

Speciale

Acceleration af Kollisionsdetektion på Parallele Computerarkitekturer

Andreas Rune Fugl [140983]

Thomas Frederik Kvistgaard Ellehøj [230774]

Computer Science at the Faculty of Engineering

University of Southern Denmark

1. oktober 2008

Vejledere: Henrik Gordon Petersen og Jimmy Alison Jørgensen

Abstract

Dansk

Inden for gribesimuleringer af fler-aksede robothænder er der behov for et stort antal kollisionsdetektioner. En acceleration af disse kan give en stor samlet tidsbesparelse og dermed lette udviklingen af effektiv kontrolsoftware.

Emnet for dette speciale er at accelerere kollisionsdetektion med specielt fokus på gribesimuleringer. Acceleration opnås igennem to indsatsområder: Udnyttelse af parallelle computerarkitekturer samt optimering af kollisionsdetektionsalgoritmer til gribesimuleringer.

Resultaterne viser at det for scener af stor kompleksitet, er muligt at opnå en tidsbesparelse vha. parallelisering af traditionelle hierarkiske algoritmer.

English

For the simulation of multi-axis grippers, there is a large demand for efficient collision detection. Speeding up the collision detection can provide large performance gains in larger applications and will help developing efficient control software.

The topic of this master thesis is to speed up the collision detection for gripper simulations. This will be approached in two ways: First, the use of parallel computer architectures and second the optimization of algorithms for gripper simulations

Results have shown that for scenes of high complexity, there is potential for accelerating traditional hierarchical algorithms using parallel hardware.

Thomas Frederik Kvistgaard Ellehøj

Andreas Rune Fugl

Indhold

1	Introduktion	1
1.1	Baggrund for projektet	1
1.2	Problembeskrivelse	4
1.3	Relateret arbejde	5
1.4	Projektorganisering/Metode	5
I	Teori	8
2	Terminologi/Baggrund	9
2.1	Model repræsentation	9
2.2	Forespørgelsestyper	11
2.3	Hvad er en kollision?	12
3	Arkitekture og arkitektur valg	15
3.1	Parallel processering	15
3.2	Moderne parallelle arkitekture	17
3.3	Valg af arkitekturer	18
4	Algoritmer	19
4.1	Kollisionsdetektion	19
4.2	Generelt	19
4.3	Hierarkiske metoder	19
4.4	Forslag til metoder	25
4.5	Segment-piercing	26
4.6	Devillers	30
4.7	Orienting Bounding Box tests	40
4.8	Implementation	47
5	Graphics Processing Unit	48
5.1	Introduktion	48
5.2	Historie	48
5.3	Arkitektur	49
5.4	General-Purpose computation	50
5.5	Compute Unified Device Architecture	51
5.6	GPU platform og ydelse	56

5.7	Toolchain	58
6	Cell Broadband Engine	59
6.1	Introduktion	59
6.2	Arkitektur	59
6.3	Playstation 3	64
6.4	Teoretisk ydelse	65
6.5	Toolchain	65
II	Implementation	70
7	Timing tests af platforme	71
7.1	GPU	71
7.2	CELL/BE	78
8	Framework og library	85
8.1	Indledning	85
8.2	Organisering af kode	85
8.3	LibOPP API	92
8.4	Build environment	103
8.5	Delkonklusion	105
9	Algoritme implementation	107
9.1	Trekant-trekant tests	107
9.2	OBB tests	126
9.3	Test	128
9.4	Delkonklusion	131
10	Implementation af hierarkisk metode	133
10.1	Partitionering	133
10.2	Test	150
III	Resultater og konklusion	154
11	Erfaringer og problemer vedr. udviklingsprocessen	155
11.1	Generelt	155
11.2	GPU	155
11.3	CELL/BE	156
12	Konklusion	157
12.1	Perspektivering	158
IV	Appendix	159
	Ordliste	160

Tabeller	161
Figurer	162
Litteraturliste	165
A Determinanten og trippel skalar produktet	169
B Installation af Fedora release 7 på Playstation 3	172
B.1 Introduktion	172
B.2 Nødvendig software	172
B.3 Forbered PS3'eren på installationen	172
B.4 Installation af Fedora release 7	173
B.5 Byg en kerne med PS3 support	173
B.6 Opdater pakker	175
B.7 Spar på hukommelsen	176
C Software Development Kit for Multicore Acceleration version 3 på Playstation 3	178
C.1 Installation af IBMSDK	178
C.2 Installation af ekstra pakker	179
D GPU timing testdata	183
D.1 Test scripts	183
D.2 Memory	184

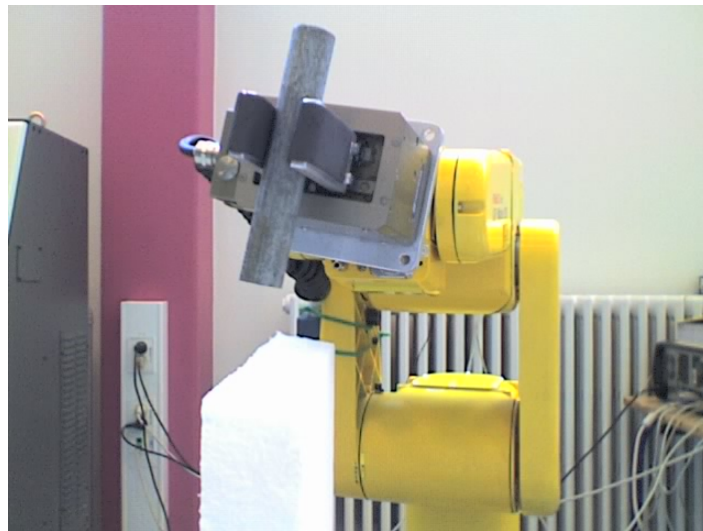
Kapitel 1

Introduktion

1.1 Baggrund for projektet

1.1.1 RoboCluster og Handyman

RoboCluster er et kompetencenetværk i Syddanmark der søger at fremme vækst og innovationen inden for robotter, automation og intelligente, mekaniske systemer [30]. Netværket består af kommercielle virksomheder, brancheorganisationer, offentlige aktører samt forsknings- og uddannelsesinstitutioner, der samarbejder om at udvikle og udnytte robotteknologi. RoboCluster's aktiviteter fokuserer på emner inden for produktinnovation, procesinnovation samt forskning og uddannelse.

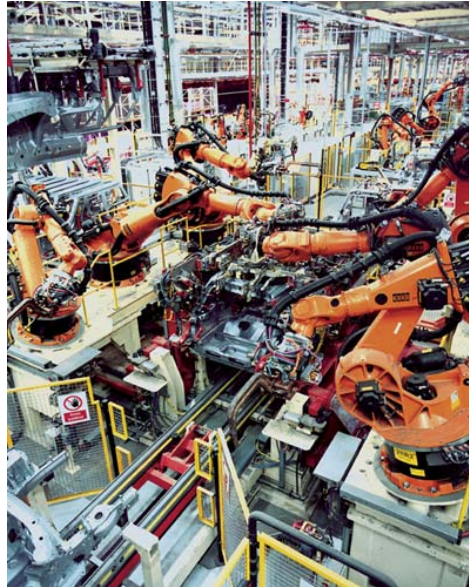


Figur 1.1: En industrirobot påmonteret en parallel griber.

Handyman er et projekt i RoboCluster der beskæftiger sig med udviklingen af en platform for griberobotter til avanceret emnehåndtering [29]. Målsætningen er at udvikle griberobotter til små produktserier, hvilket kræver stor fleksibilitet og omstillingsvenlighed. Netop de små produktserier med stor emnevariation og hyppige omstillinger af produktionsapparatet spås en væsentlig rolle for dansk produktion, da masseproduktion der førhen har ligget i Danmark i stigende grad flyttes til lande med lavere løn.

1.1.2 Griberobotter

I produktioner med store styktal har robotter været brugt med stor succes. Med deres høje præcision, hastighed og driftsikkerhed har de kunnet erstatte mennesker ved mange samlebånd. Men traditionelle industrirobotter kræver at deres bevægelser på forhånd programmeres ned til mindste detalje. De kræver desuden at de har værktøjer monteret, der ofte er specialdesignet til en bestemt opgave. Derfor er det oftest en besværlig og tidskrævende proces at omstille dem til nye opgaver.



Figur 1.2: Industrirobotter i bilproduktion. (Kilde: KUKA Robotics)

Mennesket besidder derimod en fantastisk evne til at tilpasse sig nye opgaver og er derfor lidt endnu, uundværlig i produktionssammenhænge. Med vores bevægeapparat, sanser og hjerne kan vi næsten øjeblikkeligt omstille os til en ny arbejdssituation, f.eks. nye genstande på samlebåndet. Hvor en robot vil have store problemer kan vi så let som ingenting genkende en genstand, vurdere dens vægt og overflade, gribe fat om den og udføre en arbejdsopgave.

Handyman-projektet søger at mindske forskellen på mennesker og robotter i håndtering af emner i industriproduktionen og dermed fremme fleksibiliteten og omstillingsvenligheden. Dette kræver en stor indsats på alle teknologifronter. [19].

Indsatsområder

Ved at bruge mennesket som analogi kan indsatsområderne deles op i følgende tre punkter

- Bevægeapparat
- Sanser
- Hjerne

Robotters gribeværktøjer er i dag oftest tilpasset den enkelte opgave og med disse høje priser er de ikke økonomiske for brug i mindre produktioner. Derfor ønsker man at efterligne det menneskelige

bevægeapparat, og især vores ekstremt fleksible hænder. Med moderne elektronik og mekanik er der i de senere år fremkommet en række mekaniske hænder, rækkende fra simple gribere til mange-aksede gribekløer og hænder [19].



Figur 1.3: En mangeakset hånd påmonteret en robot. (Kilde: Amtec Robotics)

Ud over at være i stand til at manipulere objekter, giver den menneskelige hånd os taktilt feedback. Især fingerspidserne har en stor tæthed af nerver, og dette giver nyttigt information når et objekt manipuleres [37], f.eks. overfladens beskaffenhed, hvor meget der trykkes, om objektet er ved at glide, temperatur mm. Trykfølsomme sensorer til robotværktøjer findes på markedet i dag og kan i mange henseende give den nødvendige information [19].

Et fleksibelt bevægeapparat og fintfølelse sanser er ikke til megen nytte, hvis der ikke er en intelligens som kan bearbejde og reagere på informationerne. Evolutionen har igennem flere millioner år udviklet den menneskelige hjerne til at have en høj intelligens og koordinationsevne der lader os udføre et væld af komplicerede opgaver [38]. Softwaren der udgør robotternes hjerner er i dag langt fra den intelligens som et menneske besidder. Hvor menneskehjernen ofte helt autonomt løser koordinerede bevægelser der anvender bevægeapparatet og sanser, er sekventielle computere helt afhængige af at have de nødvendige instruktioner.

En af de store udfordringer for Handyman-projektet er videreudvikle og integrere eksisterende software til at skabe en fleksibel og i høj grad selvlærende robotplatform, så slutbrugerens omstilling fra opgave til opgave bliver forenklet. Dette betyder blandt andet at platformen skal kunne forstå arbejdsbeskrivelser såsom 'tag dette objekt og læg det i kassen', at den hurtigt kan omstilles fra at gribe én type objekt til et andet og at den er tolerant overfor miljøændringer som f.eks. lysforhold [19].

Et centralt element i at kunne skabe den beskrevne platform er bevægelsesplanlægningen af robotten. Bevægelsesplanlægningen består overordnet set i at oversætte menneskers høj-niveaus beskrivelser af en opgave til lav-niveau bevægelsesinstruktioner [21, s.19]. Bevægelsesplanlægningen skal i Handyman simuleres på en computer i et 3D miljø, hvor man afprøver gribning. Dette vil foregå med en lang række eksperimenter, hvor man lader den virtuelle robot gribe fat om objektet og efterfølgende analyserer hvor hensigtsmæssigt grebet er. Ved hjælp af disse eksperimentelle erfaringer håber man at kunne opbygge en database, dvs. en hukommelse over hvilke greb der egner sig til forskellige situationer.



Figur 1.4: Prototype af tre-fingret griber med ni frihedsgrader, tre per finger. (Kilde: <http://www.flickr.com/photos/psinewave/1354728851>)

For stor fleksibilitet er det nødvendigt med en robotplatform der har et stort antal frihedsgrader. Dette gør sig gældende for selve robotten samt den monterede hånd. Et stort antal frihedsgrader har imidlertid store konsekvenser for udregningen af bevægelser. Det er ofte praktisk umuligt at løse problemet ved at udtrykke robotens tilladte bevægelser (også kaldet konfigurationsrum) på en matematisk eksplicit form. Imidlertid har andre metoder de seneste år haft stor succes. Disse er samplingsbaserede metoder, hvor eksplicitte modeller undgås ved at sample konfigurationsrummet på udvalgte punkter for information om hvorvidt en konfiguration ville betyde en kollision mellem robotten og omgivelserne. Dette gøres vha. et kollisionsdetections-modul, der kan anvende forskellige typer modeller uafhængigt af selve samplings-metoden [21, s.201].

Kollisionsdetection er en samlet betegnelse for algoritmer der har til formål at afgøre om objekter kolliderer. Dette speciale omhandler udviklingen af effektive kollisionsdetections-systemer til brug i simulering af emnehåndtering. Der vil blive lagt vægt på udnyttelsen af nye computerarkitekturer og optimering af algoritmer til simuleringssaplikationen.

1.2 Problembeskrivelse

Til Handyman-plattformen skal der udvikles kontrolsoftware for griberobotter der hurtigt kan omstille sig imellem forskellige arbejdsopgaver og forstå høj-niveaus arbejdsbeskrivelser. Dette kræver blandt andet at der opbygges eksperimentelle erfaringer mht, emnehåndtering og til det skal der foretages et stort antal simuleringforsøg. Da der i hvert enkelt simuleringforsøg er behov for et stort antal kollisionsdetectioner, kan acceleration af disse give en stor samlet tidsbesparelse og dermed lette udviklingen af effektiv kontrolsoftware.

Emnet for dette speciale vil være at accelerere kollisionsdetection med specielt fokus på gribesimuleringer. Acceleration skal opnås igennem to indsatsområder: Udnyttelse af parallelle computerarkitekturer samt optimering af kollisionsdetectionsalgoritmer til gribesimuleringer.

Alternative computerarkitekturer er interessante, da der på det seneste er kommet mere fokus på parallelle og vektoriserede arkitekturer, blandt andet multi-core processorer, og programmerbare GPU'er.

Der er derfor et stort potentiale for at kunne opnå en betragtelig acceleration ved at gå fra traditionelle CPU'er til specialiseret beregningshardware.

Gribesimulering er en specialisering af generel bevægelsesplanlægning og der er derfor mulighed for adskillige optimeringer i forhold til en mere generaliseret kollisionsdetektion.

1.3 Relateret arbejde

Inden for udnyttelsen af parallelle computerarkitekturer er der lavet en række implementationer af kollisionsdetektion på FPGA'er og GPU'er.

I [28] er der i en FPGA implementeret en hierarkisk metode for kollisionstests med brug af k-DOP's som bounding volumes (BV). Kollisionstests foregår vha. en separating-axis tests (SAT), som som beskrevet [8] der udføres i en dedikeret pipeline i FPGA'en. Pipelinen tager den øverste BV i hierarkiet og udfører en SAT. Kan der ikke findes en separerende akse (BV er i kollision) lægges de underliggende BV's i pipelinen for at testes. Implementationen anvender fixed-point aritmetik af hensyn til pladsforbrug i FPGA'en og der skal derfor tages et stort hensyn til dette når algoritmerne designes. Ligeledes er forsøget kun blevet udført i en simulering, og der tages derfor ikke hensyn til, hvordan data overføres fra/til kredsen.

[1] implementerer trekant-trekant kollisionstest i en FPGA baseret på metoden fra [23]. Systemet gemmer objekter og deres rumlige orientering i en lokal hukommelse der tilgås gennem et serielt interface. Ved simuleringer med en Xilinx Virtex-4 XC4VLX200 opnås der med 25 parallelle detektionskredsløb en ydelse på 56 millioner trekant-trekant tests per sekund. Dette holdes op imod en Pentium IV, 3GHz CPU med samme algoritme der kun opnår en ydelse på 1.5 millioner tests per sekund. I praksis har de dog kun lavet reelle tests på en mindre FPGA, der kun har plads til ét detektionskredsløb. Ydelsen bliver dermed kun 2.3 millioner tests per sekund. Det skal dog bemærkes at forfatterne ikke lader til at have gjort sig mange overvejelser over, hvad fixed-point beregningspræcisionen betyder for korrektheden af deres resultater.

I robotgruppen ved Maersk McKinnney Møller Institutet for Produktionsteknologi har Jimmy Alison Jørgensen (JAJ) arbejdet med hierarkiske metoder på FPGA'er målrettet Handyman-projektet.

1.4 Projektorganisering/Metode

For at have et solidt fundament til udviklingen af effektiv kollisionsdetektion, er det nødvendigt at se på hvilke analyse-mæssige resultater der ønskes opnået.

