2].

3. Graafikaprotsessorite arhitektuur

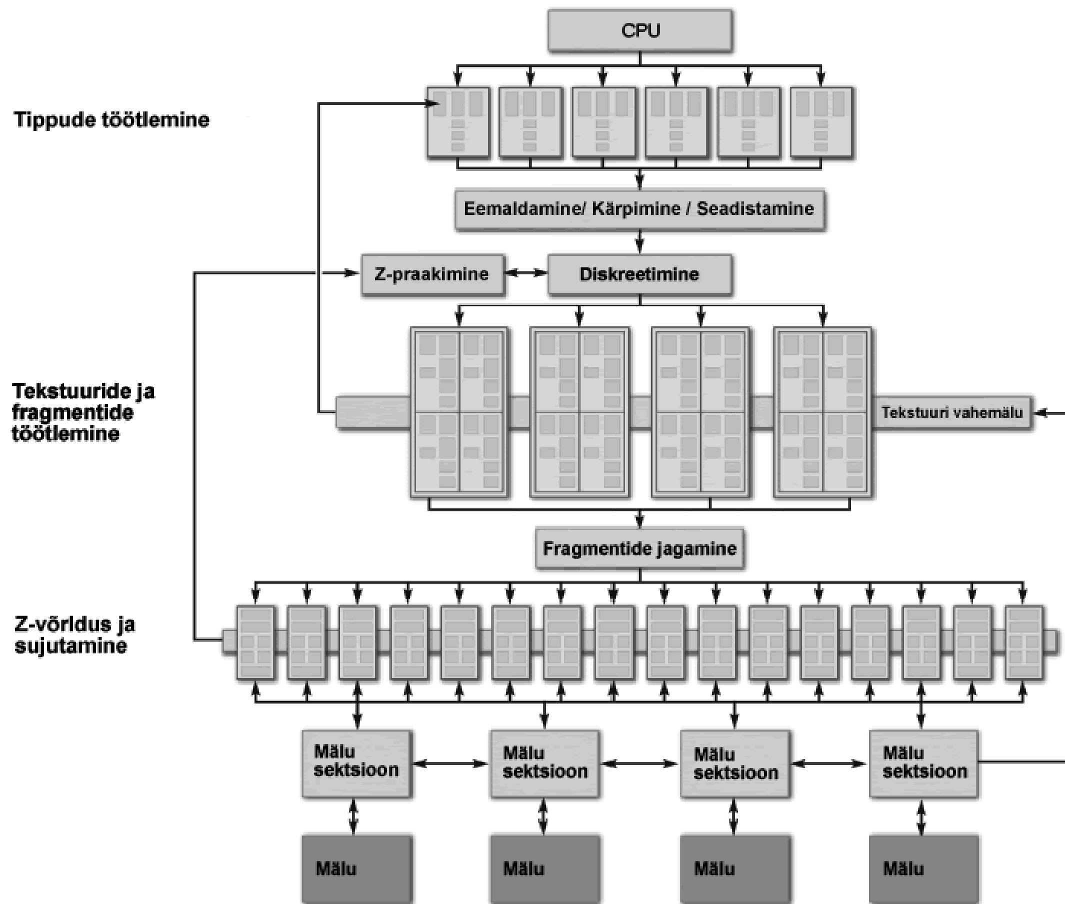
Alljärgnevas peatükis on käsitletud graafikaprotsessorite arhitektuuri Nvidia GeForce 6800 seeria graafikaprotsessorite alusel. Peatüki koostamisel on kasutatud [4] materjale.

3.1 Graafikaprotsessor ja keskprotsessor

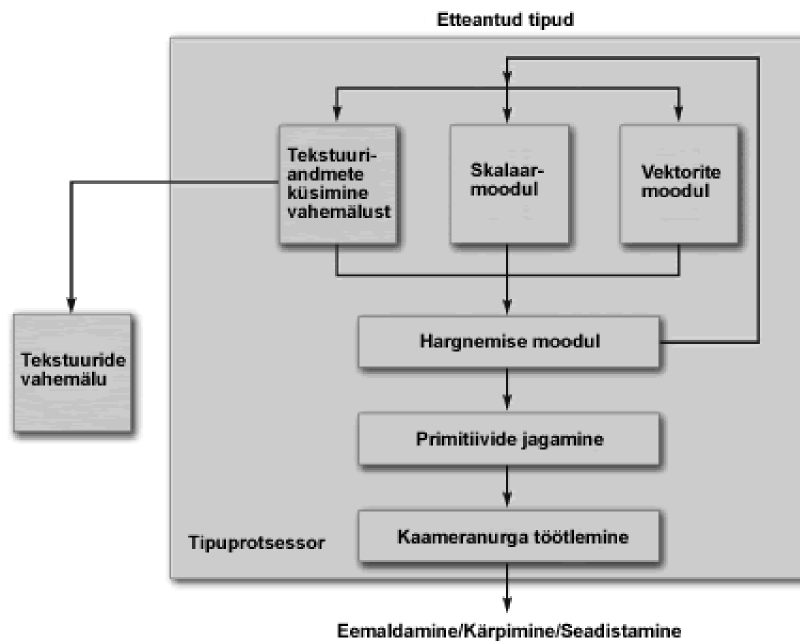
Graafikaprotsessor ja arvuti keskprotsessor suhtlevad omavahel läbi graafikasiini (AGP siin või PCI Express). Graafikasiini kaudu toimub käskude, tekstuuride ning tippude andmete edastamine keskprotsessorilt graafikaprotsessorile. Graafikaliidese maksi–maalne ribalaius on aastate jooksul kasvanud esialgsest 264 megabaiti sekundis 2,1 gigabaidini sekundis AGP siini puhul ning on 4 GB/s mõlemas suunas PCI Express siini puhul. GPU mäluliidese ribalaiuseks on 35 GB/s. CPU enda mäluliidese ribalaius 800 megahertsise taktsagedusega esisiini puhul on 6,4 GB/s. CPU ja GPU mäluliideste erinev ribalaius on üks põhjusi, miks GPU arvutusmahukate ülesannete puhul CPUga võrreldes kiirem on.

3.2 Graafikaprotsessori arhitektuur ja graafikakonveier

Joonisel 1 on näha GeForce 6 seeria GPU arhitektuuri blokkskeem. Graafikakonveieri töö algab käskude, tekstuuride ning tippude andmete saamisega CPU käest kas läbi jagatud puhvrite arvuti põhimälust või GPU kaadripuhvri mälust. Järgmisena töötleb tipuprotsessor (joonis 2) vastavalt saadud käskudele etteantud tippe, muutes tippude ruumilisi koordinaate, tekstuurikoordinaate ning värve. Tipuprotsessoril on juurdepääs ka tekstuuride andmetele. Seejärel jagatakse tipud primitiivideks – punktid, jooned ja kolmnurgad. Iga primitiivi töödeldakse eraldi: eemaldatakse primitiivid, mis pole nähtavad; kärbitakse primitiivide osi, mis jäävad vaateväljast välja; valmistatakse ette joonte ning tasandite andmed diskreetimise jaoks.

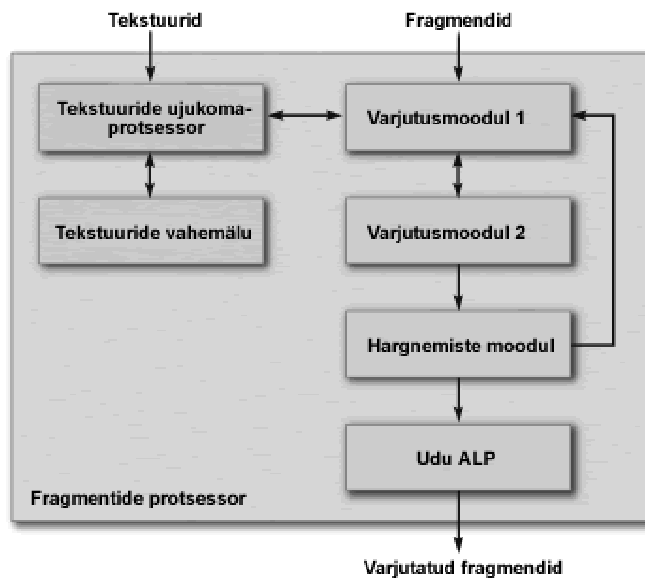


Joonis 1. GeForce 6800 seeria graafikaprotsessori arhitektuuri blokkskeem [4]



Joonis 2. Tipuprotsessori blokkskeem [4]

Diskreetimise etapis arvutatakse, millised pikslid on primitiivide poolt kaetud. Z-praakimise abil eemaldatakse need pikslid, mis jäävad vaateväljas lähemal asuvate objektide pikslite taha varju. Tulemuseks on hulk fragmente ehk võimalikke tulevasi piksleid, mis pärast fragmentide protsessori (joonis 3) ning vastavate testide edukat läbimist kannavad värvi ja sügavuse informatsiooni kaadripuhvris asuvasse pikslisse.



Joonis 3. Fragmentide protsessor [4]

Tekstuuride ning fragmentide töötlemise etapis rakendatakse varjutamise programmi igale fragmendile eraldi. Fragmentide protsessor töötleb neljast fragmendist koostatud ruudukujulisi üksusi ning töötleb sadu selliselt grupeeritud fragmente üheaegselt, rakendades sama käsku paljudele fragmentidele. Fragmentide protsessor kasutab mälust andmete saamiseks vastavat tekstuurimoodulit (*texture unit*). Tekstuur kujutab endast ühe-, kahe- või kolmemõõtmelist andmemassiivi, millest tekstuurimoodul võib lugeda andmeid suvalistest kohtadest. Fragmentide protsessoril on kaks varjutusmoodulit (*shader unit*) iga konveieri kohta, fragmendid läbivad mõlema varjutusmooduli ning hargnemiste mooduli (*branch processor*) ja sisenevad uuesti esimesse varjutus–moodulisse edasiseks töötlemiseks varjutusprogrammi poolt. Selline ringkäik juhtub korra graafikaprotsessori kellatakti jooksul. Iga kellatakti korral on võimalik sooritada kaheksa või enam matemaatilist tehet, või neli tehet, kui esimeses varjutusmoodulis loetakse tekstuurist andmeid. Enne fragmentide protsessorist väljumist asub udu aritmeetika-loogikaplokk, mille abil on võimalik sujutada fragmenti udu efekti loomiseks jõudluses kaotamata.

Fragmendid väljuvad fragmentide protsessorist samas järjekorras, nagu nad tekkisid diskreetimise etapis. Fragmendid saadetakse edasi Z-võrdluse ning sujutamise moodulitesse, kus toimub sügavuse testimine (Z-võrdlus), malli operatsioonid, alfa–sujutamine ja lõpuks värviväärtuste kirjutamine kaardipuhvrisse.

Mälusüsteem on jagatud üksteisest sõltumatuteks mälu sektsioonideks, millest igaüks kontrollib talle eraldatud dünaamilist muutmälu (*DRAM*). Väiksemad ning eraldiseisvad mälu sektsioonid võimaldavad mälusüsteemil töötada efektiivselt nii suurte kui ka väikeste andmeblokkide edastamisel.

4. Arvutusvõimsuse suurendamine

Iga uus riistvarapõlvkond on eelmisest võimekam, ühele ja samale pindalale on võimalik mahutada üha rohkem transistore. Et edasiarengust riistvaras saaks edasiareng arvutusvõimsuses, tuleb silmas pidada kahte aspekti: arvutamisel kasutatavad ressursid peavad olema kasutatavate rakenduste jaoks sobivalt organiseeritud ja suhtlus ning andmeedastus peab ressurside vahel toimuma piisavalt efektiivselt, et arvutustööks tarvitatavate andmete hilinemise tõttu arvutusressursid jõude ei seisaks.

Protsessoris kasutatavad transistorid jagunevad tööülesannetelt kolme rühma: arvutusprotsessi kontrollimiseks kasutatavad transistorid, arvutusoperatsioone teostavad transistorid ja andmete hoidmiseks kasutatavad transistorid [4]. Suure arvutusvõimuse saavutamiseks on otstarbekas kasutada võimalikult suurt arvu transistore otseselt arvutamisega tegelevas allüksuses, kasutada mitut arvutamisega tegelevat üksust samaaegselt ning tagada üksuste maksimaalselt efektiivne töö.

Kasutades mitut arvutamisega tegelevat üksust samaaegselt, on võimalik ära kasutada kolme tüüpi paralleelsust: ülesannete paralleelsust, sooritades erinevaid arvutus–operatsioone samaaegselt; andmete paralleelsust, rakendades ühte ja sama operatsiooni paljudele andmetele samaaegselt; käskude paralleelsust, sooritades mitmeid lihtsaid operatsioone samaaegselt.

Protsessorisisese suhtluse kiirus on kasvanud kiiremini, kui suhtlus väljaspool protsessorit asuvate ressurside, mälu ja kettaseadmetele talletatud andmetega. Et iga pöördumisega väljaspool protsessorit asuvate andmete poole kaasneb langus arvutusvõimsuses, tuleb selliste pöördumiste arv viia miinimumini. Üks võimalus seda teha on sooritada võimalikult palju operatsioone protsessoris leiduvate lokaalsete andmetega. Teine võimalus on säilitada hiljuti kasutatud andmeid protsessori vahemälus nende kiiremaks kättesaamiseks, kui tekib vajadus neid taas kasutada – vahemälu poolt hõivatud transistoride abil suurendatakse mälu ribalaiust. Kolmas võimalus on väliste ressursidega andmevahetusel kasutada tihendatud andmeid, suurendades tihendatud andmete töötlemiseks kasutatavate transistoride abil mälu ribalaiust.

4.1 CPU ja GPU vahelised erinevused

CPU programmid ei kasuta üldjuhul paralleelsust, nõuavad CPUlt võimalusi keerukate kontrolloperatsioonide teostamiseks ning üldjuhul ei vaja suurt arvutusvõimsust. Seega ei ole CPU arhitektuuri disanimisel keskendutud andmete paralleelsusele, CPU töötleb andmeid ükshaaval. CPU puhul on küll võimalik ära kasutada käskude paralleelsust SIMD käsustike (MMX, SSE, SSE2) abil, kuid GPU arhitektuuris on paralleelsuse poolt pakutavate võimalustega palju rohkem arvestatud.

Üks CPU vähese paralleelsuse põhjusi on vajadus keerukate kontrolloperatsioonide järele, suur osa CPU transistoridest on just selle otstarbega, tegeledes näiteks programmikoodi hargnevuste ennustamise või ümberjärjestatult (*out of order*) käskude täitmisega. Otseselt arvutusoperatsioone teostavate transistoride arv võrreldes GPUga on CPU puhul palju väiksem.

GPU on mõeldud spetsiifiliste graafikaarvutuste tegemiseks, seepärast on GPU puhul võimalik kasutada spetsiaalselt selle alamülesande arvutamiseks mõeldud riistvaralisi lahendusi, mis võimaldab suuremat efektiivsust arvutusülesande läbiviimisel. CPU seevastu peab teostama erinevate iseloomudega arvutustehteid ning ei saa spetsiaalsetest riistvaralistest lahendustest kasu.