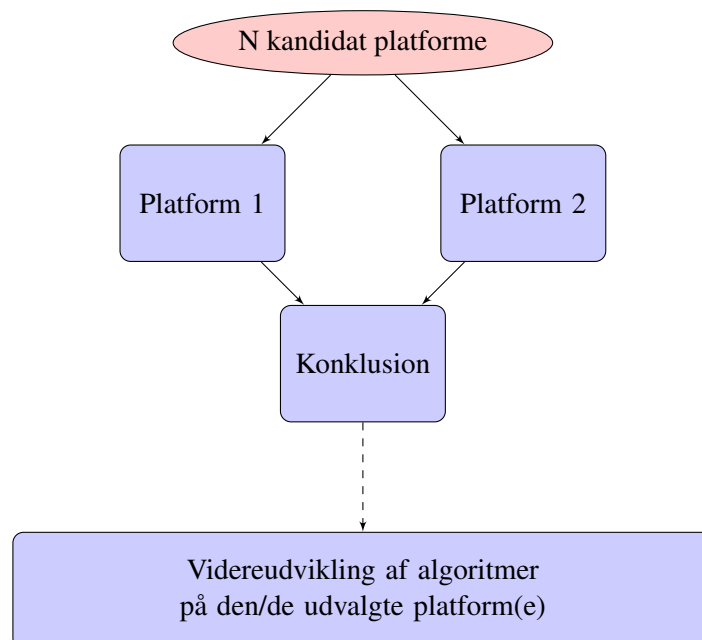
De analyse-mæssige resultater omfatter en liste af spørgsmål, som der ønskes svar på. Listen over spørgsmål er lang, men de vigtigste spørgsmål er:

- Hvordan er modellen repræsenteret?
- Hvilke former for kollisionstests ønskes udført på modellen?
- Er der særlige egenskaber ved problemet der kan udnyttes til optimering?
- Hvilke løsninger findes der i forvejen?
- Hvilke hardware platforme egner sig til acceleration af kollisionstest?
- Hvad er fordele/ulemper ved de enkelte platforme?

- Hvilke typer algoritmer er egnet til en given platform?
- Hvilken hastighedsforøgelse kan typisk forventes med en given platform?
- Hvorledes kan en given platform integreres i en samlet løsning?
- Hvilken færdiggørelsesgrad ønskes der af det endelige produkt?
- Hvordan skal det færdige produkt bruges?

Spørgsmålene på listen kan stort set opdeles i to: Hvilke platforme er interessante til acceleration af kollisionstests? Og hvorledes skal selve kollisionstestene udføres på en given platform? Dette leder naturligt frem til, at en del af specialet, skal bruges på at udvælge en eller to platforme, som er velegnet til det videre arbejde. I den kontekst ville det være hensigtsmæssigt, at bruge en del af specialet på at blive fortrolig med den eller de udvalgte platform/platforme, og tilhørende værktøjer.

På baggrund af ovenstående betragtninger er det blevet besluttet at lade specialet følge et forløb som illustreret på figur 1.5



Figur 1.5: Model for projektforsløb.

Som det ses på figuren, tager første del af specialet udgangspunkt i at der vælges en række platforme som mulige kandidater til det videre forløb. Disse kandidater gennemgår herefter en “bredde først” søgning, forstået på den måde, at deres egenskaber sammenlignes på et overordnet niveau. På baggrund af denne sammenligning udvælges to platforme, som skal gennemgå en nøjere undersøgelse. Denne undersøgelse består i at implementere et eller flere test problemer. Formålet med dette er at afdække platformenes egnethed, samt at lære platformene og deres værktøjer at kende. De test problemer der indgår i denne undersøgelse, skal naturligvis vælges med omhu, således at platformene testes i den rette kontekst.

Derefter tænkes det videre forløb at en eller begge platforme benyttes til at implementere det endelige produkt. Som sådan vil specialet efter udvælgelsen af platforme i hovedtræk komme til at handle

om, hvorledes platformen(e) kan udnyttes effektivt til at accelerere kollisionstest, herunder hvilke algoritmer der skal benyttes.

Del I
Teori

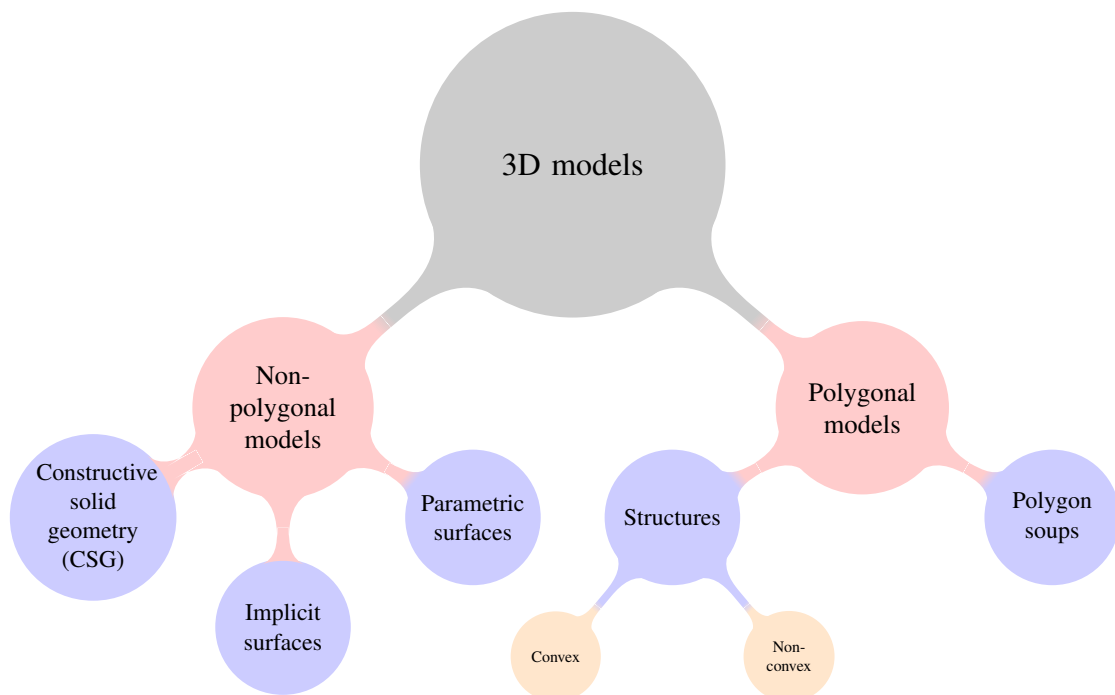
Kapitel 2

Terminologi/Baggrund

I dette kapitel vil den benyttede terminologi kort blive gennemgået. Målet med kapitlet er at give læseren en introduktion til emnet kollisionsdetektering, samt de tilhørende begreber og termer.

2.1 Model repræsentation

En model kan repræsenteres på et væld af forskellige måder. En muligt taksonomi over 3D model repræsentationer kan ses på figur 2.1. Denne opdeling stammer fra [22].



Figur 2.1: 3D model repræsentation.

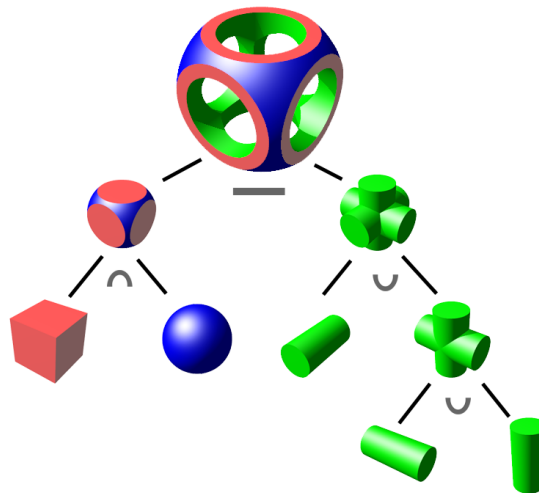
Overordnet set kan 3D modeller opdeles i modeller der er baseret på polygoner og modeller der ikke er baseret på polygoner. For overblikkets skyld følger her en kort beskrivelse af de enkelte repræsentationer.

2.1.1 Polygon baserede modeller

I en polygon baseret model, er modellen, naturligt nok, opbygget af en række polygoner. Polygonerne udgør typisk overfladen på de objekter der er repræsenteret af modellen. Når et objekt er repræsenteret af sin overflade, kan det være væsentligt om man kan skelne mellem indersiden og ydersiden af objektet, eller ej. Er modellen en såkaldt *polygon suppe*, er dette ikke muligt, idet denne kun indeholder information om positionen af de enkelte polygoner og intet andet. En *struktureret* model, derimod, inderholder yderligere information, der gør det muligt at skelne indersiden fra ydersiden af et objekt. Denne information består typisk af en normalvektor tilknyttet hver polygon. En *struktureret* model, kan yderligere opdeles i to typer: en *konveks* og en *konkav* (ikke konveks) type. Disse to typer beskriver blot om overfladerne kan være konkave eller om de kun kan være konvekse. Dette har betydning, idet mange typiske problemer bliver simplere, hvis overfladerne er konvekse, dette gælder eksempelvis kollisionstest. Det skal dog nævnes at alle konkave overflader kan opdeles i en række mindre konvekse overflader. Dette er dog en beregningsmæssig tung opgave, og er som sådan typisk en uønsket opgave. Endeligt skal det nævnes, at selvom polygonerne kan have vilkårligt mange kanter, så er de typisk opbygget af trekanter, da disse er lette at repræsentere og arbejde med, og har den egenskab, at en vilkårlig polygon kan opbygges af dem.

2.1.2 Modeller der ikke er baseret på polygoner

Der er naturligvis mange andre måder at beskrive en model på, end ved brug af polygoner. Af disse er den mest kendte nok *parametriske overflader*. Denne repræsentation beskriver en overflade ved hjælp af parameterfremstilling med to parametre. F.eks. kan $x - y$ planen beskrives som en parametrisk overflade som: $S : \mathbb{R}^2 \mapsto \mathbb{R}^3, S : (s, t) \mapsto (s, t, 0)$. Denne repræsentation betinger ikke at overfladen skal være lukket, og den er således egnet til at beskrive både åbne og lukkede objekter. Er der tale om en model bestående udelukkende af lukkede objekter kan det være en fordel at benytte en repræsentation der udnytter dette. *CSG* er et eksempel på en sådan repræsentation. I denne repræsentation beskrives objekter ved hjælp af en række 3D primitiver (f.eks. kugler, cylindre osv.) som sammensættes via operatorer fra mængdelæren (f.eks. fællesmængde, foreningsmængde osv.). Figur 2.2 viser et eksempel på hvorledes et komplekst objekt kan opbygges forholdsvis simpelt ved hjælp af CSG.



Figur 2.2: Eksempel på opbygning af kompleks objekt med CSG.

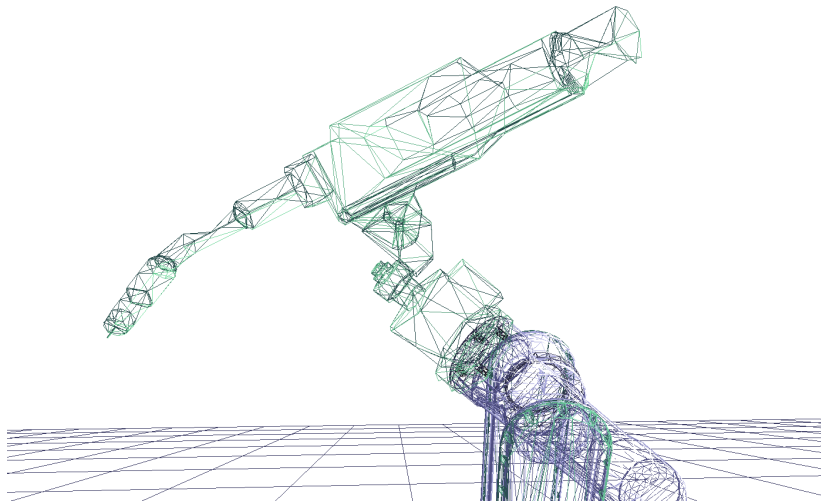
En anden måde at repræsentere modeller bestående af lukkede objekter, er med en såkaldt *implicit overflade*. Her beskrives et objekt med en funktion $f : \mathbb{R}^3 \mapsto \mathbb{R}$. Alle punkter der opfylder $f(x, y, z) < 0$ befinder sig på indersiden af objektet, ligeledes gælder for punkter der opfylder $f(x, y, z) > 0$ at de befinder sig udenfor objektet. Punkter der opfylder $f(x, y, z) = 0$ befinder sig netop på overfladen af objektet.

2.1.3 Anvendt modelrepræsentation

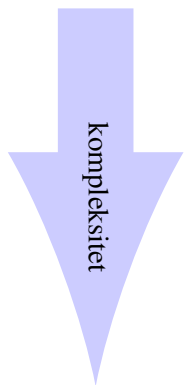
I det forliggende projekt er der benyttet en modelrepræsentation baseret på polygoner. Dette skyldes primært ydre omstændigheder, idet denne repræsentation i forvejen benyttes af robotgruppen. I spe-cialet benyttes en *polygon suppe*, hvor de enkelte polygoner består af trekanter. Dette gør det nemt at repræsentere modellen i form af datastrukturer, da antallet af kanter og noder er kendt på forhånd. Det er også enkelt at gemme data i et simpelt filformat. At modellen er opbygget af trekanter betyder at kollisionstestene på det laveste niveau består af overlap tests mellem to trekanter, et forholdsvis simpelt geometrisk problem, hvortil der findes flere velkendte løsninger. Et eksempel på en model kan ses på figur 2.3.

2.2 Forespørgelsestyper

Afhængig af hvilke informationer der ønskes, er der forskellige grader af kollisionsdetektering der kan udføres. Jo flere informationer man ønsker, jo mere krævende bliver forespørgelsen, i form af beregningskompleksitet. Derfor er det en fordel på forhånd at gøre sig klart, præcist hvilke spørgsmål der ønskes besvaret ved kollisionsdetekteringen. Herunder er de typiske spørgsmål, der kan søges svar på ved en kollisionsdetektering, opstillet i orden efter stigende kompleksitet:



Figur 2.3: Eksempel på model af robot med svejsepistol.



Givet to objekter A og B:

- Er A og B i kollision? (Kollisionstest/Collision test)
- Hvis A og B er i kollision, hvor kolliderer de så? (Bestemmelse af kontaktpunkt/Contact determination)
- Hvis A og B er adskilte, hvad er minimumsafstanden da imellem dem? (Separationsafstand/Minimum separation)
- Hvis A og B er i kollision, penetrerer A da B, eller omvendt? (Penetrationsdybde/Penetration depth)

Specialet fokuserer først og fremmest på at tilvejebringe resultatet af en kollisionstest så hurtigt som muligt. Men med tanke på anvendelsen af den udviklede software, vil det også være nødvendigt at se på hvorledes oplysninger om eksempelvis kontaktpunkter kan fremskaffes.

2.3 Hvad er en kollision?

Et relevant spørgsmål at stille sig i forbindelse med kollisiondetektion er hvordan en kollision defineres. Hvad skal der til før der er tale om en kollision, og er der specifikke grænsetilfælde der skal tages højde for? I de følgende afsnit behandles disse spørgsmål for kollision i hhv. 2 dimensioner og 3 dimensioner. Der tages udgangspunkt i kollision imellem primitiver (trekanter), idet det er imellem primitiver den endelige test skal afgøre om der er tale om kollision eller ej.

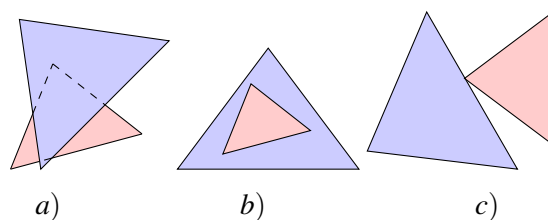
2.3.1 Kollision i 2 dimensioner

Kollision i 2D kan forekomme når de to trekanter ligger i samme plan, og er som sådan et special tilfælde af kollision i 3 dimensioner.

Generelt kan kollisions betingelsen i 2D udtrykkes matematisk som følger:

Definition 2.3.1 Givet et sæt af alle punkter (også indre punkter) indeholdt i trekant T_1 , defineret som $S_1 = \{p \in \mathbb{R}^2 \mid p \in T_1\}$, og et tilsvarende sæt af punkter for T_2 , defineret som $S_2 = \{q \in \mathbb{R}^2 \mid q \in T_2\}$. Da er der tale om kollision hvis, og kun hvis, betingelsen $S_1 \cap S_2 \neq \emptyset$ er opfyldt. Ellers er de to sæt disjunkte, og der kan ikke være tale om kollision.

Herunder følger eksempler på de forskellige kollisioner der kan forekomme i 2D. Når en eller flere kanter fra en trekant skære en eller flere kanter fra en anden trekant er der helt åbenlyst tale om kollision. Se eksempel a) på figur 2.4.



Figur 2.4: Eksempler på kollision i 2D.

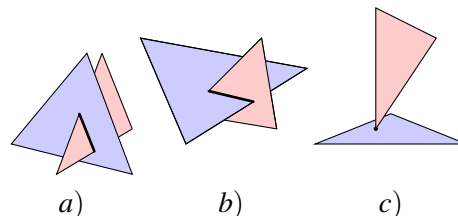
Eksempel b) på figur 2.4 viser et andet eksempel på en åbenlys kollision, nemlig det tilfælde hvor den ene trekant ligger fuldstændig inden i den anden trekant.