CPU mälusüsteem on disainitud silmas pidades võimalikul väikest viivitust andmete kättesaamisel mälust ja tagastamisel mällu. Paralleelsuse puudumise tõttu peab andmevahetus mäluga toimuma võimalikult kiiresti, sellepärast koosneb CPU mälusüsteem mitmest erineva taseme, kiiruse ja suurusega vahemälust, mis hõivavad tänapäeval suurima osa CPU transistoridest [5]. Korduvalt kasutatavate andmete hoidmiseks mõeldud vahemäludest on vähe kasu selliste andmete puhul, mida kasutatakse vaid korra, nagu näiteks graafikaarvutustes kasutatavad andmed. GPU mälusüsteem rõhub mälu ribalaiusele võimalikult suure hulga andmete edastamiseks viivituste vähendamiste asemel, mis tagab üldkokkuvõttes suurema jõudluse.

5. Voogarvutus

Suure arvutusjõudlusega protsessor peab olema efektiivne nii arvutuste teostamisel kui andmete vahetamisel mäluga. Käesolevas peatükis kirjeldatav voogprogrammeerimise mudel võimaldab neid kahte nõuet rahuldada.

5.1 Voogprogrammeerimise mudel

Voogprogrammeerimise mudeli puhul esitatakse kõik andmed voona [6]. Andmevoog koosneb järjestatud, sama tüüpi andmetest. Andmed võivad olla lihttüüpi (täis- või ujukomaarvude voog) või keerukamad (punktide või nelinurkade voog). Andmevoog võib olla suvalise pikkusega, kuid andmevoogudele rakendatavad operatsioonid on efektiivsemad pikkade voogude korral, mis koosnevad sadadest või rohkematest elementidest. Rakendatavate operatsioonide hulka kuuluvad näiteks voogude kopeerimine, alamvoogudeks jagamine ja arvutusoperatsioonide teostamine kernelite abil.

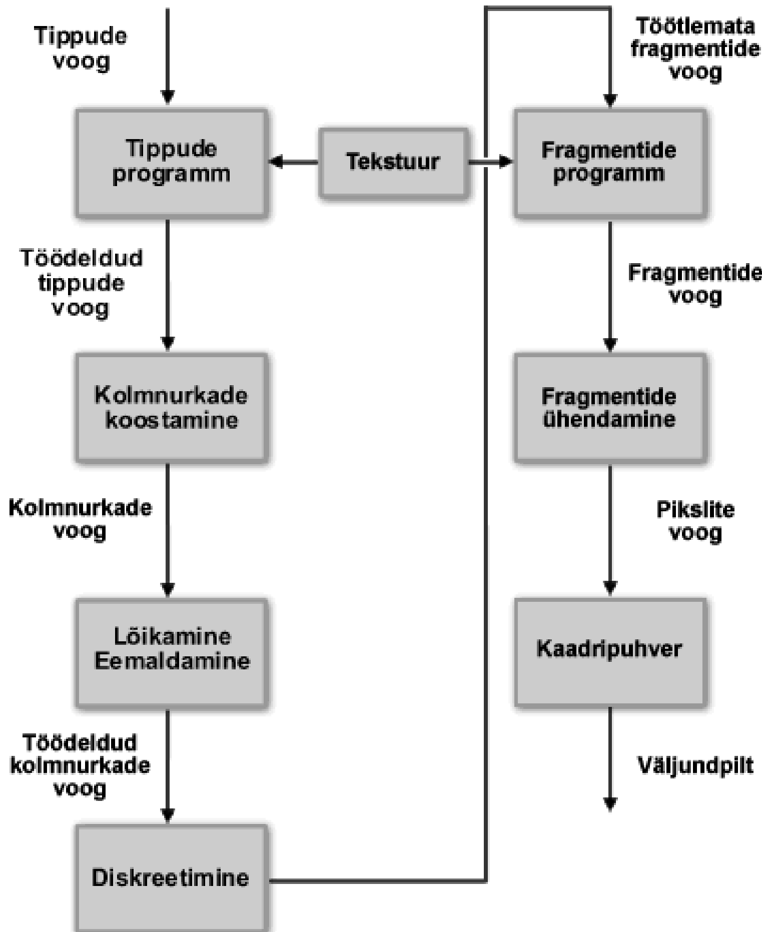
Kernel teostab operatsioone elementide voogudel, mitte üksikutel elementidel, võttes sisendina ühe või mitu andmevoogu ning väljastades ühe või mitu andmevoogu. Kerneleid kasutatakse peamiselt funktsioonide rakendamiseks andmevoo elementidele, näiteks punktide voo teisendamine ühest koordinaadisüsteemist teise. Kernelite operatsioonide hulka kuuluvad veel näiteks laiendamine (ühest kernelisse sisenevast elemendist tekib arvutuste käigus rohkem kui üks väljundelement), vähendamine (mitu sisendelementi kombineeritakse üheks väljundelemendiks) või filtreerimine (kerneli väljundelementide voost puuduvad osad kernelisse sisenenud elemendid).

Kerneli väljund on funktsioon kerneli sisendist, kerneli sees teostatavad operatsioonid ühe andmeelemendiga ei sõltu operatsioonidest, mis teostatakse mõne teise elemendiga. Antud piirangul on kaks eelist. Esiteks, kerneli tööks vajalikud andmed on teada kerneli programmeerimise hetkel. See võimaldab kernelil efektiivsemalt töötada, kui kerneli sisendandmeid ja töö käigus tekkivaid vaheandmed hoida protsessori enda mälus. Teiseks, elementidele rakendatavate operatsioonide sõltumatus teistele elementidele

rakendatavatest operatsioonidest sama kerneli sees võimaldab kerneli tööd paralleelseks muuta.

Voogprogrammeerimise mudeli puhul koostatakse arvutamiseks kasutatav programm mitmetest üksteisega jadamisi ühendatud kernelitest [6]. Joonisel 4 on näidatud, kuidas kogu graafikakonveier sobib voogprogrammeerimise mudeliga.

Graafikakonveier sobib voomudeliga hästi kokku. Graafikakonveier on jagatud erinevateks arvutusetappideks, mille vahel liiguvad andmevood, graafikakonveieri ülesehitus vastab voogprogrammeerimise mudelis kasutatavatele voogudele ja kernelitele. Arvutusetappide vahelised andmevood liiguvad ühe etapi väljundist otse teise etapi sisendisse. Samuti on sarnaselt kernelitega ühes arvutusetapis rakendatavad operatsioonid üldjuhul samad kõikide andmeelementide puhul.



Joonis 4. Graafikakonveieri sobivus voogprogrammeerimise mudeliga

Voogprogrammeerimise mudel on efektiivne arvutuste teostamisel, sest võimaldab ära kasutada kõiki kolme paralleelsuse tüüpi. Käskude paralleelsust – kernelite poolt teostatavaid arvutusoperatsioone on võimalik sooritada paralleelselt, näiteks koordi–naadisüsteemi teisendamisel arvuta punkti iga koordinaadi teisendus eraldi samaaegselt. Andmete paralleelsust – andmevoo üksikuid elemente on võimalik töödelda paralleelselt. Ülesannete paralleelsust – ülesandeid on võimalik kernelite vahel ära jagada.

Voogprogrammeerimise mudel on efektiivne ka mäluga suhtlemisel. Andmete edastamine voogudena, mitte üksikute elementidena, võimaldab vähendada edastus–protsessi alustamisega seotud ajalisi kulusid. Kernelite ühendamine jadasse teeb võimalikuks arvutuste vahetulemuste hoidmise protsessoris endas, vahetulemusi pole vaja saata aeglasemasse üldmällu. Konveiersüsteem võimaldab arvutusprotsessil pidevalt jätkuda, töö ei pea seiskuma, kuni mälust vajalikke andmeid laetakse. Voogprogrammeerimise puhul pole oluline mälu väike latentsus, vaid mälu suur ribalaius.

6. Nõuded ja piirangud graafikaprotsessoril töötavatele arvutustele

Graafikaprotsessor on mõeldud arvutigraafika ülesannete lahendamise jaoks ning erineb oma ülesehituselt arvuti keskprotsessorist. Graafikaprotsessori programmeerimine nõuab paralleelset lähenemist lahendatavale ülesandele.

Graafikaprotsessori tavaülesanne – väljundpildi loomine arvutiekraanile etteantud andmete põhjal, on väga töömahukas ülesande teostamiseks vajalike arvutus–operatsioonide poolest, arvutamiseks vajalikke andmeid laetakse mälust tavaliselt vaid korra kümnete või sadade teostatud operatsioonide järel. Seepärast saavutavad suurima kasvu kiiruses sellised graafikaprotsessori jaoks ümbertehtud tavaprogrammid, mille puhul vahekord arvutusoperatsioonide ning mälu poole pöördumiste vahel on tugevasti arvutusoperatsioonide kasuks. Sellisteks tavaprogrammideks on näiteks lineaar–algebralisi arvutusi teostavad programmid. Samuti ka füüsikavaldkonna kristallvõrede arvutussimulatsioonid.

Graafikaprotsessorile on kerge üle viia arvutusoperatsioone, mis kasutavad arvutuste tegemisel kahemõõtmelisi andmestruktuure, näiteks maatriksarvutused, pilditöötlus, füüsikasimulatsioonid ja kiirtejälituse abil visualiseerimine. Graafikaprotsessori vasteks kahemõõtmelisele andmestruktuurile on tekstuur. Tekstuuride suurus on piiratud, tekstuuri suurim võimalik mõõde ühes suunas on 4096 pikslit ehk andmeelementi.

Andmevoo ühele elemendile arvutusoperatsioonide rakendamise käigus on sageli vaja ligipääsu teiste elementide andmetele kas kirjutamiseks või lugemiseks. Graafika–protsessori puhul on võimalikud kahte tüüpi andmeoperatsioonid: andmete lugemine (*gather*) ja andmete kirjutamine (*scatter*). Kui andmevoo elementi töötlev kernel vajab arvutusteks teiste andmevoo elementide andmeid, toimub andmete kogumine mälust, kui kernel edastab andmeid teistele andmevoo elementidel, toimub andmete kirjutamine mällu. Andmete kogumisel on vajalik ainult laadida mälus suvalistest kohtadest asuvaid andmeid (*random-access read*), kirjutamisel on vajalik ainult kirjutada andmeid suvalistesse kohtadesse mälus (*random-access write*) [4].

Graafikaprotsessor teostab arvutustehteid vaid ujukomaarvudega, täisarvudega operatsioone teha ei saa. Samuti on ujukomaarvud piiratud täpsusega. Nvidia GeForce FX ja 6 seeria graafikaprotsessorid toetavad 16 bitiste ja 32 bitiste ujukomaarvude kasutamist, Ati Radeon 9800 ja X800 graafikaprotsessorid 16 bitiste ja 24 bitiste ujukomaarvude kasutamist. Nvidia 32 bitine ujukomaarv (1 märgibitt, 10 mantissa bitti ja 23 eksponendi bitti) on sarnane IEEE 754 ujukomaarvude standardiga, samuti on toetatud ka arvutamisel tekkivad erijuhud: nulliga jagamine ja lõpmatus arvuruumi ületäitumise korral [7].

6.1 Graafikaprotsessoris sisalduvad arvutusressursid

Graafikaprotsessor sisaldab kahte tüüpi programmeeritavaid protsessoreid: tipuprotsessoreid ja fragmentide protsessoreid. Tipuprotsessorid töötlevad tippudest koosnevaid andmevooge ja rakendavad neile tipuprogrammi (*vertex program*, vahel nimetatakse ka *vertex shader*'iks). Seejärel arvutatakse iga tipunurkade kolmiku põhjal kolmnurgad, millest luuakse fragmentide andmevoog. Fragmentide protsessorid rakendavad igale fragmendile fragmendiprogrammi (*fragment program*, vahel nimetatakse ka *pixel shader*'iks), mis määrab kindlaks iga piksli lõpliku värvuse.

Kaasaegsed graafikaprotsessorid, näiteks Nvidia GeForce 6800 Ultra või Ati Radeon X800 XT, sisaldavad 6 tipuprotsessorit, mis on täies ulatuses programmeeritavad. Protsessorid töötlevad tippe kas SIMD või MIMD paralleelse andmetöötluse vormis. Et iga tipu ruumilisi koordinaate (x, y, z ja w) ja värvust (punane, roheline, sinine, alfaväärtus) esitatakse neljaelemendilise vektorina, on tipuprotsessorite riistvaraline arhitektuur optimeeritud töötlema neljaelemendilisi vektoreid parema jõudluse saavutamiseks. Tipuprotsessorid saavad muuta tippude koordinaate ja määrata seeläbi pikslite asukohti pildil ehk andmete asukohta mälus (piksli asukoha muutmine muudab tema asukohta mälus), aga ei saa lugeda teiste tippude andmeid. Seega omavad tipuprotsessorid võimet andmeid kirjutada, muutes tippudega seotud atribuute [3], kuid mitte andmeid lugeda.

Fragmentide protsessoreid on graafikaprotsessoris tavaliselt rohkem kui tipuprotsessoreid, Nvidia GeForce 6800 Ultra või Ati Radeon X800 XT sisaldavad 16 fragmentide protsessorit, mis on samuti täies ulatuses programmeeritavad. Fragmentide protsessorid töötlevad fragmente SIMD moel. Fragmentide protsessorid saavad lugeda

tekstuurides sisalduvaid andmeid, kuid ei saa muuta pikslite asukohti. Ehk fragmentide protsessor saab fragmentide andmeid küll lugeda, kuid tal puudub võimalus andmeid muuta.

GPGPU rakenduste puhul kasutatakse arvutustes tavaliselt fragmentide protsessoreid, mitte tipuprotsessoreid. Fragmentide protsessoreid on esiteks rohkem kui tipu–protsessoreid, teiseks läbib fragmentide protsessorite väljundandmete voog enne mällu kirjutamist väiksema teekonna. Tipuprotsessorite väljund peab eelnevalt läbima diskreetimisüksuse (*rasterizer*) ning fragmentide protsessorid.

Fragmentide protsessorid loevad andmeid tekstuuridest. Kui graafikaprotsessor visualiseerib pilti, kirjutatakse visualiseeritud pilt kas kaadripuhvrisse ekraanile saatmiseks või tekstuurimällu (*render-to-texture*). Tekstuurimällu kirjutamine on ainus otsene viis graafikaprotsessori arvutustulemusi ilma keskprotsessori abita edasistes graafikaprotsessori poolt teostatavates arvutustes kasutada. Tekstuurimällu kirjutamine ja sealt andmete lugemine ei sarnane keskprotsessori poolt tehtavate mälu lugemis- ja kirjutamisoperatsioonidega. Fragmentide protsessor võib oma kernelprogrammi töö käigus sooritada lugemisoperatsioone vabalt, kuid kirjutamisoperatsioon toimub kernelprogrammi lõpus. Kernelprogrammi valikud kirjutamisoperatsiooni teostamisel on piiratud: võimalik on üksikute pikslite kirjutamine keelata või kirjutada ühe puhvri asemel mitmesse puhvrisse (*multiple render target*).