Eksempel c) viser situationene hvor et hjørne fra en trekant rører en kant fra den anden trekant, dette tilfælde udgør naturligvis også en kollision. Et special tilfælde af dette er, når de to trekanter har et hjørne tilfælles.

2.3.2 Kollision i 3 dimensioner

Betingelsen for kollision i 3D kan udtrykkes på stort set samme måde som i 2D:

Definition 2.3.2 Givet et sæt af alle punkter (også indre punkter) indeholdt i trekant T_1 , defineret som $S_1 = \{p \in \mathbb{R}^3 \mid p \in T_1\}$, og et tilsvarende sæt af punkter for T_2 , defineret som $S_2 = \{q \in \mathbb{R}^3 \mid q \in T_2\}$. Da er der tale om kollision hvis, og kun hvis, betingelsen $S_1 \cap S_2 \neq \emptyset$ er opfyldt. Ellers er de to sæt disjunkte og der kan ikke være tale om kollision.



Figur 2.5: Eksempler på kollision i 3D.

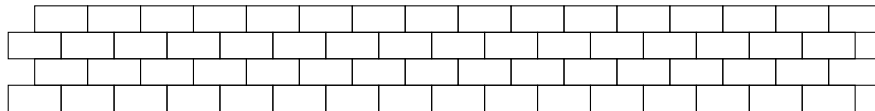
Figur 2.5 viser nogle eksempler på kollision i 3 dimensioner. Eksempel *a*) viser hvorledes den ene trekant penetrerer den anden trekants indre. Et lignende eksempel kan ses i *b*), her er det dog kun en kant der går igennem den anden trekant. Endelig viser eksempel *c*) hvorledes et hjørne fra den ene trekant lige akkurat rører den anden trekants indre. Denne situation har også et par special tilfælde, nemlig hjørne-til-hjørne og hjørne-til-kant. Yderligere er der også tilfældet hvor de to trekanters kanter berører hinanden.

Kapitel 3

Arkitekture og arkitektur valg

3.1 Parallel processing

For at dække koncepter og problemer i forbindelse med parallel processing, inspireret af [7], vil opmuringen af en murstensvæg af et hold murere blive brugt som eksempel.

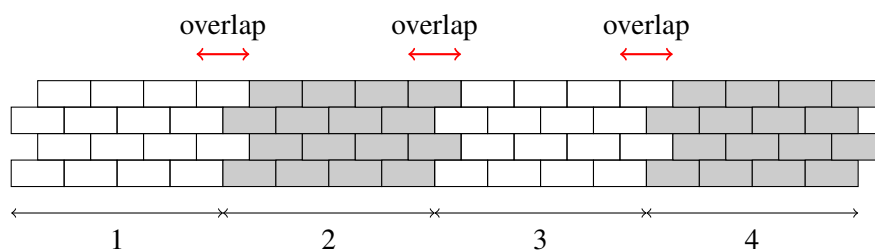


Figur 3.1: Problemdomænet

Generelt ønsker vi at udføre en algoritme på et stort datasæt. Vi siger at vi former et *domæne* med mange *members*

3.1.1 Domæne dekomposition

Givet et problem af en vis størrelse falder det naturligt at dele arbejdet op i flere delproblemer og løse dem individuelt. En vertikal dekomposition af væggen er vist på figur 3.2



Figur 3.2: Vertikal dekomposition af problemdomænet i fire dele.

Generelt, kan effektiviteten karakteriseres ved

$$\epsilon = \frac{S}{N} \tag{3.1}$$

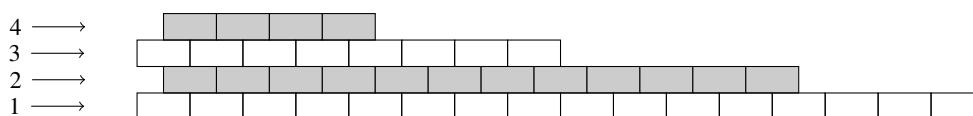
for N murere der bygger en mur S gange hurtigere end en enkelt murer.

Ved at sætte 4 murere til at arbejde på væggen forventes det umiddelbart at det vil tage en fjerdedel af tiden for at færdiggøre murerarbejdet ($\epsilon = 1$). Imidlertid vil områderne, hvor murstenene overlapper hinanden kræve at murerne samarbejde om at lægge stenene. Denne koordination kræver en vis mængde kommunikation der så at sige, forstyrrer murerne og effektiviteten falder dermed.

For at opnå en høj effektivitet, er det derfor nødvendigt at reducere tiden der anvendes på kommunikation i forhold til tiden der bruges på at udføre arbejde. Dette kan gøres ved at reducere tiden det tager for at samarbejde om et overlappende stykke (f.eks. ved at have en effektiv fremgangsmåde for, hvem der lægger først osv.), eller ved relativt set at gøre tiden der bruges på kommunikation negligerlig i forhold til arbejdstiden.

Dette bringer os frem til en af de første regler i parallel processing: Problemdomænet skal være af en vis størrelse, før at overheadet i at behandle eventuelle kantkonditioner ikke vil dominere.

Ovenstående dekomposition er en af mange mulige. F.eks. kan der også dekomponeres horisontalt som vist på figur 3.3

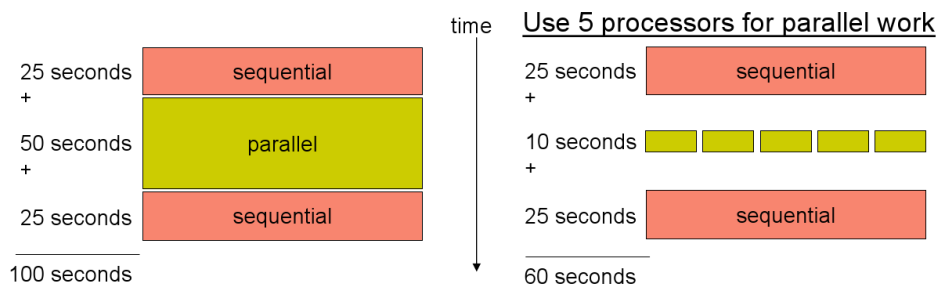


Figur 3.3: Horisontal dekomposition af problemdomænet.

En horisontal dekomposition giver imidlertid problemer, da en murer på et lag ikke kan lægge en sten før laget nedenunder kan understøtte den. Dette er en constraint som indskrænker friheden for den enkelte murer til at arbejde selvstændigt. For den viste dekomposition vil en pipeline kunne give en vis parallelisme, men der vil ikke kunne opdeles i mere end antallet af lag mursten. For en meget lang mur vil det derfor være mere hensigtsmæssigt at dele vertikalt.

For det simple problemdomæne som der indtil videre har været vist, har den jævne struktur af arbejdet gjort at det var ligetil at give den samme mængde arbejde til hver murer.

3.1.2 Parallelitet



Figur 3.4: Illustration af Amdahl's lov.

Graden af speedup der kan forventes af paralleliseringen af et problem afhænger af hvor meget indbygget serialitet der ikke kan omgås. Dette er udtrykt som Amdahl's lov, illustreret på figur 3.4.

$$S = \frac{1}{T_s + 1/N(1 - T_s)} \quad (3.2)$$

hvor S er speedup, T_s er den tid der anvendes på seriel processering og N er andelen der kan paralleliseres.

3.2 Moderne parallelle arkitekture

I det følgende afsnit gives en kort introduktion til en række af de platforme der har været overvejet i forbindelse med dette speciale.

3.2.1 Multi-core PC

Efterhånden som chipproducenterne er begyndt at indse at drømmen om processorer med clock frekvenser på 10 GHz eller derover, drukner i varme og memory latency, er der fremkommet flere og flere processorer med flere kerner. Indtil videre er der tale om processorer med omkring to til fire kerner. Men fremtiden peger mod processorer med mange flere kerner. I PC verdenen, der er bundet af bagud kompatibilitets krav, har man valgt en homogen model for disse processorer. Det vil sige at alle kerne er ens og lige. Hver kerne er så at sige en mere eller mindre „fuld“ PC processor, der eventuelt deler cache med de andre kerner. Dette har en række fordele (bagud kompatibilitet, enklere design mm.), men også en række ulemper. En af ulemper er netop at kernerne blot er PC processorer, med alt hvad det indebærer, herunder cache, branch speculation, høj memory latency osv. Alle disse ting fører til at processorerne er udemærkede allround enheder, men de er f.eks ikke optimale til tunge beregninger. Dette skyldes at de i høj grad er svære at programmere med et deterministisk resultat til følge. Der er dog igen tvivl om at vi kommer til at se en spændende udvikling inden for multi-core PC'er i de kommende år.

3.2.2 Clusters

I forlængelse af ovenstående er det naturligt at se på clusters af PC'er. Disse består af et netværk af PC'er, ofte op til flere tusinde PC'er. Tilsammen byder sådan et cluster på en enorm regnekraft, og sådanne cluster benyttes da også flittigt i f.eks. videnskabelige sammenhænge. Da noderne (de enkelte PC'er) typisk er koblet sammen ved hjælp af standard netværksteknologi (bla. for at holde prisen ned), er et cluster typisk karakteriseret ved en høj regnekraft i de enkelte noder, men en forholdsvis lav båndbredde imellem de noderne. Derfor skal problemer der løses af clusteret også helst være løst koblet, således at de enkelte noder kan stå og regne uden at kommunikere alt for meget med de andre noder.

3.2.3 GPU

I den seneste tid er flere grafikkortproducenter begyndt at åbne op omkring deres teknologi, således det bliver muligt at udnytte det potentiale der ligger i moderne grafikkort. Dette sker som en konsekvens af at moderne grafikkort er blevet mere generelle i deres opbygning. De kan derfor benyttes til en række generelle problemstillinger. De fleste højtydende grafikkort er opbygget som array af mere eller mindre generelle processorer. Dette gør dem velegnet til en række beregningstunge opgaver. Ydermere har et sådan grafikkort typisk en meget høj båndbredde til sin hukommelse, hvilket yderligere markerer grafikkortet som en potent regneenhed. Kombineret med en kraftig PC, kan et grafik kort altså benyttes som en kraftig co-processor, der kan tage sig af beregningstunge opgaver. Grafikkortet har dog den ulempe at de enkelte processorelementer har en masse restriktioner på hvordan de kan arbejde, og det kan derfor være svært at udnytte deres fulde potentiale.

3.2.4 CELL/BE

CELL processoren er en ny processor der er udviklet til bla. den nye PlayStation 3 spillekonsol fra Sony. Denne processor er en såkaldt heterogen processor, der består af en generel hovedprocessor og et antal specialiserede „slave“ processorer. Alle processorene er bundet sammen af en højtydende bus. Slave processorer er udformet så de har en meget høj ydelse i single precision floating point, bla. understøtter de vektor operationer. Tanken er at hovedprocessoren uddelegerer tunge beregningsmæssige opgaver til slave processorerne. Slaveprocessorene indeholder desuden deres egen hukommelse der arbejder ved fuld clock frekvens, de har derfor ingen behov for cache. Dermed bliver det lettere at skrive programmer med en deterministisk opførelse. Alt i alt forssøger CELL processoren at overkomme mange af de begrænsninger som traditionelle fler-kerne processorer har.

3.2.5 FPGA

Der har igennem en årrække eksisteret en del forskellige typer af programmerbar logik, og FPGA'erne er nok en af de mest fleksible af slagsen. En FPGA programmeres ikke i traditionel forstand, istedet beskrives, mere eller mindre indirekte, det logiske kredsløb der ønskes. Det betyder at et kredsløb med en helt specifik funktion kan opbygges. Det siger sig selv at hvis man kan skræddersy det logiske kredsløb til at udføre en helt specifik funktion, så kan man optimere dette kredsløb langt mere, end hvis der skulle anvendes en allround processor til den samme funktion. Derudover er det muligt at ligge et stort antal af sådanne logiske blokke på en chip, hvilket giver mulighed for en stor grad af parallelitet. Idag fåes desuden FPGA'er med indbyggede multiplier blokke og lignende, der gør dem endnu mere kraftfulde og lettere at udvikle til. Ulempen ved FPGA'erne er at de er mere udfordrende at udvikle til, og kræver mere low level viden, end traditionel udvikling.

3.3 Valg af arkitekturer

Det var oprindeligt planen at projektgruppen ville arbejde med en FPGA platform i projektet. Men da JAJ allerede arbejder på den platform, blev det besluttet at se på hvilke andre hardware platforme der kunne være interessante at benytte til projektet. Det stod ret hurtigt klart at udvalget af platforme der var praktisk anvendelige i projektet ikke var ret stort. Men på to områder har der være en del spændende tiltag på det seneste. Begge områder udspringer mere eller mindre direkte af spil industrien. I det ene tilfælde er der tale om den nye generation af grafikkort. Disse kort er uhyre kraftfulde, men har tidligere været hæmmet af mangel på udviklingsværktøjer, så det har været svært at udnytte kortene til andet end grafik. Denne udvikling er ved at vende, idet de store grafikkort producenter er begyndt at udvikle disse værktøjer, samtidig med at de tilpasser grafikkortene til mere generel brug. Det andet tilfælde er den meget omtalte Cell processor, som blandt andet benyttes i Sony's Playstation 3. Denne processor adskiller sig på en række områder fra traditionelle processorer. Den har en ekstrem høj ydelse og er samtidig meget power effektiv. Desuden har den et meget omfattende sæt af udviklingsværktøjer.

Begge disse platforme egner sig umiddelbart til projektet. Projektgruppen har derfor valgt at kigge nærmere på begge platforme.

Kapitel 4

Algoritmer

4.1 Kollisionsdetektion

4.2 Generelt

I situationer, hvor modeller bestående af mange tusinde trekanter skal testes for indbyrdes kollision, skal der udføres et meget stort antal tests. For en udtømmende test af alle trekanter i to trekantsupper, er kompleksiteten kvadratisk i antallet af trekanter.

I dette speciale angribes problemet på to måder

- Hierarkier af bounding volumes til at reducere antallet af trekanttests.
- Effektive trekant og bounding volume tests.

Dette afsnit beskriver teorien omkring hierarkiske metoder, trekant-trekant tests og OBB-OBB tests.

4.2.1 Operationer

I de efterfølgende afsnit vil algoritmerne blive analyseret på basis af det antal basale operationer (addition, subtraktion, multiplikation, division) som de kræver. Som udgangspunkt er følgende tabel med vektoroperationer blevet anvendt.

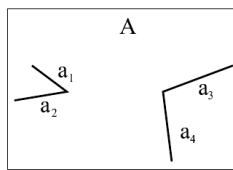
Operation	add	sub	mul	div
ADD_3D	3			
SUB_3D		3		
CROSS_3D		3	6	
DOT_3D	2		3	
TRIPLESCALAR_3D	2	3	9	

Tabel 4.1: Basale operationer i vektoroperationer

4.3 Hierarkiske metoder

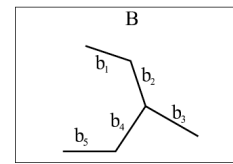
Figur 4.1 og 4.2 viser to polygonsupper A og B, med henholdsvis 4 og 5 polygoner. En udtømmende test der tester alle polygoner fra objekt A imod alle polygoner fra objekt B, kræver 20 polygon-

polygon tests. For modeller af denne størrelsesorden er det ikke voldsomt, men p.g.a. af den kvadratiske kompleksitet bliver antallet af tests urealistisk højt ved større modeller.



Model A

Figur 4.1: Objekt A



Model B

Figur 4.2: Objekt B

Brug af f.eks. bounding volumes (BV) til at dække hvert enkel polygon vil ganske givet hjælpe på den tid det tager at lave en enkelt test, men den asymptotiske køretid forbliver uændret.

Bounding volumes kommer til deres ret når de konstrueres til at dække grupper af polygoner og efterfølgende arrangeres i en træstruktur kaldet et bounding volume hieraki (BVH). Denne træstruktur gør det muligt hurtigt at udelukke grupper af polygoner fra kollision, hvilket kan give en drastisk reduktion af antal tests.

Følgende er en kort gennemgang af hierarkiske metoder og bounding volumes. For en mere generel diskussion omkring opbygning af forskellige træer henvises der til [6] og [9]. En række af illustrationerne i dette kapitel er gengivet fra disse kilder.

I det efterfølgende vil først opbygningen af BVH'er og dernæst deres anvendelse i kollisionsdetektion blive beskrevet.

4.3.1 Partitionering

Figur 4.3 viser en mulig hierarkisk opdeling af de to polygongrupper A og B fra henholdsvis figur 4.1 og 4.2. Hvert polygon er på leaf-niveau dækket af en BV. Disse BV's er så samlet i større grupper og dækket af nye BV der indeholder alle de underliggende polygoner. Dette fortsætter op igennem træet indtil at top-niveauet nås med kun en BV der indeholder samtlige polygoner.

Ovenstående beskrivelse ligger op til en bottom-up opbygning af træet. Ofte er det dog lettere at foretage det top-down ([6]). I top-down startes der fra en enkelt stor top-niveau node, der så deles op i gradvist mindre bidder indtil at leaf-niveauet nås.