7. Keskprotsessori ja graafikaprotsessori programmeerimise analoogiad

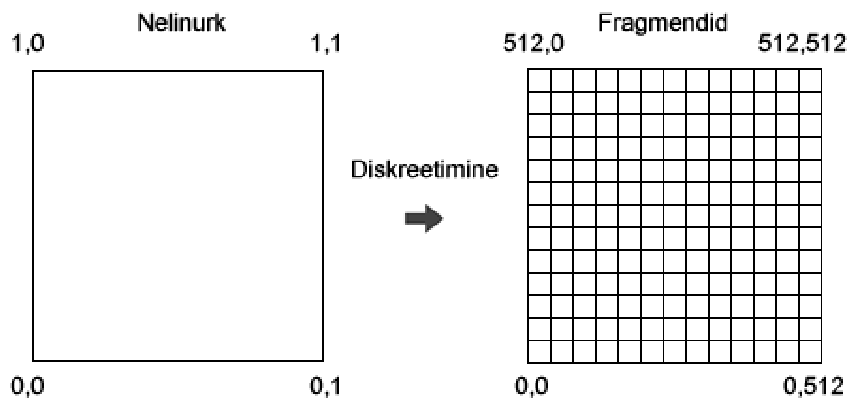
Keskprotsessori ja graafikaprotsessori programmeerimise metoodikad on küllaltki erinevad, kuid siiski on võimalik välja tuua sarnasusi nende kahe metoodika vahel.

Keskprotsessori programmeerimisel andmete hoidmiseks kasutatavale massiivile vastab graafikaprotsessori puhul tekstuur. Erinevalt massiivist on tekstuur piiratud suuruse ja mõõtmetega, tekstuuril võib olla kuni kolm mõõdet ning kuni 4096 elementi mõõtme kohta: kahemõõtmeline tekstuur saab maksimaalselt koosneda 4096*4096 elemendist.

Fragmentide protsessorite (ka tipuprotsessorite) poolt teostatav andmevoogude kernelarvutus sarnaneb keskprotsessorite programmitsüklitele [8]. Keskprotsessori puhul töödeldakse tsükli abil järjest massiivis leiduvaid elemente jadamisi. Ühe tsükli sees täidetavad käsud moodustavad kerneli. Graafikaprotsessori puhul kirjutatakse andmeelementidele rakendatavatest käskudest fragmentide programm, mida raken–datakse fragmentide protsessorite abil kõigile jada elementidele paralleelselt.

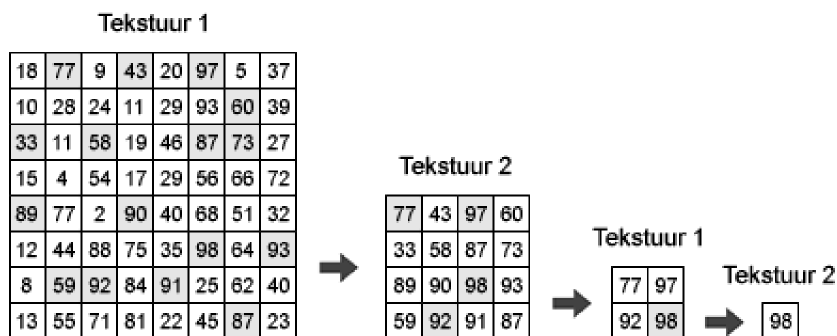
Keskprotsessor võib vabalt kirjutada ja lugeda andmeid mälust, nii vahepeal kui ka ühe arvutusetapi lõppedes on andmed lihtsasti kättesaadavad ja kasutatavad programmi edasise töö jaoks. Graafikaprotsessori puhul ei ole võimalik arvutustulemuste vahe–pealne kirjutamine mällu enne fragmentide programmi lõppu. Arvutusetapi tulemuste edasiandmiseks järgmisele arvutusetapile on vaja tulemused kirjutada tekstuurimällu.

Keskprotsessori puhul on arvutusprotsessi käivitamine triviaalne. Graafikaprotsessoril arvutatava fragmentide programmi käivitamiseks on vajalik esmalt tekitada fragmentide voog ehk anda käsk geomeetrilise kujundi joonistamiseks [9]. Tipuprotsessorid muudavad vastavalt kaameranurgale kujundi asukohta ekraanil ning diskreetimisüksus määrab kindlaks, milliseid ekraanipunkte kujund katab ja tekitab iga kaetud punkti kohta fragmendi (joonis 5). GPGPU arvutuste puhul antakse programmi käivitamiseks tavaliselt käsk joonistada nelinurk (*quad*).



Joonis 5. Fragmentide tekkimine: ekraanilahutus 512*512 pikslit, joonistatakse ekraanisuurune nelinurk.

Arvutuste puhul, mil sisendiks on palju andmeelemente ning väljundelementideks üks või mõni element, nagu summa või suurima elemendi leidmine, tuleb arvutamisel kasutada puhvrite paare. Arvutusetapi algandmeid sisaldav puhver suuremate mõõtmetega ja arvutusetapi väljundandmeid hoidev puhver vajaliku suurusega. Kui arvutusetappe on mitu, vahetatakse etappide vahepeal väljund- ja sisendpuhver ning vähendatakse väljundpuhvri mõõtmeid. Näiteks kahemõõtmelise massiivi suurima elemendi leidmine (joonis 6): igal arvutusetapil võrreldakse neljakaupa grupeeritud väärtusi, millest valitakse suurim, mis kirjutatakse väljundpuhvrisse. Igal arvutusetapil väheb väljundväärtuste arv 4 korda ning massiivi mõõtmed 2 korda. Vastavalt vähendatakse ka väljundina kasutatava puhvri mõõtmeid.



Joonis 6. Suurima elemendi leidmine.

Tekstuurikoordinaadid on analoogsed massiiviindeksitega. Tekstuurikoordinaadid on seotud tippudega, diskreetimisüksus tekitab tippudelt saadud tekstuurikoordinaatide põhjal iga fragmendi jaoks vastavad tekstuurikoordinaadid [10]. Fragmentide tekstuurikoordinaate kasutatakse indeksitena tekstuurist andmete lugemisel.

Arvutustehete tulemuste tekstuuri (massiivi) suurus on seotud fragmentide arvuga. Fragmentide arv on määratud tipuprotsessoritele antud tipuandmete ja tipuprogrammi poolt. Tavaliselt määratakse nelja tipupunkti abil vajaliku suurusega nelinurk ja tipu–punkte tipuprogrammiga ei muudeta. Seega teiseneb nelinurk diskreetimisel valitud nelinurga pindalale vastavalt fragmentideks.

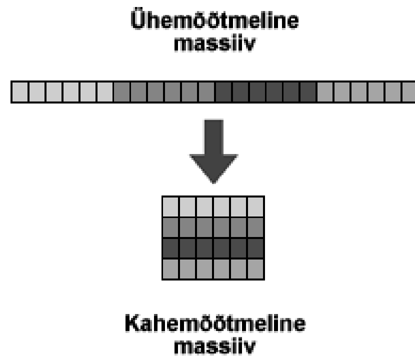
8. Andmestruktuurid

GPGPU töös kasutatavad andmestruktuurid on sõltuvad graafikaprotsessori poolt toetatavatest tekstuuridest. Järgnevalt on ära toodud nimekiri graafikaprotsessori tekstuuridest koos maksimaalsete mõõtmetega:

- Ühemõõtmeline tekstuur, maksimaalne suurus 4096 elementi.
- Kahemõõtmeline tekstuur, maksimaalne suurus 4096^2 elementi.
- Kolmemõõtmeline tekstuur, maksimaalne suurus 512^3 elementi.
- Kuuptekstuur, koosneb kuuest ruudukujuliste mõõtmetega kahemõõtmelisest tekstuurist.

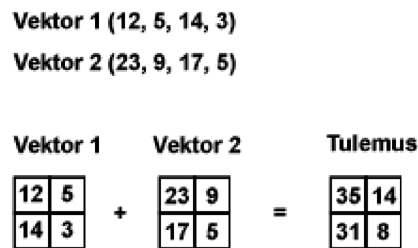
Graafikaprotsessoril teostatavate arvutuste tulemused kirjutatakse kahemõõtmelise pildimassiivina kaadripuhvrisse [11], kust neid on edasiste arvutuste tarbeks või tulemuste tagastamiseks keskprotsessorile võimalik kirjutada tekstuuri. Et tulemused väljastatakse kahemõõtmelise massiivina, ei ole nende kirjutamine ühemõõtmelisse tekstuuri otstarbekas. Kolmemõõtmelisse tekstuuri kirjutamise puhul on vaja eelnevalt määrata tekstuuri kiht, millesse andmed kirjutatakse. Samuti osutub paljude arvutuste juures kolmemõõtmelise tekstuuri maksimaalne suurus arvutuste mahukust piiravaks asjaoluks. Kuuptekstuuri kirjutamisel tuleb määrata kuubi külg, millele tekstuuri andmed kirjutatakse. Seepärast kasutatakse GPGPU arvutuste puhul andmete hoidmiseks kahemõõtmelisi tekstuure [8].

Kui arvutuste algandmed ja väljund on kahemõõtmelisest andmemassiivist erinevates mõõtmetes, tuleb arvutuste teostamise ajaks andmed kahemõõtmelisse massiivi ümber paigutada. Ühemõõtmelise massiivi puhul tuleb kahemõõtmeline massiiv täita rida-realt (joonis 7) ühemõõtmelise massiivi andmetega. Kolmemõõtmelise massiivi puhul tuleb massiiv kahemõõtmelisteks kihtideks eraldada, sama tuleb teha ka kõrgemamõõtmeliste andmemassiivide puhul.



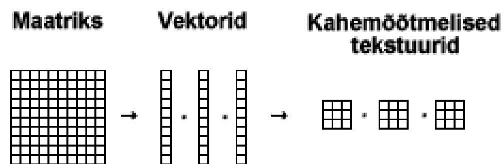
Joonis 7. Ühemõõtmelise massiivi teisendamine kahemõõtmeliseks

Vektoritega arvutamisel tuleb vektorid teisendada operatsioonide teostamise jaoks kahemõõtmelisele kujule. Joonisel 8 on toodud näide kahe vektori liitmise kohta.



Joonis 8. Vektorite esitamine kahemõõtmelisel kujul operatsioonide teostamiseks

Tihedate maatriksitega arvutamisel käsitletakse tihedat maatriksit kui veergude vektorite hulka (joonis 9).



Joonis 9. Tiheda maatriksi seadmine vastavusse kahemõõtmelise tekstuuriga

Maatriksi iga veerg seatakse vastavusse vektoriga, mis omakorda viiakse üle kahe–mõõtmelisele kujule.

9. Programmeerimiskeeled ja –vahendid

Graafikaprotsessori muutumine kasutaja poolt programmeeritavaks on toimunud alles hiljuti. Esimesed programmeeritavad graafikaprotsessorid ilmusid aastal 2001, 2002. aastal lisati ujukomaarvutuste tugi ja 2004. aastal programmikoodis hargnemiste kasutamise võimalus. Esialgu oli ainus viis tipu- ja fragmentide programmide kirjutamiseks kasutada graafikaprotsessorite assemblerkeelt, mis ühest küljest pakub küll täielikku kontrolli täidetava programmikoodi üle, kuid oli programmeerimisel tülikas ning liialt riistvaraspetsiifiline. Programmeerimise lihtsustamiseks ja programmikoodi porditavuse parandamiseks on arendatud mitmeid kõrgtaseme keeli, millest käesolev peatükk järgnevalt lühikese ülevaate annab.

9.1 Cg

Cg (*C for graphics*) on nVidia poolt loodud kõrgtaseme keel tipu- ja fragmentide protsessorite programmeerimise jaoks. Cg põhineb programmeerimiskeelel C ning on sama süntaksiga kuid väikeste muutustega, samuti on keelele lisatud uusi andmetüüpe. Et varasemate graafikaprotsessorite poolt pakutavad võimalused tipu- ja fragmentide programmide koostamisel erinevad uuemate graafikaprotsessorite omadest, kasutab Cg graafikaprotsessorite eristamiseks profiile. Profiilid määravad, kui keerukas saab olla Cg programm. Cg ja C põhilised erinevused on järgmised [12]:

- Cg ei toeta võtmesõnu *goto*, *break*, *continue*, *switch*, *case* ja *default*.
- Pole võimalik kasutada viitasid ja nendega seotud operatsioone.
- Massiivid on toetatud, kuid piiratud suuruse ja mõõtmetega.
- Muutujaid võib deklaleerida ka mujal kui vaid koodi alamlõigu alguses.
- Numbriliste muutujate tüüpide käsitlemine sõltub graafikaprotsessori profiilist. Kuigi kõik profiilid toetavad Cg andmetüüpe, võib profiil asendada täisarvud ujukomaarvudega või kasutada täistäpsusega ujukomaarve vähendatud täpsusega.

Cg toetab seitset andmetüüpi, kuid hetkel pole muutujatüüp *string* ühegi profiili puhul programmikoodis kasutatav, *stringi* väärtusi on küll võimalik seada ning lugeda, nendes

võib hoida näiteks informatsiooni Cg faili kohta. Ülejäänud kuus toetatud andmetüüpi on järgmised:

- *Float*, 32 bitine ujukomaarv
- *Half*, 16 bitine ujukomaarv
- *Int*, 32 bitine täisarv
- *Fixed*, 12 bitine fikskomaarv
- *Bool*, loogikamuutuja
- *Sampler**, viit tekstuuriobjektile

Cg toetab ka eespool loetletud andmetüüpidel põhinevaid vektoreid ja maatrikseid suurustega kuni vastavalt neli või neli korda neli andmeelementi. Toetatud on ka struktuurid ja massiivid. Cg saab kasutada nii DirectX kui ka OpenGL graafilise programmiliidesega.

9.2 HLSL

HLSL (*High Level Shader Language*) on Microsofti poolt loodud kõrgtaseme keel tipu- ja fragmentide programmide loomiseks. HLSL on väga sarnane Cg programmeerimis–keelega, erinevused on väikesed, näiteks toetab HLSL *double* andmetüüpi, mis puudub keeles Cg. HLSL on kasutatav ainult DirectX programmiliidesega ja seega töötab vaid koos Windowsi operatsioonisüsteemiga.

9.3 OpenGL Shading Language

OpenGL Shading Language (kasutatakse ka lühendit GLSL) on OpenGL standardi osa alates OpenGL versioonist 2.0. GLSL on kasutatav koos OpenGL programmiliidesega. Sarnaselt HLSL ja Cg on ka GLSL edasiarendus programmeerimiskeelest C ning on keeltega HLSL ja CG väga sarnane. Põhierinevus seisneb keelte kompilaatorite asukohas [13]. HLSL ja Cg puhul asub kompilaator enne programmiliidese draiverit, kompilaator tõlgib programmikoodi assemblerkeelde, mis saadetakse edasi draiverile. GLSL kompilaator asub otse OpenGL draiveris, mis võimaldab programmikoodi otse ja efektiivsemalt masinkeelde tõlkida.