Træet for den beskrevne opdeling af objekt A og B kan ses i figur 4.4.

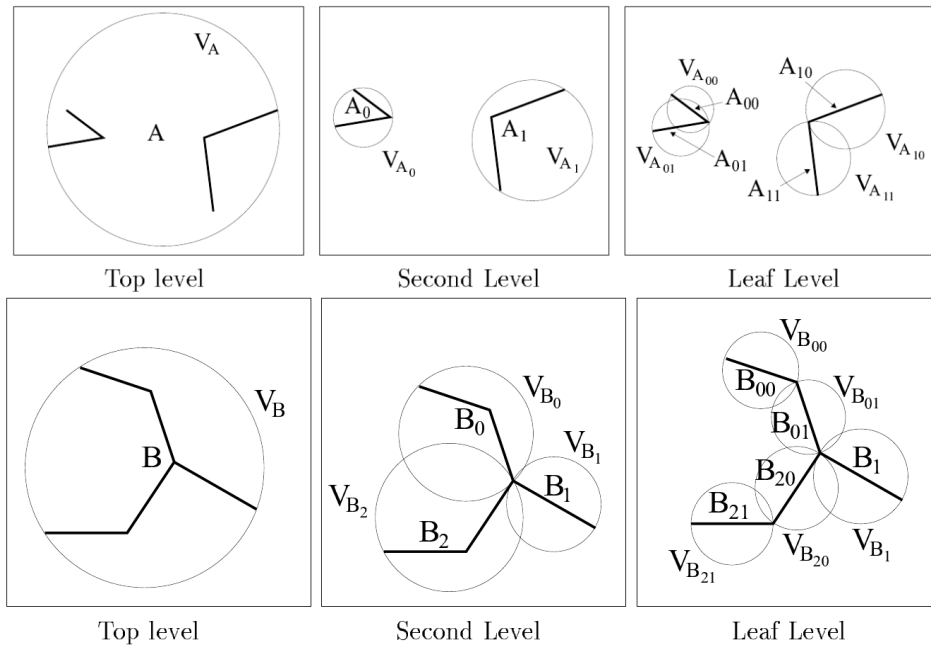
Generelt for både bottom-up og top-down gælder det at

- Top-noden indeholder hele polygongruppen
- Leaf-noderne indeholder en polygon hver
- Hver node indeholder alle polygoner som børnene indeholder

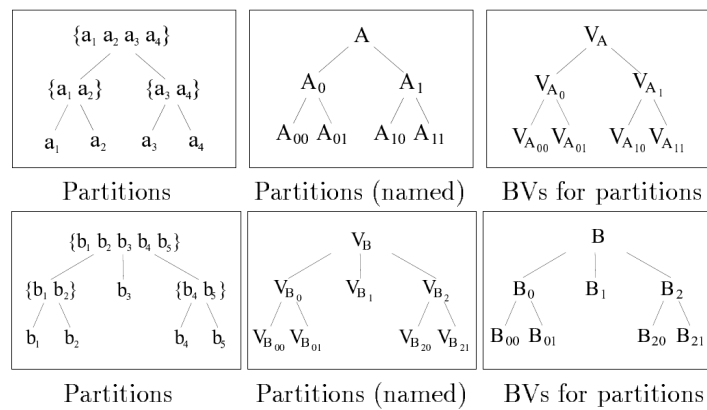
Ud fra dette er der mange parametre der skal tages stilling til heriblandt valget af BV, graden af træet (binær, quad, oct mm.), balanceringen mm.

4.3.2 Kollisionsdetektion med BVH

Som tidligere nævnt vil en udtømmende kollisionstest af objekt A mod B kræve 20 trekant-trekant tests.



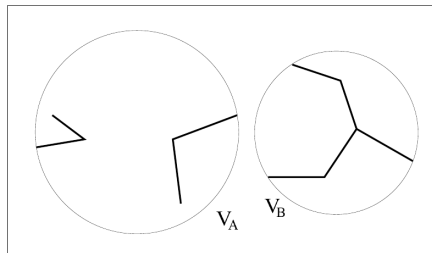
Figur 4.3: Niveauerne i hierarkierne for objekt A og B



Figur 4.4: Træerne for objekt A og B

Hvis der derimod udføres en test af det øverste niveau i deres respektive træer, fås der et konservativt svar, da deres BV er en grov approksimation der er garanteret til at dække objektet fuldstændigt (se afsnit 4.3.3). Testen afgør hurtigt at deres BV ikke kolliderer, hvis objekterne er tilstrækkelig separerede.

Hvis test af det øverste niveau giver et positivt resultat (kollision), gås der længere ned i træet og udføres test på de umiddelbart underliggende noder. Disse vil per konstruktion af træet have et bedre fit til de underliggende polygoner, og giver dermed et mere præcist resultat af kollisionstesten. Dette er illustreret på figur 4.5, med de to objekter A og B, hvor de øverste BV for deres hierarkier er separerede.



Figur 4.5: Øverste niveauer i objekt A og B's træer. Deres øverste BV's er separerede og der er derfor heller ikke kollision mellem de underliggende BV's og polygoner i træet.

Hvis de underliggende polygoner er i kollision er det dog nødvendigt at udføre tests helt indtil leaf-noderne i træet nås.

Besparselsen ved at anvende bounding volume hierarkier er størst når objekterne har stor separation, da kun få noder i træet skal testes. Men selv ved kollisioner imellem polygonerne er det (alt afhængig af de anvendte BV's fit) værd at anvende træet, på trods af overheadet ved at teste BV's. Træet giver nemlig en spatiel opdeling af objektet, så hvis at f.eks. den ene ende af objektet er i kollision mens den anden ende ikke er, vil en masse overflødige tests stadig kunne undgås.

Antallet af BV tests der skal udføres er afhængigt af mange faktorer, bl.a. BV fit, træets grad (binært, quad, osv.), antal kontaktpunkter og den valgte strategi for træ-traversering. For balancerede træer vil det generelt være logaritmisk i antallet af polygon-polygon tests der skal udføres.

For en nærmere diskussion henvises der til [9] kapitel 2 og [6] kapitel 6.

Træ-traversering

For at afgøre kollision imellem to træer af bounding volumes er det nødvendigt at have en strategi for hvilken rækkefølge tests imellem BV udføres. Problemet kan deles op (som [6] kap. 6) i

- Hvordan traverseres det enkelte træ?
- Hvilket træ nedstiges først?

Traversering af en træstruktur, hvor hver node besøges en enkelt gang er kendt fra mange andre sammenhænge. I forbindelse med kollisiondetektion er de mest relevante dybde-først (DFS) og bredde-først (BFS) søgning samt bedste-først. DFS udforsker den nuværende nodes børn i dybden indtil leaf-niveauet nås før at der gås videre med noder på samme niveau. BFS derimod udforsker alle noder på samme niveau før at der gås videre med børnene. BFS er dog ofte urealistisk at bruge da

køen med noder vil kunne risikere at indeholde op til n^2 noder. DFS derimod vil kun kræve plads på stacken til højden af det største træ, og er derfor den mest anvendte strategi ([6]). Hvor DFS og BFS er blinde metoder i den forstand at de ikke kigger på nodernes data for at afgøre traverseringen, har en bedste-først søgninger en heuristik der ud fra nodernes indhold, f.eks. position og størrelse, placerer en node på en prioritetskø. Selvom dette kan give et bedre bud på hvilken node der er velegnet at udforske på det givne tidspunkt, kan kriteriet være vanskeligt at finde og styringen af køen så dyr at det ikke kan betale sig ([6]).

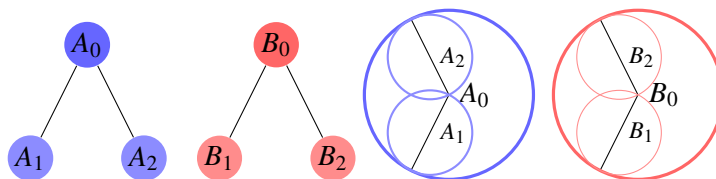
Strategien for nedstigningen i træerne bestemmer fra hvilket træ nye child-nodes skal hentes, hvis der er kollision. Der findes mange forskellige strategier (se [6] kap 6.3.1) men de mest interessante for nedstigning i to træer, A og B er

1. Nedstig i træet med den BV der har det største volume
2. Samtidig nedstigning af træ A og B
3. Skift imellem træ A og B

I den første strategi laves der en sammenligning af de to nuværende BV's størrelser, og det træ der har den største er det som der gås ned i. Den store fordel er at søgerummet hele tiden indskrænkes mest muligt. Dette giver en fordel i scener, hvor der er store grove objekter og samtidig fine detaljer, som f.eks. møtrikker på et stort rør. Prisen er at der skal bruges tid på denne sammenligning og at der skal tilgås mere data i hukommelsen.

Samtidig nedstigning går som navnet antyder ned i begge træer samtidig. Dermed gås der hurtigst muligt ned i træet. Dette kræver ikke et evalueringskriterie og gør strategien til en af de hurtigste. Samtidig dannes der hurtigt meget arbejde hvilket er interessant i parallel processering. Ulempen er at i forhold til en volumen-bestemt strategi, vil søgerummet ikke nødvendigvis blive indskrænket maksimalt. Dette er dog stærkt afhængig af objekternes træer. Strategien er vist på figur 4.7:

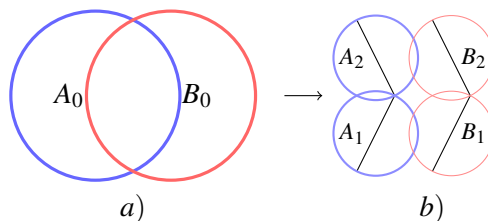
I den tredje strategi skiftes der hele tiden imellem at gå ned i det ene og det andet træ. Dette er vist på figur 4.8. I forhold til den samtidige nedstigning vil der blive udført flere tests hvis at der skal søges helt til bunden af træet. Der kan dog sagtens forekomme at denne strategi udfører færre tests (f.eks. ville en lidt større separation af objekterne i figur 4.8 give *b*) mulighed for at hoppe ud), men generelt er det svært at afgøre. En fordel ved strategien er at der i forhold til den samtidige nedstigning spares transformationer, da der hele tiden beregnes en enkelt node imod en række andre.



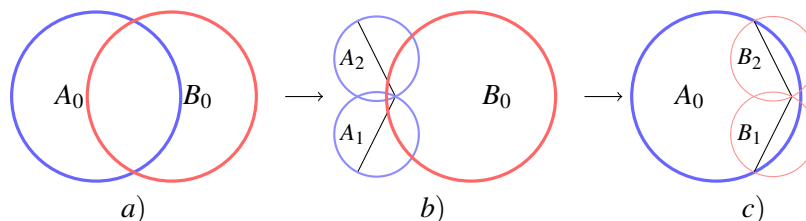
Figur 4.6: Objekttræer og deres tilhørende bounding volumes.

4.3.3 Bounding volumes

En bounding volume (BV) er en approksimation af et mere komplekst objekt (f.eks. en trekantsuppe) der rumligt omslutter det fuldstændigt. I forbindelse med kollisionsdetektion er BV's særdeles nyttige. For to BV's der komplet omslutter hver deres trekantsuppe, kan trekantene ikke være i kollision med



Figur 4.7: Samtidig traversering: I *a)* er BV'erne A_0 og B_0 er i kollision så i *b)* bliver A_1, A_2 testet imod B_1, B_2



Figur 4.8: Skiftende traversering: BV'erne A_0 og B_0 er i kollision i *a)*. A_0 testes først imod B_1 og B_2 (*b)*). Da disse også er i kollision skiftes der træ og A_1 og A_2 testes imod B_0 i *c)*.

mindre at BV'erne også er i kollision. Omvendt, hvis de to BV's er i kollision er flere tests nødvendige. Ved at udføre tests på BV'erne først, kan man på denne måde voldsomt reducere antallet af trekant-trekanter tests der skal udføres i en scene.

Typer

I valget af BV til en kollisionstest algoritme er der en række kvaliteter som ønskes. [6] fremhæver blandt andet

- Billig kollisionstest
- Bedre fit af underliggende trekanten
- Let at transformere under kørsel
- Lille hukommelsesforbrug

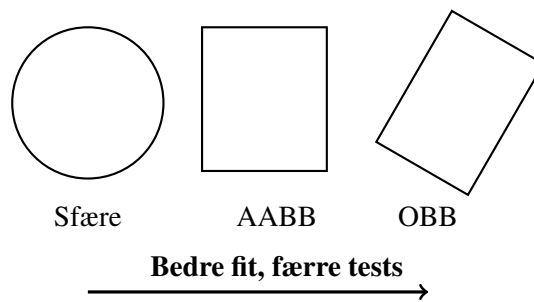
Simple BV geometrier er billige at udføre tests på men vil omvendt ikke kunne slutte så tæt til den underliggende geometri. Dette vil gøre at der oftere skal udføres test på trekantene, når objekterne er i nær-kollision. Figur 4.9 viser de mest brugte BV typer.

I en scene hvor objekterne roterer er det for nogle BV geometrier nødvendigt at refitte dem til de underliggende trekanten for hvert enkelt tidsskridt. Dette giver et betragteligt overhead, hvis at beregningen af nye BV's er dyr.

Som BV er der en lang række af volumener der typisk anvendes ([9], [6]), bl.a. sfærer, axis-aligned bounding boxes (AABB), oriented bounding boxes (OBB), k-direction discrete orientation polytoper (k-DOPs) og convex hulls.

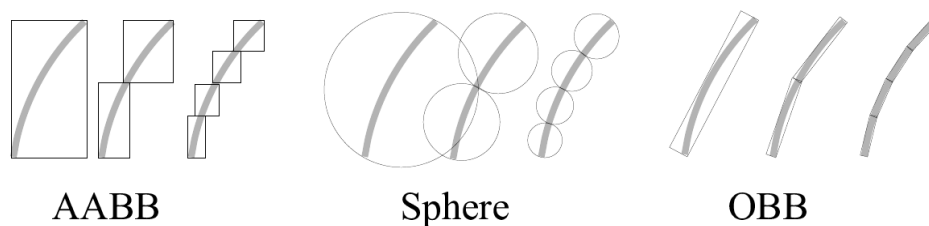
Figur 4.10 viser hvordan typen af BV kan have indflydelse på, hvor godt der approksimeres til den underliggende geometri. Det kan ses at OBB'er pga. deres variable rotation har langt større mulighed

Billigere kollisionstest, mindre hukommelsesforbrug



Figur 4.9: Typer af bounding volumes, sfære, axis-aligned bounding box (AABB) og oriented bounding box (OBB).

end sfærer og AABB for at fitte bedre i generel position. Et bedre fit vil i nær-kollision gøre at der skal traverseres en mindre del af træet. [8] og [9] indeholder en evaluering af AABB'er imod OBB'er. Selvom resultatet ikke er entydigt ses der i visse situationer op til en størrelsesorden mindre BV tests når der anvendes OBB'er frem for AABB.



Figur 4.10: Fit af forskellige bounding volumes

4.4 Forslag til metoder

Der findes mange måder at implementere kollisions test på. De hierarkiske metoder har vundet indpas, fordi de fleste tilfælde kan mindske mængden af arbejde der skal udføres ganske betragteligt. Desværre indfører de også en seriel afhængighed i kraft af deres hierarkiske natur. Denne afhængighed er uønsket når en algoritme skal paralleliseres. I dette speciale har fokus været på at implementere en parallel udgave af en hierarkisk metode. Undervejs har der dog også været tænkt alternative tanker om hvordan en kollisions test algoritme til parallel hardware kunne se ud. Et par af dem vil blive præsenteret her.

4.4.1 Pseudo hierarkisk (baseret på Quinlan)

Grundideen i denne algoritme er at mindske den serielle afhængighed ved at gøre hierarkiet lavere. Metoden benytter sfærer som bounding volumes, men istedet for at opbygge et fuldt træ, benyttes kun de nederste par lag i træet. På denne måde opnåes det et stor antal tests fra starten af. Samtidig er de enkelte test meget billige. Metoden har yderligere den fordel at den er forholdsvis let at implementere.

Naturligvis har metoden også ulemper, herunder at sfærer ikke fitter specielt godt, og der kan derfor blive udført en del unødvendigt arbejde.

4.4.2 Æther metode

Æther metoden har sit navn i mangel på et bedre navn. I denne metode lægges et grid ind i scenen. I starten har alle grid punkter værdien nul. Alle hjørner på trekantene „farver“ herefter de grid punkter der ligger inden for en vis radius af hjørnet. Farvningen sker ved at et hjørnepunkt skriver sit objekt id (id på det objekt det tilhører) ind i de punkter det kan „se“ Hvis et hjørne under denne farvning støder på et grid punkt med en værdi forskellig fra nul, tilføjes trekanten til en liste over trekanter der skal tjekkes. Selve farvningen forgår et objekt af gangen, og det objekt der startes med skal igennem en ekstra tur til sidst. Ellers vil der ikke blive tilføjet trekanter fra dette objekt. Alternativt tilføjes alle trekanter fra dette objekt til listen over trekanter der skal tjekkes. Efter farvningen køres der massiv trekant test på listen over kandidat trekanter.

4.5 Segment-piercing

Algoritmen der her præsenteres er baseret på ideer og primitiv-primitiv algoritmer fra kapitel 5 i *Real-Time Collision Detection* [6].

Algoritmen blev valgt fordi den var let forståelig, let at fejlfinde på (pga. dens opbygning af primitiv-tests) og samtidig numerisk robust. Den hører ikke til de hurtigste algoritmer på en traditionel CPU, men på parallel platformen kan den dog have fordel i at have få branches.

Ideen til den overordnede trekant-trekant algoritme er fra s. 172-173 kap 5.2.10, segment-plan algoritmen fra kap. 5.3.1 s. 175-177, linie-trekant fra 5.3.4 s. 184-187 og segment-trekant fra kap. 5.3.4 s. 188.

4.5.1 Princip

Når to trekanter kolliderer kan det overordnet set ske på to måder: To kanter fra den ene trekant skærer den anden trekants indre eller én kant fra hver trekant skærer den andens (se figur 4.11).

En kollision kan dermed detekteres ved at foretage 2×3 segment-trekant tests, tre for hver trekants kanter imod den anden trekant. Hvis mindst én af de seks tests giver et positivt resultat, er trekanterne i kollision. Hvis alle seks tests er negative skærer trekanterne ikke hinanden. Dette giver anledning til en ligefrem algoritme som efterfølgende vil blive omtalt som 'segment-pierce' algoritmen.

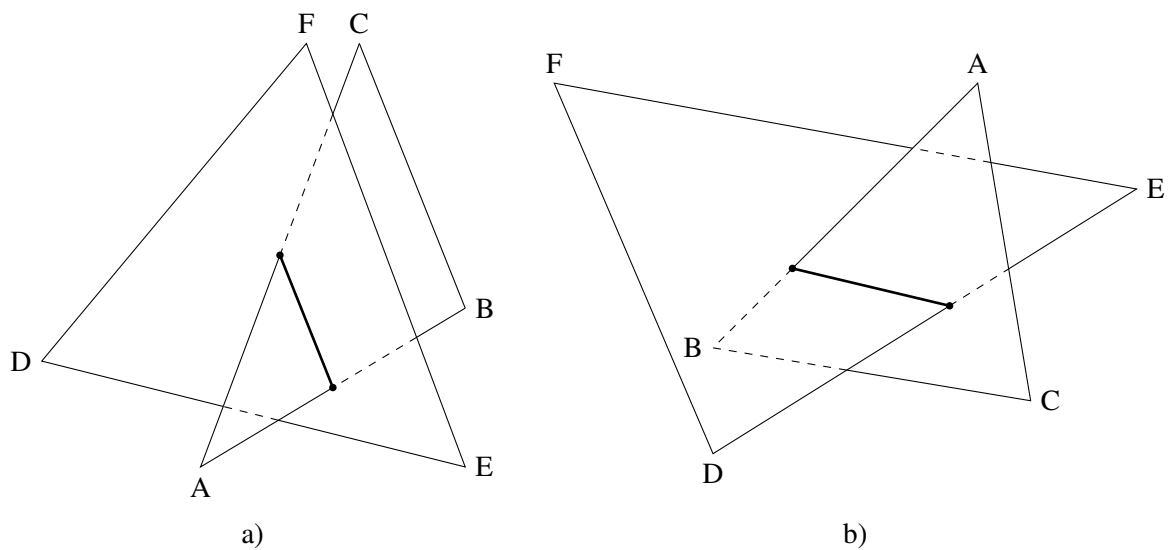
Segment-pierce algoritmen baserer sig på segment-trekant tests implementeret vha.

- Linie-trekant test
- Segment-plan test

Disse samt deres anvendelse i segment-trekant testen vil blive beskrevet efterfølgende.

Linie-trekant

Givet en trekant ABC og en linie igennem P og Q begge i \mathbb{R}^3 kan det afgøres om PQ skærer ABC ved at afgøre om skæringspunktet R imellem PQ og ABC 's plan ligger inden for ABC 's kanter. Denne test kan gennemføres ved eksplicit at bestemme R og udføre en punkt-trekant 'sidedness test' der afgør



Figur 4.11: To trekanter skærer hinanden hvis de: a) to kanter fra den ene trekant skærer eller b) kun én kant skærer.

om punktet ligger inden for trekantens sider. Denne fremgangsmåde har imidlertid problemer mht. numerisk robusthed (For en nærmere diskussion se [6, kap. 11.3.3]).

Istedet anvendes testen fra [6, kap. 5.3.4], hvor der direkte ses på liniens placering i forhold til trekantens sider. Dette gøres ved at udregne et skalar trippelprodukt for hver side i trekanten og se på dets fortegn. Geometrisk set udregner de tre skalar trippelprodukter¹

$$u = PQ \cdot (PC \times PB)$$

$$v = PQ \cdot (PA \times PC)$$

$$w = PQ \cdot (PB \times PA)$$

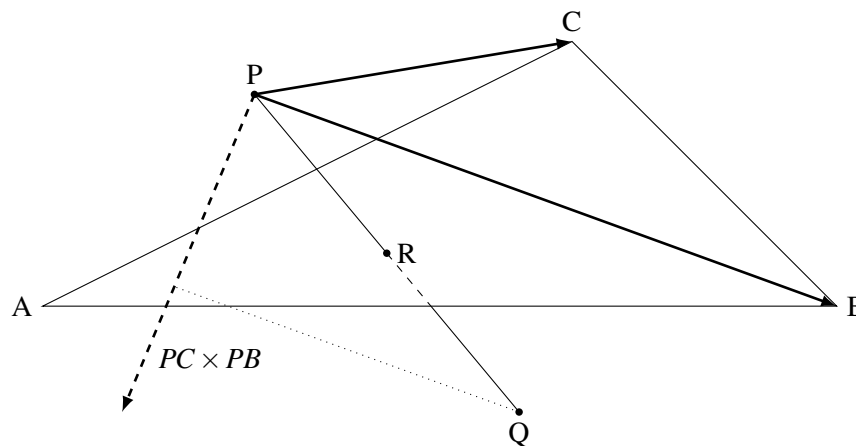
volumenelementet (med fortegn) af parallelepipedet der dannes af de tre givne vektorer. Fortegnet kan siges at give orienteringen af volumenelementet. Ved at se på deres fortegn kan det afgøres til hvilken side at linien PQ skærer trekant ABC .

På figur 4.12 er det vist hvordan at krydsproduktet $(PC \times PB)$ i trippelproduktet formes af siden BC og linien PQ . Fortegnet af prikproduktet bestemmes af skalarprojektion af PQ på $(PC \times PB)$. Dermed afgøres det om PQ ligger på den ene eller anden side af et plan udspændt af PC og PB , og deraf om PQ ligger på den ene eller anden side af siden BC . For trekant ABC på figur 4.12 der ses m. spidserne imod uret, vil trippelproduktet u give en positiv værdi, da PQ er til venstre for BC , omvendt hvis spidserne ses med uret. Testen kan generaliseres til at der ses på trippelprodukternes fortegn: hvis de alle har samme fortegn skærer PQ trekanten.

Segment-plan

En plan er givet ved $P: (\vec{n} \cdot X) = d$ og et liniesegment ved parameterfremstillingen $S(t) = A + t(B - A)$ hvor $t \in [0, 1]$.

¹Er lig determinanten af den 3×3 matrice der har de tre vektorer som rækker eller kolonner



Figur 4.12: Linie-trekant skæring.

Ved at indsætte $S(t)$ i planets ligning i stedet for X og løse mht. t kan det afgøres om linesegmentet vil skære planet. Substitutionen er ligetil (se [6, s. 175]) og giver følgende udtryk

$$t = \frac{(d - \vec{n} \cdot A)}{\vec{n} \cdot (B - A)} \quad (4.1)$$

Segmentet vil skære planet hvis $S(t) \in P$ for et $t \in [0, 1]$, dvs. at testen giver et positivt resultat hvis $0 \leq t \leq 1$.

Hvis linesegmentet er parallelt med planet og $\vec{n} \cdot (B - A)$ dermed giver nul, vil der forekomme division med nul i ligning 4.1. Ved brug af IEEE 754 [18] floating point aritmetik² vil algoritmen fra [6, s.176] dog stadig give et korrekt resultat. Implementationen kontrollerer eksplicit for at t ligger inden for $[0, 1]$ (se [6, kap. 11.2.2] for nærmere diskussion).

Hvis en anvendt platform ikke er IEEE 754 standardiseret skal der sørges for en korrekt opførsel ved at teste eksplicit for en mulig division med nul.

Segment-trekant

Testen for om et linesegment L afgrænset af to punkter P og Q skærer en trekant kan sammensættes af to separate primitiv tests.

- *Linie-trekant*: Skærer linien PQ trekant ABC 's indre?
- *Segment-plan*: Skærer linesegmentet L trekant ABC 's plan?

Disse er begge nødvendige men ikke tilstrækkelige betingelser. Dermed kan segment-trekant testen laves som logisk konjunktion (*AND*) af de to tests.

²I IEEE 754 giver division af et positivt/negativt tal med nul henholdsvis plus/minus uendeligt

4.5.2 Segment piercing - Det koplanare tilfælde

Ovenstående beskrivelse af segment piercing algoritmen tager udgangspunkt i at de to trekanter ligger i hvert deres plan. Algoritmen kan ikke håndtere det koplanare tilfælde hvor de to trekanter ligger i samme plan, og dette tilfælde må derfor håndteres separat. Det koplanare tilfælde opstår når mindst to af trippel skalar produkterne i algoritmen giver nul³.

Når algoritmen har afgjort at de to trekanter ligger i samme plan projicerer trekanterne til et 2D plan, således at hver hjørne koordinat nu beskrives i \mathbb{R}^2 . Normalt fortages en sådan projektion ved at projicerer hvert hjørne vinkelret ind på enten XY , XZ , eller YZ planet. Dette er dog en dyr operation, da den kræver brug af prikprodukter. I stedet benyttes en såkaldt orthografisk projektion. Denne projektion fjerner ganske simpelt en koordinat fra hver koordinat sæt (den samme koordinat fra alle koordinat sæt, naturligvis!). Dette forvrænger naturligvis trekanterne, men dette har ingen praktisk betydning, idet dette ikke ændre ved om de overlapper eller ej. For at bestemme hvilken koordinat der skal fjernes, benyttes følgende metode:

- Først dannes en normal n til det plan trekanterne ligger i.
- Dernæst undersøges det hvilken komponent af n der er størst.
- Hvis n_x er størst fjernes x koordinaterne (projektion på YZ planet).
- Hvis n_y er størst fjernes y koordinaterne (projektion på XZ planet).
- Hvis n_z er størst fjernes z koordinaterne (projektion på XY planet).

Ovenstående metode kikker, med andre ord, på hvilke af de planer som koordinataakserne ligger i, der er mest parallel med trekanternes plan. Og projicerer dernæst trekanterne til dette plan.

For at undersøge de nye 2D trekanter T_1 og T_2 for overlap, benyttes der en udtømmende metode, der først undersøger om nogle af trekant T_1 's hjørner ligger inden i trekant T_2 . Herefter undersøges det om trekant T_2 har hjørner inden i trekant T_1 . Hvis det konstateres at bare en trekant har et hjørne fra den anden trekant i sit indre, er der tale om kollision imellem de to trekanter.

Til at bestemme om et punkt p ligger inden i en trekant ABC , benyttes krydsproduktet. Det skal dog bemærkes at da testen forgår i \mathbb{R}^2 , så kan det sædvanlige krydsprodukt ikke anvendes, da dette kun er defineret i \mathbb{R}^3 . Derfor benyttes et pseudo krydsprodukt defineret som

$$\begin{pmatrix} v_x \\ v_y \end{pmatrix} \times \begin{pmatrix} u_x \\ u_y \end{pmatrix} = \begin{pmatrix} -v_y \\ v_x \end{pmatrix} \cdot \begin{pmatrix} u_x \\ u_y \end{pmatrix} = (-v_y \cdot u_x) + (v_x \cdot u_y) \quad (4.2)$$

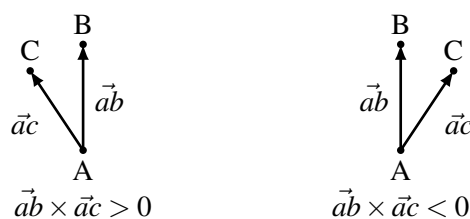
Figur 4.13 illustrere hvorledes 2D krydsproduktet kan benyttes til at bestemme på hvilken side af en vektor et punkt ligger.

Med 2D krydsproduktet kan følgende betingelser, for at d ligger i trekant ABC , stilles op

$$\text{sgn}(\vec{a}\vec{p} \times \vec{a}\vec{b}) = \text{sgn}(\vec{b}\vec{p} \times \vec{b}\vec{c}) = \text{sgn}(\vec{c}\vec{p} \times \vec{c}\vec{a}) \quad (4.3)$$

Hvis et af disse krydsprodukter afviger i fortegn fra de to andre, kan p ikke ligge i trekant ABC .

³Det tredje trippel skalar produkt giver strengt taget også nul i det koplanare tilfælde, men det er kun nødvendigt at test to af dem, for at bestemme om trekanterne ligger i samme plan.



Figur 4.13: Egenskaber ved 2D krydsprodukt.

4.5.3 Operationer

Segment-plan testen består af 1 krydsprodukt, 3 vektorsubtraktioner, 3 prikprodukter, 1 skalar subtraktion og 1 division. Linie-trekant testen består af 4 vektorsubtraktioner, 1 krydsprodukt, 2 prikprodukter og 1 triple skalar produkt.

Operation	add	sub	mul	div
Segment-plan	6	13	15	1
Linie-trekant	6	18	21	

Tabel 4.2: Basale operationer i segment-plan og linie-trekant tests

En trekant-trekant test er sammensat af 6 segment-trekant test, der hver består af en segment-plan og en linie-trekant.

Operation	add	sub	mul	div
Trekant-trekant	72	186	216	6

Tabel 4.3: Basale operationer i en segment-piercing trekant-trekant test

Dette giver i alt 480 operationer, worst case, hvilket forekommer når de to testede trekanter ikke er i kollision.

4.6 Devillers

Denne algoritme er baseret på arbejdet udført af Olivier Devillers og Philippe Guigue og som er beskrevet i artiklen *Faster Triangle-Triangle Intersection Tests* [5].

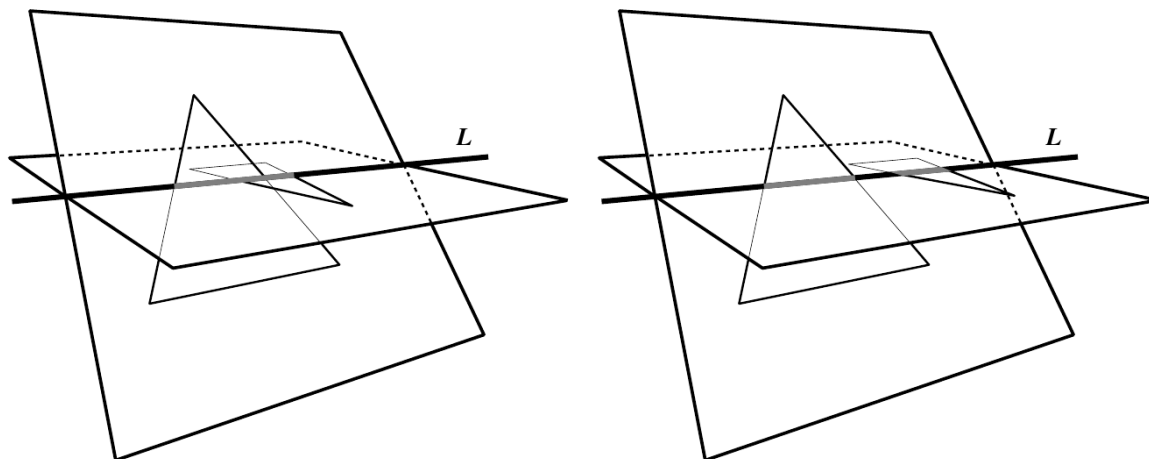
Algoritmen har den fordel at den er hurtig, og samtidig er den numerisk robust. For en nærmere diskussion af algoritmens fordele og ulemper henvises til artiklen.

4.6.1 Princip

Algoritmen bygger, som det nævnes i artiklen, videre på de principper som benyttes i tilsvarende algoritmer, se f.eks. [23]. En grundlæggende egenskab ved disse algoritmer, er at de forsøger at udelukke kollision, så hurtigt som muligt. Denne egenskab bygger på en formodning om at de fleste af de testede trekanter ikke er i kollision, og det derfor kan betale sig at udelukke disse tilfælde hurtigt.

Algoritmen udnytter det faktum, at hvis der er kollision mellem de to trekanter, så må den ene trekant skære det plan som den anden trekant ligger i, og omvendt. Dette er en nødvendig, men

ikke tilstrækkelig betingelse. Så hvis det kan påvises, at den ene trekants hjørner alle ligger på samme side af den anden trekants plan, så kan der ikke være tale om kollision trekanterne imellem. Omvendt ses det ved at betragte figur 4.14, at der kan opstå to situationer, når trekanterne rent faktisk skærer hinandens planer. Kun i den ene situation er der tale om en reel kollision trekanterne imellem.