9.4 BrookGPU

Brook on C laiendus eesmärgiga lisada programmeerimiskeelele C paralleelne andmetöötlus ehk voogprogrammeerimise võimalus [14]. BrookGPU on Brook'i kompilaator graafikaprotsessorite jaoks. Brook defineerib uue andmetüübi voog (*stream*) ning spetsiaalse funktsioonitüübi kernel. Voog on kogum andmetest, mida on võimalik paralleelselt töödelda. Voog on sarnane C massiiviga, kuid esinevad järgmised erinevused:

- Väljaspool kerneleid pole lubatud indeksite abil voo elementide poole pöörduda.
- Väljaspool kerneleid pole lubatud voole väärtusi omistada.
- Vooge pole võimalik staatiliselt algväärtustada.
- Voogude lugemine ja kirjutamine toimub kernelites või spetsiaalsete käskude abil.

Kernelid on spetsiaalsed funktsioonid voogude töötlemiseks. Kernelit rakendatakse kõigile sisendvoo elementidele. Kerneli definitsioon erineb funktsiooni definitsioonist järgnevalt:

- Definitsioonile eelneb märksõna *kernel*.
- Tagastustüüp on alati *void*.
- Üks kerneli voog tüüpi parameetritest peab olema märgitud väljundvooks märksõna *out* abil.

Brook toetab voogude paralleelseid vähendusoperatsioone – voo teisendamist väiksemaks vooks või üheks elemendiks. Vähendusoperatsioonid peavad olema assotsiatiivsed ja kommutatiivsed. Lubatud vähendusoperatsioonide hulka kuuluvad liitmine, korrutamine, minimaalse ja maksimaalse elemendi leidmine, *or*, *and* ja *xor* bitioperatsioonid. Mittelubatud vähendusoperatsioonide hulka kuuluvad lahutamine ja jagamine. Vähendusoperatsioone teostav funktsioon võtab sisendparameetritena vaid kaks voogu, sisendvoo ning märksõnaga *reduce* tähistatud väljundvoo, sisend- ja väljundvoog peavad olema sama tüüpi.

9.5 Glift

Glift on arendatud graafikaprotsessori andmestruktuuride loomise ja kasutamise lihtsustamiseks [15]. Glifti eesmärkideks on esiteks abstraheerida graafikaprotsessori mälumudel, et teha võimalikuks keerukate andmestruktuuride loomine vähese koodihulgaga. Teiseks eesmärgiks on eraldada üksteisest GPU andmestruktuurid ja algoritmid, muutes seeläbi keerukate programmmide loomise lihtsamaks. Kolmas eesmärk on abstraheerimise tõttu tekkivad jõudluskadusid võimalikult minimaliseerida.

Glift on üles ehitatud viiele komponendile: füüsiline mälu, virtuaalne mälu, aadresside teisendaja, iteraator ja konteinerite adapter.

Füüsiline mälu on graafikaprotsessori tekstuurimälu abstraheering, füüsilise mälu komponent määrab andmete hoidmise struktuuri. Virtuaalse mälu komponendiga määratakse algoritmi töös kasutatav andmestruktuur vastavalt algoritmi vajadusele, mitte kasutatavale füüsilisele komponendile. Näiteks võib algoritm kasutada kolmemõõtmelist andmestruktuuri, kuigi andmeid hoitakse kahemõõtmelistes struktuurides.

Aadresside teisendaja seab füüsilise ja virtuaalse mälu teineteisega vastavusse, aadresside teisendaja on Glift'i komponentidest suurima tähtsusega. Aadresside teisendaja muudab võimalikuks üksikute andmeteelementide lugemise ja kirjutamise andmestruktuuri suvalisest kohast ning suurte andmehulkade kopeerimise, kirjutamise ja üle nende itereerimise.

Iteraator moodustab liidese algoritmide ja andmestruktuuride vahel, kõrvaldades vajaduse tegeleda detailselt andmete järjestikulise töötlemise protseduuridega, andmete ligipääsu õiguste ja mustritega.

Konteinerite adapter on Glift'i kõrgema taseme andmestruktuur. Konteineri adapter kasutab oma töös juba eelnevalt loodud konteinerit. Konteineri adapterite näideteks on Glift'i massiivi peale loodud graafikaprotsessori pinu ja *quadtree*/*octree* andmestruktuurid, mis on loodud aadresside teisendajat kasutades.

Kokkuvõte

Antud bakalaureusetöös on esitatud ülevaade graafikaprotsessori arhitektuurist ning graafikakonveieri tööst, samuti on võrreldud graafikaprotsessori ning keskprotsessori arhitektuuridest tulenevaid erinevusi. Graafikaprotsessori paralleelsele andmetöötlusele suunatud ülesehitus lubab suure arvutusmahuga ülesannete puhul saavutada kesk–protsessoriga võrreldes oluliselt lühemaid arvutusaegu. Seda vaatamata asjaolule, et programmeeritavate graafikaprotsessorite ilmumisest on möödunud vaid viis aastat. Ülesannete programmeerimiseks on olemas kõrgtaseme programmeerimiskeeled ja arendusvahendid. Graafikaprotsessorite arvutusvõimsuse kiire kasv jätkub ka tulevikus, paralleelne arhitektuur võimaldab vaid lihtsalt protsessorile arvutusüksuste lisamisega jõudlust tõsta.

Siiski pole keskprotsessori asendamine graafikaprotsessoriga lihtne ning mõnede ülesannete puhul ka võimalik. Graafikaprotsessori mäluarhitektuur seab piiranguid kasutatavatele andmestruktuuridele ja –mahtudele. Arvutuste täpsus on piiratud 32bitiliste ujukomaarvudega, suurema täpsuse jaoks on vajalik keskprotsessori kaasamine arvutustesse, mis vähendab graafikaprotsessori kasutamisest saavutatavaid kiirusevõite. Samuti puudub täisarvude kasutamise võimalus. Sellele vaatamata võimaldab graafikaprotsessori kasutamine arvutusülesannete lahendamist tunduvalt kiirendada. Antud bakalaureusetööd on võimalik kasutada tulevastes töödes graafika–protsessorite programmide ja teekide kirjutamisel.

Usage of graphics processors for speeding up scientific computation

Bachelor thesis

Holger Biene

Abstract

At present, central processor manufacturers are developing parallel processor technologies in order to produce processors with higher computational capability, as raising clock speeds to speed up processors is no longer a viable option. When it comes to parallel processor technologies, graphics processors are ahead of central processors: graphics processors have inherently parallel architecture in order to keep up with constant demand for greater computational capability. As in recent years graphics processors have become user programmable, graphics processors are able to outperform central processors at arithmetically intense computing tasks.

This bachelor thesis gives an overview of a graphics processors architecture (based on Nvidia GeForce 6800) and graphics pipeline and discusses differences in the architecture of graphics and central processors. Stream processing is introduced, as the work procedure of graphics processor is very similar to a stream processor. To use the high computational capability of a graphics processor, the user has to deal with certain problems and limitations as programming a graphics processor differs from the programming of a central processor, but some similarities do exist; both the differences and similarities are discussed. Finally, an overview of usable data structures and programming languages is given. Current thesis can be used in future work of writing programs and libraries to use graphics processors for speeding up calculations.

Kasutatud kirjandus

[1] N.Evanson, *Hardware vocabulary*, 2005. http://www.futuremark.com/community/hardwarevocabulary/?page=6 (viimati vaadatud 24.05.2006).

[2] H. Vallaste, *E-teatmik: IT ja sidetehnika seletav sõnaraamat*. http://www.vallaste.ee/ (viimati vaadatud 26.05.2006).

[3] NVidia, *Vertex shaders*. http://www.nvidia.com/object/feature_vertexshader.html (viimati vaadatud 26.05.2006).

[4] M. Pharr. R. Fernando. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison Wesley Proffesional, 2005.