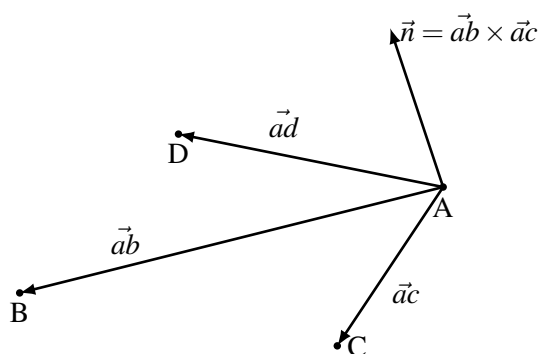


Figur 4.14: De to tilfælde der kan opstå, når de to trekanter skærer hinandens planer.

Hvis de to trekanter er i kollision, må denne nødvendigvis opstå på linien, hvor de to planer skærer hinanden (se figur 4.14). Kollisionstesten kan derfor afgøres ved at undersøge om de to intervaller, som hver trekant afgrænser på linien, overlapper. De omtalte intervaller er markeret med gråt på figur 4.14.

4.6.2 Algoritme beskrivelse

Som beskrevet ovenfor klassificerer algoritmen hvert hjørne i trekanterne, efter hvordan de er placeret i forhold til det plan den anden trekant ligger i. Denne klassificering forgår ved hjælp af et geometrisk prædikat der knytter en værdi til hvert hjørne. Denne værdi fortæller om et punkt (hjørne), ligger i det positive halvplan (positiv værdi), i det negative halvplan (negativ værdi), eller i selve planet (værdien er præcis nul).



Figur 4.15: Illustration af hvorledes det geometriske prædikat fungerer.

Figur 4.15 illustrerer hvoledes det geometriske prædikat fungere. Givet en trekant ABC i \mathbb{R}^3 , ønskes

det bestemt om et punkt D , ligeledes i \mathbb{R}^3 , ligger over eller under det plan som ABC ligger i. Først dannes to vektorer \vec{ab} og \vec{ac} , ud fra disse dannes en normalvektor til planet, givet som $\vec{n} = \vec{ab} \times \vec{ac}$. Normalvektorens retning bestemmes af den sædvanlige højrehånds regel. Dermed kan orienteringen af planet bestemmes, hvis det vedtages at punkterne i trekanten enummereres mod uret i rækkefølgen: A , B og C . Nu kan punktet D 's position i forhold til planet bestemmes, ved at benytte prikproduktet på vektor \vec{ad} og \vec{n} : $pd = \vec{ad} \cdot \vec{n}$. Da prikproduktet, som bekendt, afhænger af \cos til vinklen mellem de to vektorer, fås at pd er positiv når D ligger i det positive halvplan, nul når D ligger i selve planet, og negativ når D ligger i det negative halvplan. Samlet set kan det geometriske prædikat skrives et såkaldt trippel skalar produkt, som det ses i ligning 4.4.

$$pd = \vec{ad} \cdot (\vec{ab} \times \vec{ac}) \quad (4.4)$$

I artiklen benyttes en lidt anden metode, baseret på determinanter. Denne metode er dog essentielt den samme som den der er blevet præsenteret her (Se appendix A for en nærmere forklaring).

Hvert enkelt hjørne på de to trekanter, T_1 og T_2 , kan nu klassificeres i forhold til hinandens planer, π_1 og π_2 . Det skal i øvrigt bemærkes i ligning 4.4, at det kun er nødvendigt at udføre krydsproduktet en gang for hver trekant, idet det kun er \vec{ad} der ændrer sig for hvert tje. Når hjørnerne er klassificeret, sammenlignes fortegnet af prædikaterne. Tre situationer kan opstå for en enkelt trekant:

1. Det tre prædikater har samme fortegn, og ingen af dem er nul.
2. De tre prædikater er alle nul.
3. To af de tre prædikaters fortegn er forskelligt fra det tredje prædikats fortegn.

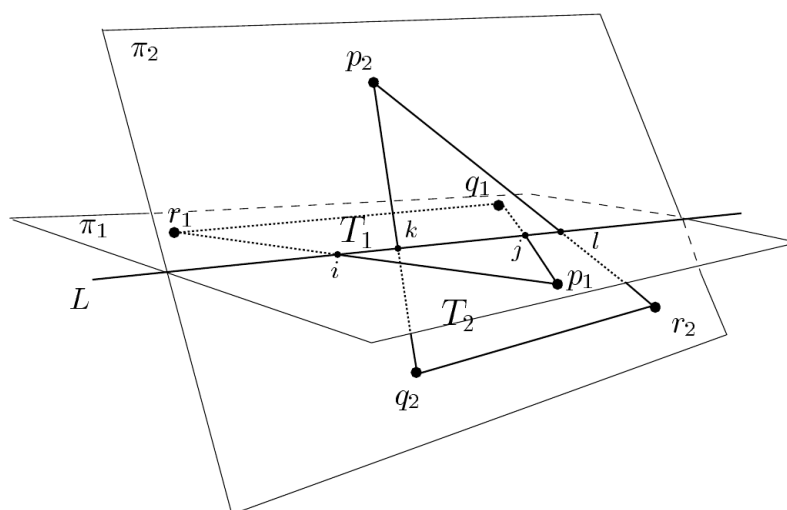
I det første tilfælde ligger de tre hjørner i samme halvplan, og der kan derfor ikke være tale om kollision de to trekanter imellem, og algoritmen kan derfor terminere.

I det andet tilfælde, ligger de to trekanter i samme plan. Dette tilfælde kaldes det koplanare tilfælde. Dette tilfælde er sjældent forekommende, og der er ikke i dette projekt blevet implementeret kode til at dække dette tilfælde. I artiklen kan der findes information om hvordan det koplanare tilfældes kan håndteres.

I det tredje tilfælde ligger to af trekantens hjørner på den ene side af planet, mens det tredje hjørne ligger på den modsatte side. Det skal bemærkes at det ene sæt punkter også kan ligge i selve planet, dette har dog ingen betydning, og dette tilfælde vil ikke blive behandlet separat. I dette tilfælde skærer trekanten planet og der kan derfor være tale om kollision.

Kun hvis begge trekanter skærer den anden trekants plan undersøges det nærmere om der reelt er tale om en kollision de to trekanter imellem, ellers konkluderer algoritmen at der ikke er nogen kollision, og terminerer derefter.

For at afgøre om de to trekanter kolliderer, undersøges det om de linje segmenter som hver trekant beskriver på den linje der opstår hvor de to planer skærer hinanden, overlapper eller ej. Se figur 4.16. Hvis segmenterne overlapper er der tale om kollision, ellers er der ikke. Hvis hjørnerne på de to trekanter nummereres som vist på figuren ($T_1 : \{p_1, q_1, r_1\}$ og $T_2 : \{p_2, q_2, r_2\}$), så benævnes skæringspunkterne mellem linien L og kanterne p_1r_1 , p_1q_1 , p_2q_2 og p_2r_2 , henholdsvis i , j , k og l . For at kunne kontrollere om intervallerne $I_1 = [i, j]$ og $I_2 = [k, l]$ overlapper, er det nødvendigt at ordne trekanternes hjørner således at $i < j$ og $k < l$. For at opnå dette udføres der en cirkulær permutation på trekanternes hjørner, således at p_1 og p_2 kommer til at ligge alene i hver deres halvplan. Yderligere udføres der evt. en ombytning af q_2 og r_2 , sådan at p_1 kommer til at ligge i det positive halvplan.

Figur 4.16: Placering af intervaller på linien L .

Dette samme gøres for q_1 og r_1 , så også p_2 kommer til at ligge i det positive halvplan. Efter disse operationer ligger hjørnerne på trekantene som vist på figur 4.16. Det skal nævnes at linien L 's orientering er bestemt af vektoren $\vec{N} = \vec{N}_1 \times \vec{N}_2$, hvor $\vec{N}_1 = p_1\vec{r}_1 \times p_1\vec{q}_1$ og $\vec{N}_2 = p_2\vec{r}_2 \times p_2\vec{q}_2$. Denne orientering skal sikre at der er muligt at sammenligne punkterne i , j , k og l .

Da skæringspunkterne nu ligger i den ønskede rækkefølge, er det enkelt at undersøge om intervallerne overlapper. Figur 4.17 viser et forsimplet billede af de situationer der kan opstå. Som det ses kan betingelsen for at de to intervaller overlapper, og dermed for kollision, udtrykkes som $k \leq j \wedge i \leq l$. Hvis den ene af disse to uligheder ikke er opfyldt kan der ikke være tale om overlap. Disse to situationer er også medtaget på figur 4.17, for overblikkets skyld.

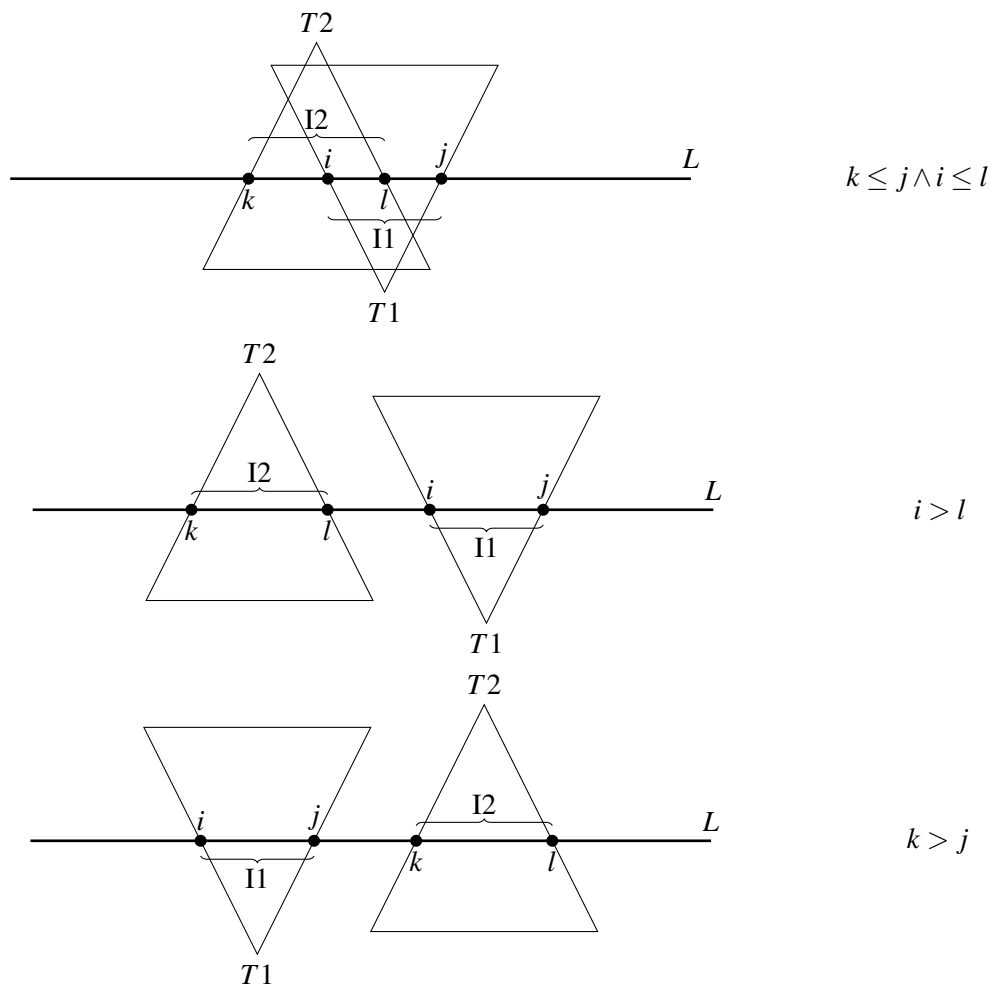
For at kunne udføre kollisionstesten ved hjælp af de beskrevne interval test, vil det være nødvendigt at kende linien L . Denne kunne findes ved at finde skæringen mellem de to planer. Der findes dog en nemmere måde at kontrollere om de to intervaller overlapper. I afsnit 4.5.1 er det beskrevet hvorledes det er muligt, at afgøre om en vektor ligger på den ene eller anden side af en bestemt kant i en trekant. I dette tilfælde kan denne metode benyttes til at afgøre om intervallerne overlapper, uden at kende linien L . Testen er todelt, da der er to intervaller der skal testes, nemlig $k \leq j$ og $i \leq l$. Den første test afgør på hvilken side vektor $p_2\vec{q}_2$ ligger i forhold til kanten p_1q_1 . Da skæringspunktet mellem kanten p_2q_2 og L er k , og skæringspunktet mellem kanten p_1q_1 og L er j , kan betingelsen $k \leq j$ udtrykkes ved hjælp af et trippel skalar produkt som

$$p_2\vec{q}_2 \cdot (p_2\vec{p}_1 \times p_2\vec{q}_1) \leq 0 \quad (4.5)$$

På samme vis kan uligheden $i \leq l$ udtrykkes som

$$r_2\vec{p}_2 \cdot (r_2\vec{r}_1 \times r_2\vec{p}_1) \leq 0 \quad (4.6)$$

Kun hvis begge ulighederne 4.5 og 4.6 er opfyldt, er der tale om kollision de to trekanter imellem.

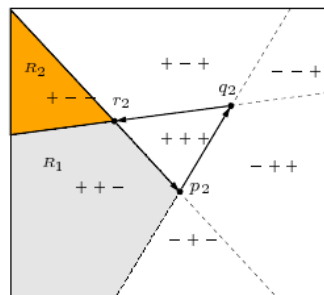


Figur 4.17: Overblik over interval test.

4.6.3 Devillers - Det koplanare tilfælde

I lighed med segment-piercing algoritmen, håndterer devillers algoritmen det koplanare tilfælde separat. Hvis devillers algoritmen tester to trekanter i samme plan, vil det geometriske prædikat for alle hjørner være nul. Hernæst benyttes en orthografisk projektion (se sektion 4.5.2) til at bringe trekanterne til \mathbb{R}^2 .

I det følgende betragtes de to trekanter $T_1 : \{p_1, q_1, r_1\}$ og $T_2 : \{p_2, q_2, r_2\}$ og det antages at deres hjørner er orienteret imod uret. Algoritmen benytter 2D krydsproduktet (se sektion 4.5.2) til at klassificere hjørnerne fra den ene trekant i forhold til den anden trekant. Først klassificeres p_1 i forhold til T_2 's kanter (vektorene $p_2\vec{q}_2$, $q_2\vec{r}_2$ og $r_2\vec{p}_2$). Denne klassificering deler planet, hvori T_2 ligger, op i en række regioner, som vist på figur 4.18.



Figur 4.18: Klassificering af p_1 i forhold til T_2 . Gengivet fra [5]

Fortegnene indikerer hvordan punktet p_1 klassificeres hvis det ligger i den pågældende region. Det skal bemærkes at hvis en klassificering er nul, betyder det at punktet ligger på en kant. Når denne klassificering er fortaget kan det i følgende tilfælde med det samme konkluderes at der er tale om kollision:

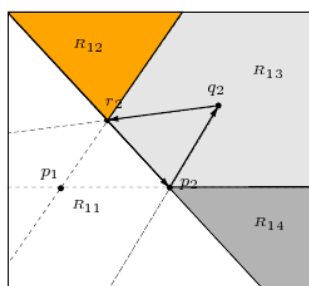
- Alle tre klassificeringer er positive $\mapsto p_1$ ligger inde i T_2 .
- En af klassificeringerne er nul, og de to andre er positive $\mapsto p_1$ ligger på det indre af en kant af T_2 .
- To af klassificeringerne er nul $\mapsto p_1$ ligger på et hjørne af T_2 .

Tilfældet hvor alle tre klassificeringer er nul, kan ikke opstå hvis det antages at trekanterne ikke kan være degenereret. Ligeledes opfattes tilfældet hvor p_1 ligger på det ydre af en kant (en klassificering er nul, og mindst en af de andre er negative) ikke som en kollision.

Hvis der på dette tidspunkt er konstateret en kollision, stopper algoritmen. Ellers udføre algoritmen en permutation på T_2 , således at p_1 kommer til at ligge internt i regionen R_1 , eller internt eller på kanten af region R_2 , se figur 4.18.

Hvis p_1 efter permutationen kommer til at ligge i region R_1 opdeles planet i en række regioner som vist på figur 4.19.

Hernæst klassificeres q_1 med hensyn til de nye regioner. Først klassificeres q_1 i forhold til kanten r_2p_2 . Hvis denne klassificering er negativ, ligger q_1 i regionen R_{11} , og kanten p_1q_1 kan således ikke skære T_2 , og det er derfor nødvendigt at undersøge r_1 's placering i planet. Hvis den derimod er positiv er der en mulighed for at kanten p_1q_1 skære T_2 . For endeligt at afgøre om der er kollision eller ej, er



Figur 4.19: Opdeling i regioner hvis p_1 ligger i regionen R_1 . Gengivet fra [5]

der en hel række af tilsvarende tests der skal udføres. Figur 4.20 viser et beslutningstræ der inkluderer alle disse tests.

Hvis p_1 efter permutationen kommer til at ligge i region R_2 opdeles planet i en række regioner som vist på figur 4.21.

Igen undersøges det hvor de andre punkter fra T_1 ligger i planet, for derved at afgøre om nogle af kanterne fra T_1 skærer T_2 . Disse tests er illustreret i det beslutningstræ der er gengivet i figur 4.22.

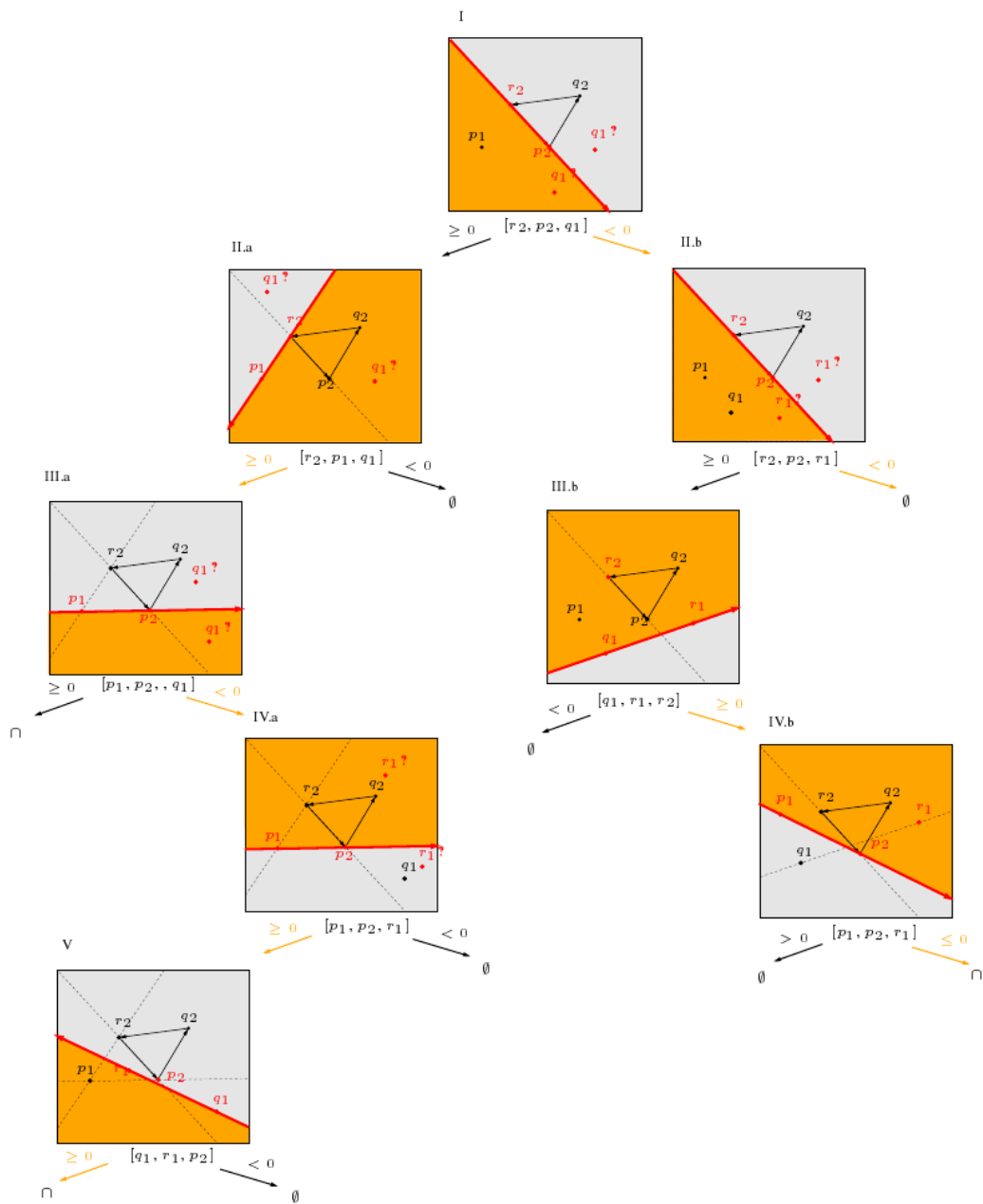
4.6.4 Operationer

Devillers algoritmen består af 4 krydsprodukter + 16 vektorsubstraktioner + 8 prikprodukter.

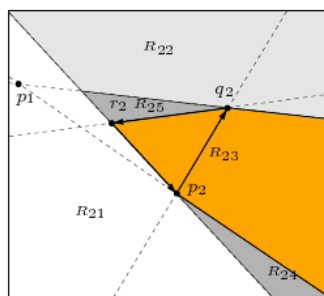
Operation	add	sub	mul	div
Devillers	16	60	48	0

Tabel 4.4: Basale operationer i Devillers trekant-trekant testen

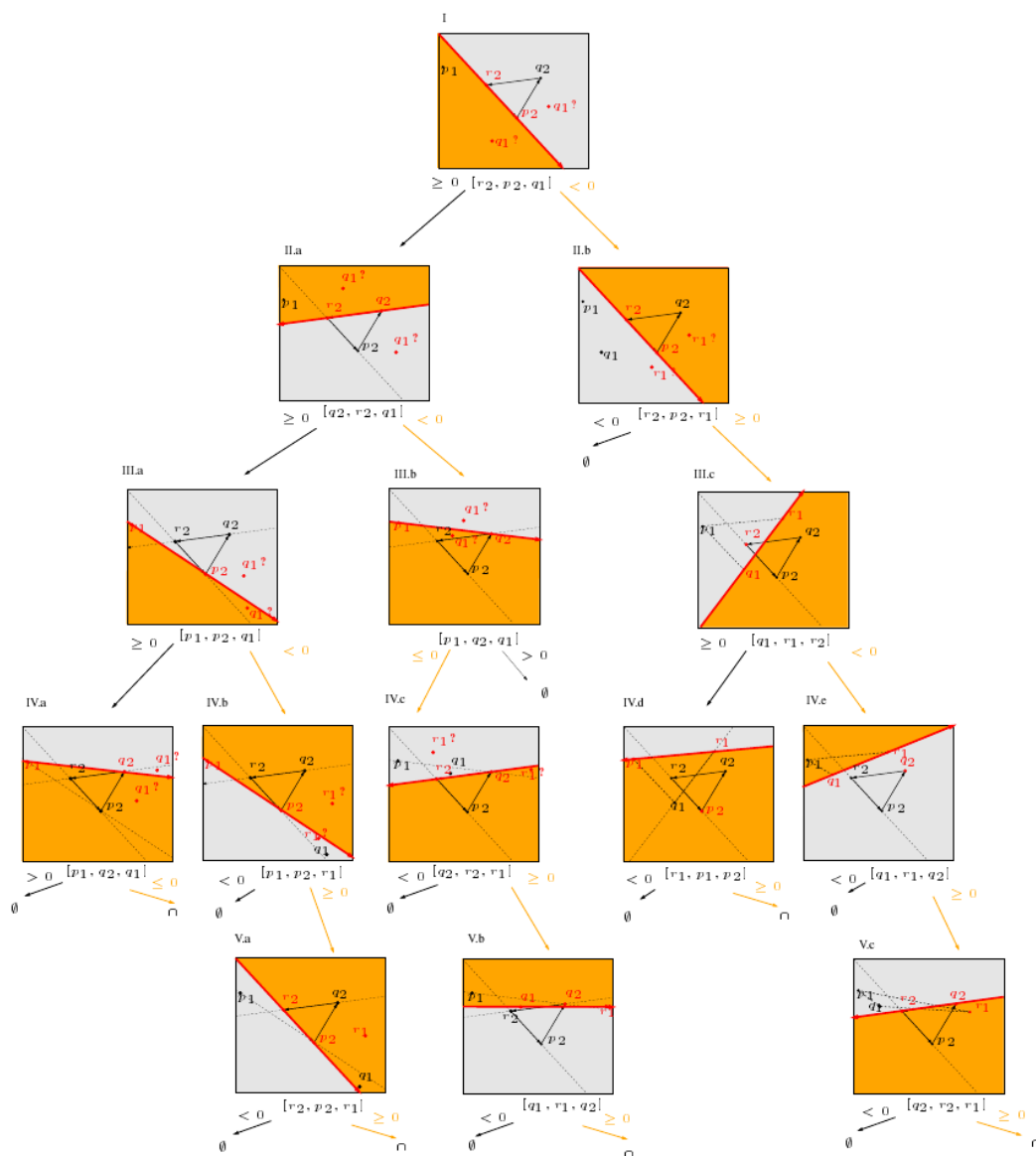
Dette giver i alt 124 basale operationer.



Figur 4.20: Beslutningstræ, der viser de tests der skal udføres hvis p_1 ligger i region R_1 . Gengivet fra [5]



Figur 4.21: Opdeling i regioner hvis p_1 ligger i regionen R_2 . Gengivet fra [5]



Figur 4.22: Beslutningstræ, der viser de tests der skal udføres hvis p_1 ligger i region R_2 . Gengivet fra [5]

4.7 Orienting Bounding Box tests

Som beskrevet i afsnit 4.3.3 er det interessant at anvende OBB'er i kollisionsdetektion da de pga. deres arbitrære orientering kan have et godt fit til den underliggende geometri. Den arbitrære orientering gør det imidlertid mere kompliceret at udføre OBB-OBB end f.eks. AABB-AABB tests.

Algoritmerne beskrevet i de følgende afsnit har til formål at teste om to OBB'er, A og B med arbitrære rotationer og translationer kolliderer eller ej.

A og B er hvis ikke andet er nævnt, givet i samme koordinatframe.

4.7.1 Exhaustive Edge-Face

Exhaustive algoritmen (navnet inspireret af [9]) er en ligetil løsning, der laver en udtømmende test af alle kanter fra den ene boks imod den andens flader og vice-versa. Desuden skal der laves to punkt-i-boks tests for at dække tilfældene af at den ene boks skulle være indeholdt i den anden.

Algoritmen er meget lig segment-piercing trekant-trekant algoritmen beskrevet i afsnit 4.5. Liniesegment-trekant testen der anvendes i segment-piercing består som bekendt af linie-trekant og segment-plan. Segment-plan testen kan anvendes direkte, og linie-trekant testen er let at udvide til at omfatte en ekstra sidedness-test.

Algoritmen har følgende overordnede trin

1. Test alle kanter fra boks A imod alle flader fra boks B
2. Test alle kanter fra boks B imod alle flader fra boks A
3. Test et punkt fra hver boks indeholdt i den anden boks

Hvis der forekommer en kollision imellem en kant og en boks i trin 1 eller 2, terminerer algoritmen og der rapporteres kollision. Hvis der ikke findes kollisioner i trin 1 eller 2 er det nødvendigt at udføre trin 3, hvor det testes om den ene boks er komplet indeholdt i den anden eller vice-versa.

Operation	add	sub	mul	div
Segment-flade	14	40	51	1

Tabel 4.5: Basale operationer i Segment-OBB face test

Dette kræver op til 144 segment-flade tests og 2 punkt-i-boks tests, hvis OBB A og B ikke er i kollision. Hver segment-flade test kræver alene i floating-point beregninger omkring 100 instruktioner⁴ som det fremgår af tabel 4.5, og gør derfor algoritmen meget beregningstung.

Operation	add	sub	mul	div
Exhaustive	1728	4464	5184	144

Tabel 4.6: Basale operationer i Exhaustive OBB-OBB test

At algoritmen tager længst tid når der ikke forekommer kollision er uhensigtsmæssigt i forbindelse med en hierarkisk metode. Her vil en stor del af træets OBB'er ofte ikke være i kollision, og disse skal udelukkes hurtigst muligt.

⁴Antallet af instruktioner kan dog reduceres ved at transformere den ene OBB til at være axis-aligned.

I lighed med segment-piercing trekant-trekant testen, er det nødvendigt at tage hensyn til ko-planaritet. Dette kan gøres på samme måde som for trekant-trekant testen ved at detektere når de to trippel skalar produkter i algoritmen giver nul som beskrevet i afsnit 4.5.2.

4.7.2 Gottschalk

Gottschalk-algoritmen (navngivet efter Stefan Gottschalk der beskriver den i [8] og [9]) er baseret på test af separerende akser (SAT). Algoritmen kan afgøre om to OBB'er i generel position kolliderer eller ej.

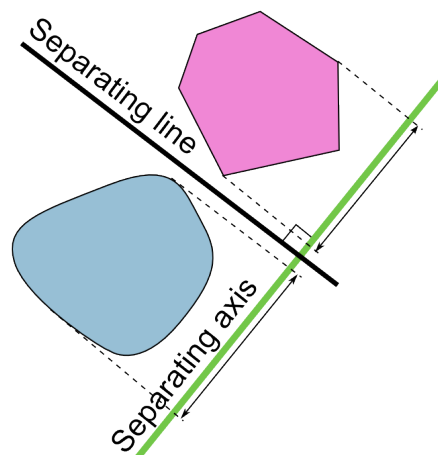
Korollaren for eksistensen af en separerende akse imellem to objekter stammer fra theoremet omkring separerende hyperplaner for konvekse mængder. Theoremets er et resultat af konveks analyse, og vil derfor ikke blive nærmere behandlet

Theorem 4.7.1 *To ikke-tomme konvekse delmængder af \mathbb{R}^3 kan separeres af et plan, hvis og kun hvis de er disjunkte.*

Dualen til dette theorem er en separerende akse, vinkelret på det separerende plan

Kollorar 4.7.2 *En akse n er en separerende akse for to, ikke-tomme konvekse delmængder af \mathbb{R}^3 , hvis og kun hvis deres orthogonale projektioner på n ikke overlapper.*

Sammenhængende er vist på figur 4.23 i 2D. Fordi der kun eksisterer en separerende akse, hvis og kun hvis der eksisterer et separerende plan, kan kollisionstests baseres på at bevise en af delene. Det er dog oftest lettere at udføre de orthogonale projektioner på en kandidat-akse og se på overlappet end at skulle søge efter et separerende plan.



Figur 4.23: Separerende plan og ækvivalent den separerende akse i 2D. (Kilde: http://en.wikipedia.org/wiki/Separating_axis_theorem)

For at kunne anvende ovenstående i en effektiv kollisiondetektion af OBB'er er der to spørgsmål der skal afklares

- Givet en kandidat til en separerende akse, hvordan udfører vi den orthogonale projektion og tester for overlap af de to OBB'ers intervaller?

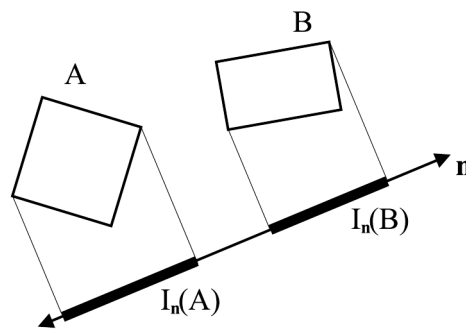
- Hvilke akser skal minimum afprøves for at vi har været alle kandidater igennem?

I det følgende vil Gottschalk-algorithmens løsning på disse to spørgsmål blive præsenteret.

Test af kandidatakse for OBB'er

To arbitrære roterede og translaterede OBB'er A og B , er givet i den samme koordinatframe. Problemet er nu at bestemme om de to OBB'er kolliderer eller ej.

OBB'ernes dimensioner i deres x -, y - og z -retninger er givet ved henholdsvis a_1, a_2, a_3 for A og b_1, b_2, b_3 for B . Centrum af A 's koordinatframe er givet ved \mathbf{T}^A , \mathbf{T}^B for B og endelig er rotationerne givet ved \mathbf{R}^A og \mathbf{R}^B .



Figur 4.24: Orthogonal projektion af OBB'er på kandidatakse. (Gengivet fra [9])

Generelt for at teste for overlap af to konvekse polyhedroner A og B i 3D med SAT, er det nødvendigt at udføre en orthogonal projektion af A og B på den potentielt separerende akse og finde deres maksimale interval. Er A og B OBB'er, laves dette med et prikprodukt imellem hvert af de 8 hjørner og akse, som vist på figur 4.24. Dette giver længden af projektionen, og ved at kombinere dette med $7 \cdot 2 = 14$ sammenligninger for min/max, fås intervallet af alle hjørnernes projektion på akse. Beregning af A og B 's intervaller giver tilsammen 16 prikprodukter og 28 sammenligninger. Testen for overlap af de to intervaller er yderligere 2 sammenligninger. Tælles det forventede antal aritmetiske operationer fås der 48 multiplikationer og 32 additioner fra prikprodukterne⁵.

OBB'er har imidlertid en række egenskaber der gør en del optimeringer mulige. Fordi at de er symmetriske, vil en OBB's centrum altid projicere ned på midten af projektionsintervallet. Derfor kan testen, som vist på figur 4.26 (a) omformuleres til at teste for om separationen s imellem projektionen af OBB'ernes centrum \mathbf{T}^A og \mathbf{T}^B på akse er større end summen af deres intervaller 'halv-bredde' (figur 4.26 (b)).