[5] Wikipedia, *Stream Processing*. http://en.wikipedia.org/wiki/Stream_processing (viimati vaaadatud 23.05.2006).

[6] J. D. Owens, *Computer Graphics on a stream architecture*, Stanford University, 2002.

[7] C. Cebenoyan, *Floating point specials on the GPU*. Nvidia, 2005.

[8] D. Göddeke, *GPGPU tutorial*. http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html (viimati vaadatud 26.05.2006).

[9] GPGPU.org *The official GPGPU FAQ*. http://www.gpgpu.org/w/index.php/FAQ (viimati vaadatud 22.05.2006).

[10] J. Hoxley, *Texture coordinates*, 2005. http://nexe.gamedev.net/directKnowledge/default.asp?p=texture%20coordinates (viimati vaadatud 22.05.2006).

[11] SGI. *Framebuffer object*, 2006. http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt (viimati vaadatud 25.06.2006).

[12] Nvidia. *Cg toolkit, users manual, release 1.4.1*. 2006.

[13] R. J. Rost, *OpenGl shading language, second edition*. Addison Wesley Proffesional, 2006.

[14] Stanford University, *BrookGPU*. http://graphics.stanford.edu/projects/brookgpu/index.html (viimati vaadatud 25.06.2005).

[15] A. Lefohn, J. M. Kniss, R. Strzodka, S. Sengupta, J. D. Owens. *Glift: Generic, Efficient, Random-Access GPU Data Structures*. ACM Transactions on Graphics, 25(1), lk 60-99, 2006.

Lisad

Lisa 1

Näidisprogramm

```
// Antud näidisprogramm korrutab etteantud massiivi
// väärtused konstandiga ning trükib esialgsed andmed
// ning lõpptulemused arvutiekraanile. Näidisprogrammi
// koostamisel on kasutatud [8] näiteid ja juhiseid.

#include <stdio.h>
#include <stdlib.h>
#include <GL/glew.h>
#include <GL/glut.h>

int main(int argc, char **argv) {
   // paneme paika kasutatavate tekstuuride mõõtmed
   // texSize = tekstuuri küljepikkus
   int texSize = 2;

   // konstant, kasutame hiljem andmete korrutamisel
   float alpha = 2.0;

   // loome massiivid alg- ja lõppandmetega
   // täidame data massiivi algandemtega
   float* data = (float*)malloc(texSize*texSize*sizeof(float));
   float* result = (float*)malloc(texSize*texSize*sizeof(float));
   for (int i=0; i<texSize*texSize; i++)
      data[i] = i+1.0;

   // initsialiseerime GLUT ja GLEW
   glutInit (&argc, argv);
   glutCreateWindow("TEST");
   glewInit();

   // loome FrameBuffer objekti
   GLuint fb;
   glGenFramebuffersEXT(1,&fb);
   glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,fb);
   // seame visualiseerimise nii, et pikslite aadressid vastaksid
   // tekstuurikoordinaatidele üks-ühele
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
```

```
gluOrtho2D(0.0,texSize,0.0,texSize);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glViewport(0,0,texSize,texSize);

// loome andmeid sisaldava tekstuuri texY
GLuint texY;
glGenTextures (1, &texY);
glBindTexture(GL_TEXTURE_RECTANGLE_ARB,texY);
// seame tekstuuri parameetrid
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
            GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
            GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
            GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
            GL_TEXTURE_WRAP_T, GL_CLAMP);
// määrame tekstuuri andmeformaadiks ujukomaarvud
glTexImage2D(GL_TEXTURE_RECTANGLE_ARB,0,GL_FLOAT_R32_NV,
          texSize,texSize,0,GL_LUMINANCE,GL_FLOAT,0);
// laadime algandmed tekstuuri
glBindTexture(GL_TEXTURE_RECTANGLE_ARB,texY);
glTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB,0,0,0,texSize,texSize,
            GL_LUMINANCE,GL_FLOAT,data);

// loome lõppandmeid sisaldava tekstuuri texX
GLuint texX;
glGenTextures (1, &texX);
glBindTexture(GL_TEXTURE_RECTANGLE_ARB,texX);
// seame tekstuuri parameetrid
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
            GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
            GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
            GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
            GL_TEXTURE_WRAP_T, GL_CLAMP);
// määrame tekstuuri andmeformaadiks ujukomaarvud
glTexImage2D(GL_TEXTURE_RECTANGLE_ARB,0,GL_FLOAT_R32_NV,
          texSize,texSize,0,GL_LUMINANCE,GL_FLOAT,0);
// kinnitame tekstuuri FrameBuffer objekti külge
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                GL_COLOR_ATTACHMENT0_EXT,
                GL_TEXTURE_RECTANGLE_ARB,texX,0);
glBindTexture(GL_TEXTURE_RECTANGLE_ARB,texX);
// täidame tekstuuri andmetega
glTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB,0,0,0,texSize,texSize,
            GL_LUMINANCE,GL_FLOAT,data);
```

```
// GLSL muutujad
GLhandleARB programObject;
GLhandleARB shaderObject;
GLint yParam, alphaParam;

// loome programmi objekti
programObject = glCreateProgramObjectARB();
// loome fragmentide programmi objekti ning lisame selle GLSL programmi
shaderObject = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);
glAttachObjectARB (programObject, shaderObject);
// fragmentide programmi kood
const GLcharARB* program_source =
	"uniform sampler2DRect textureY;" \
	"uniform float alpha;" \
	"void main(void) { " \
	"float y = texture2DRect(textureY, gl_TexCoord[0].st).x;" \
	"gl_FragColor.x = y * alpha;"\
	"}";

glShaderSourceARB(shaderObject, 1, &program_source, NULL);
// kompileerime koodi ja ühendame kokku
glCompileShaderARB(shaderObject);
glLinkProgramARB(programObject);
// salvestame viitade andmed
yParam = glGetUniformLocationARB(programObject, "textureY");
alphaParam = glGetUniformLocationARB(programObject, "alpha");

// lubame GLSL programmi kasutamise ja seame muutujad
glUseProgramObjectARB(programObject);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_RECTANGLE_ARB,texY);
glUniform1iARB(yParam,0);
glUniform1fARB(alphaParam,alpha);

// seame texX tekstuuri arvutustulemuste salvestamise kohaks
glDrawBuffer (GL_COLOR_ATTACHMENT0_EXT);

// tekitame seest täidetud nelinurga
glPolygonMode(GL_FRONT,GL_FILL);
// ja joonistame selle arvutamisprotsessi alustamiseks
glBegin(GL_QUADS);
glTexCoord2f(0.0, 0.0);
glVertex2f(0.0, 0.0);
glTexCoord2f(texSize, 0.0);
glVertex2f(texSize, 0.0);
glTexCoord2f(texSize, texSize);
glVertex2f(texSize, texSize);
glTexCoord2f(0.0, texSize);
glVertex2f(0.0, texSize);
glEnd();
```

```
    // loeme texX tekstuurist fragmendiprogrammiga
    // töödeldud andmed
    glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
    glReadPixels(0, 0, texSize, texSize,GL_LUMINANCE,GL_FLOAT,result);

    // trükime alg -ja lõppandmed ekraanile
    printf("Algandmed:\n");
    for (int i=0; i<texSize*texSize; i++)
        printf("%f\n",data[i]);
    printf("Lõppandmed:\n");
    for (int i=0; i<texSize*texSize; i++)
        printf("%f\n",result[i]);

    // vabastame hõivatud ressursid
    free(data);
    free(result);
    glDeleteFramebuffersEXT (1,&fb);
    glDeleteTextures (1,&texY);
    glDeleteTextures (1,&texX);
    return 0;
}
```