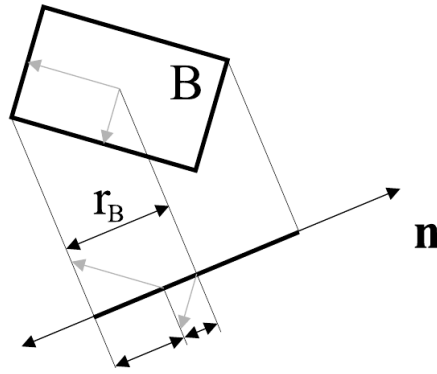
Halv-bredderne kan opfattes som de to OBB'ers radii, og benævnes r_a og r_b . De kan beregnes som summen af længderne af projektionerne af OBB'ernes dimensioner på \mathbf{n} , som vist på figur 4.25.

To OBB'er er dermed adskilte under orthogonal projektion på en akse, hvis og kun hvis

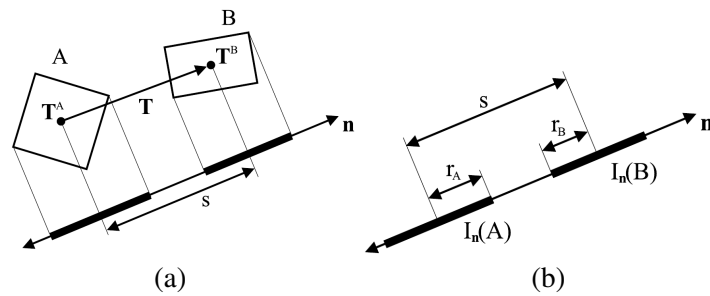
$$s > r_A + r_B \quad (4.7)$$

Separationen s er givet ved

⁵Det antages her at hjørnerne umiddelbart er tilgængelige uden ekstra beregning



Figur 4.25: Halvbredde af intervaller. (Gengivet fra [9])



Figur 4.26: Separation af intervaller. (Gengivet fra [9])

$$s = |(\mathbf{T}^A - \mathbf{T}^B) \cdot \mathbf{n}| / |\mathbf{n}| \quad (4.8)$$

Radius for boks A udregnes som

$$r_A = (a_1 |\mathbf{R}_1^A \cdot \mathbf{n}| + a_2 |\mathbf{R}_2^A \cdot \mathbf{n}| + a_3 |\mathbf{R}_3^A \cdot \mathbf{n}|) / |\mathbf{n}| \quad (4.9)$$

og for B

$$r_B = (b_1 |\mathbf{R}_1^B \cdot \mathbf{n}| + b_2 |\mathbf{R}_2^B \cdot \mathbf{n}| + b_3 |\mathbf{R}_3^B \cdot \mathbf{n}|) / |\mathbf{n}| \quad (4.10)$$

Ligning 4.7 bliver så, hvis der ganges igennem med $|\mathbf{n}|$

$$|(\mathbf{T}^A - \mathbf{T}^B) \cdot \mathbf{n}| > (a_1 |\mathbf{R}_1^A \cdot \mathbf{n}| + a_2 |\mathbf{R}_2^A \cdot \mathbf{n}| + a_3 |\mathbf{R}_3^A \cdot \mathbf{n}|) + (b_1 |\mathbf{R}_1^B \cdot \mathbf{n}| + b_2 |\mathbf{R}_2^B \cdot \mathbf{n}| + b_3 |\mathbf{R}_3^B \cdot \mathbf{n}|) \quad (4.11)$$

Tilsammen er der 7 prikprodukter, 7 abs, 1 vektor subtraktion, 6 skalar-vektor multiplikationer og 4 additioner. Dette giver 39 multiplikationer, 18 additioner, 3 subtraktion og 7 abs for en total på 67

operationer. I forhold til den tidligere foreslåede test er dette en besparelse på 13 operationer. Selvom det ikke lyder af meget, skal det bemærkes at beregning af hjørnernes placering ikke var medtaget. Yderligere optimering af Gottschalk-algoritmen beror på strukturen for kandidater til separerende akser. Det er derfor beskrevet senere, i afsnit 4.7.2.

Valg af kandidatakse for OBB'er

Generelt kan to konvekse polyhedra komme i kontakt på 6 måder. Dette er for at bruge de engelske betegnelser: face-face, face-edge, face-vertex, edge-edge, edge-vertex og vertex-vertex. Da edges som bekendt indeholder de tilstødende vertices reduceres ovenstående til 3 tilfælde: face-face, face-edge og edge-edge.

I [9] præsenterer Gottschalk et bevis for, hvordan disse tre tilfælde kan dækkes af at anvende følgende som separerende akser

- Vektorer normal til en face fra polyhedra A
- Vektorer normal til en face fra polyhedra B
- Vektorer vinkelret på en edge fra taget fra hver polyhedra (krydsproduktet af alle edges fra A imod alle edges fra B)

I den tilgængelige kilde er beviset desværre ret uklart. Gottschalk har ifølge litteraturhenvisningerne i [9] udgivet et supplerende paper netop til separerende akse theoremet men dette har været umuligt at finde, selv efter henvendelse til The University of North Carolina at Chapel Hill.

I stedet præsenteres her et argument, for specialtilfældet af OBB'er, at ovenstående kandidatvektorer er tilstrækkelige for at dække alle mulige kandidatakser.

To OBB'er, A og B er i generel position med nærmeste punkter entydigt bestemt. Der ønskes argumenteret hvorfor at face-face, face-edge og edge-edge tilfældene dækkes med separerende akser konstrueret af

1. Vektorer normal til en face fra hver OBB
2. Vektorer vinkelret på en edge fra hver OBB

Face-edge håndteres af tilfælde 1. Normalen til facen vil nemlig altid være parallel med vektoren imellem nærmeste punkter.

I tilfældet, hvor nærmeste punkter ligger rent⁶ edge-edge, vil en normal til en edge udgøre den bedste kandidat. Da normaler fra de to tilstødende faces opfylder at være normale til edgen, dækkes tilfældet af 1.

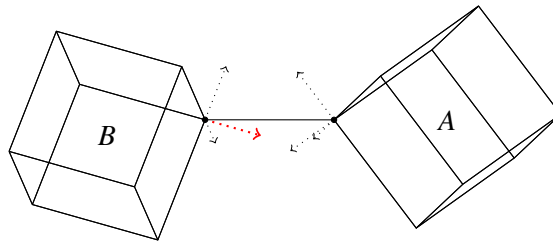
Specialtilfældene vertex-vertex og edge-vertex (fra edge-edge) kræver en nærmere forklaring.

Vertex-vertex

I figur 4.27 er vektoren imellem nærmeste punkter placeret vandret. For de nærmeste punkter er de udadrettede normalvektorer for de tilstødende faces vist. Den normalvektor der er tættest på vandret er en separerende akse (på figuren markeret med rødt), da den har størst komponent af vektoren imellem de nærmeste punkter.

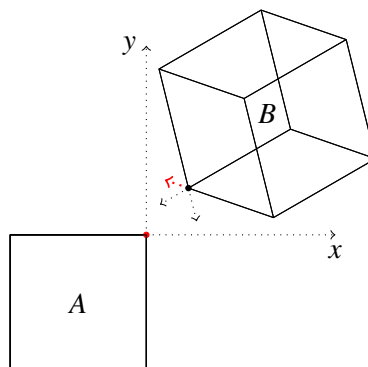
Edge-vertex

⁶I den forstand at det ikke er de indeholdte vertex der er nærmeste punkter



Figur 4.27: Vertex-vertex specialtilfældet. En udadrettet normalvektor til en face fra B (markeret med rødt) har størst komponent af vektoren imellem de nærmest punkter, og udgør i dette tilfælde en separerende akse.

I figur 4.28 er boks A placeret så den ligger i tredje kvadrant af et xy -koordinatsystem. Projektionen er udført så at to edges fra A er vinkelrette og den sidste edge går ud af papiret. Nærmeste punkt fra B må nødvendigvis ligge i første kvadrant, hvis ikke er vi i et andet tilfælde eller der er kollision. Vist på figuren er normalerne til de tilstødende faces for B . Hvis en normal i nærmeste punkt fra B skal kunne bruges til at denne et separerende plan parallelt med må den nødvendigvis gå imod anden eller fjerde kvadrant. Da de tre normaler er indbyrdes orthogonale (alle parvise skalarprodukter nul) vil, hvis der ikke er kollision, altid være mindst en normal der kan bruges. Dette er det samme som at en vektor vinkelret på vil danne en separerende akse. Dette er på figuren opfyldt af krydsproduktet imellem z -aksen der går ud af papiret og edge fra B begge markeret med rødt.



Figur 4.28: Edge-vertex specialtilfældet. Krydsproduktet af edges fra hver af boksene, vist med rødt udgør separerende akse.

For en naiv udvælgelse af kandidatakser vil der, selv med brug af ovenstående, være nødvendigt at teste et væsentligt antal koordinatakser. For to OBB'er der hver har 8 edges og 6 faces, ville der være $E^2 2F = 76$ kandidatakser.

OBB'er (og AABB som specialtilfælde) er imidlertid ganske symmetriske. Der er kun 3 unikke face-normaler og 3 unikke edge-retninger, hvis man ser bort fra fortegnene. Dette reducerer antallet af kandidatakser til 15: 9 kombinationer af krydsprodukter af edges fra henholdsvis A og B og 6 face normaler, 3 fra hver boks.

Samlet optimering

Testen af en kandidatakse til separation af to OBB'er er som i ligning 4.12, generel i den forstand at den kan bruges på enhver kandidatvektor \mathbf{n} . For at teste 15 kandidatakser skal der for en arbitrær vektor, jævnfør tidligere afsnit, anvendes 585 multiplikationer, 315 additioner/substraktioner og 105 abs.

Der er imidlertid muligt at reducere antal beregninger ved at udnytte strukturen af kandidatvektorerne præsenteret i forrige afsnit. Testen tilpasses så for tilfældende: $\mathbf{n} = \mathbf{R}_i^A$ (normaler fra A), $\mathbf{n} = \mathbf{R}_i^B$ (normaler fra B) og $\mathbf{n} = \mathbf{R}_i^A \times \mathbf{R}_i^B$ (edges fra hver boks, svarende til en face normal fra hver boks) hvor $i, j \in \{1, 2, 3\}$.

De to første tilpasninger, findes nemt ved at indsætte $\mathbf{n} = \mathbf{R}_i^A$ og $\mathbf{n} = \mathbf{R}_i^B$ for $i \in \{1, 2, 3\}$ i 4.12. Mange af vektorerne vil være indbyrdes orthogonale og mange led der bidrager til projektionen vil forsvinde (se [9] for nærmere detaljer).

Den tredje optimering udnytter ligeledes orthogonalitet af vektorerne. Derudover foretages der omarrangering af udtrykkene. Kombineret tager det i alt for test af de 15 kandidat akser 261 multiplikationer, 180 additioner/substraktioner og 69 abs.

De to sidste optimeringer som Gottschalk præsenterer beror på valget af koordinatsystem samt udregning af fælles udtryk i de 15 tests.

Ved at transformere boks A til at ligge i origin og med dens akser parallelt med origins akser og anvende samme transformation på B kan der netto opnås yderligere besparelser. Transformationen der skal anvendes for at bringe A til origin, parallelt med koordinatakserne svarer til den homogene transformation $(\mathbf{F}^A)^{-1} = (\mathbf{F}^A)^T$. Denne anvendes på B , så der fås $(\mathbf{F}^A)^T \mathbf{F}^B$.

Efter optimeringen vha. valget af koordinatsystem er de 15 tests reduceret til at kræve 81 multiplikationer, 60 additioner/substraktioner og 69 abs. Den homogene transformation af B kræver yderligere 36 multiplikationer og 27 additioner⁷.

I alt tager test af de 15 kandidatakser, med caching af absolutte værdier, 117 multiplikationer, 87 additioner/substraktioner, 24 abs og 15 sammenligninger. Dette er et upper-bound, da det ikke er nødvendigt at lede efter flere separerende akser når først en er fundet.

Operation	add/sub	mul	abs	cmp
Gottschalk	87	117	24	15

Tabel 4.7: Basale operationer i Gottschalk OBB-OBB test

Numerisk robusthed

I praktiske implementationer, hvor der anvendes endelig-præcisions floating-point aritmetik er det nødvendigt at have en algoritme der er robust over for numeriske unøjagtigheder.

Uden at komme ind på de nærmere detaljer er Gottschalk-algoritmen generelt solid. Der skal dog tages hensyn til, at rotationsmatricer i praksis kan gå hen og miste de egenskaber der gør dem til rotationsmatricer (orthonormal, $\det R = 1$, $R^{-1} = R^T$). Dette er diskuteret i [9] og løsningen der præsenteres er at tilføje en lille værdi, $1 \cdot 10^{-6}$ til de absolutte værdier på højresiden af 4.12 (dvs. også for det optimerede tilfælde). For tilfældet hvor to OBB'er har deres faces parallelle, gør dette testen en anelse mere konservativ.

⁷Det antages her at $(\mathbf{F}^A)^T$ på forhånd er kendt

4.8 Implementation

Der henvises til afsnit 9.1.5 og 9.1.4 for en beskrivelse af implementationen af alle de beskrevne algoritmer på hhv. GPU og Cell processoren.

Kapitel 5

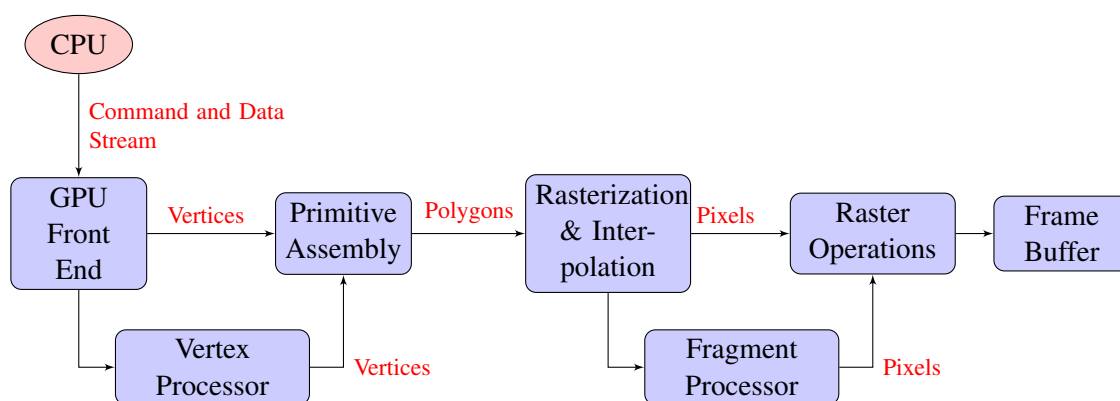
Graphics Processing Unit

5.1 Introduktion

En GPU eller grafikaccelerator er en specialiseret processor, beregnet til at aflaste CPU'er i grafikberegninger. I modsætning til en CPU der skal kunne køre en lang række forskellige applikationer er en GPU udviklet specielt til de matematiske operationer og algoritmer med floating-point tal der udføres inden for computergrafik. Moderne GPU'er har en meget høj ydelse og er derfor interessant til andre formål end grafikapplikationer.

5.2 Historie

De første udbredte grafikacceleratorer stammer fra starten/midten af 80'erne bl.a. som set i de populære Amiga computere [36]. Amiga'ernes chipset havde dedikeret hardware [32] til at kunne overføre bitmaps fra hovedhukommelsen samt til at lave linie-tegninger og område-fyldninger. Disse funktioner lå på en dedikeret kreds og blev tilgået vha. et specielt instruktions sæt. Hardwaren gjorde at Amiga'ens CPU blev væsentlig aflastet og gjorde den i stand til at lave væsentligt mere avanceret 2D GUI grafik end andre samtidige systemer.







Figur 5.1: Klassisk 3D GPU pipeline (Gengivet fra [24]).

I starten af 90'erne fandt PC-plattformen og Microsoft Windows stor udbredelse. Da Windows var et grafisk miljø var der et stort behov for 2D acceleration og grafikort producenterne begyndte derfor

at lave funktioner i deres kort specielt til Windows' grafikrutiner. I samme årrække blev interessen for 3D computerspil gradvist større og det skabte et behov for dedikeret hardware acceleration af de tunge opgaver såsom Z-buffering, anti-aliasing og teksturfiltrering.

I 1999 tog NVIDIA det næste store skridt inden for grafik kort ved lanceringen af deres GeForce serie [2], der understøttede Transform and Lighting (T&L) direkte i hardware. GPU'ens funktionalitet lignede nu den viste på figur 5.1. GPU'en var nu i stand til at lave geometri transformationer imellem referencesystemer samt beregning af lysforhold. Dette var førhen forbeholdt CPU'er, da det kræver en omfattende implementation af floating-point matrixoperationer i hardware.

Application tasks (move objects according to application, move/aim camera)	CPU	CPU	CPU	CPU	3D Application and API
Scene level calculations (object level culling, select detail level, create object mesh)	CPU	CPU	CPU	CPU	
 Transform	CPU	CPU	CPU	GPU	3D Graphics Pipeline
 Lighting	CPU	CPU	CPU	GPU	
 Triangle Setup and Clipping	CPU	Graphics Processor	Graphics Processor	GPU	
 Rendering	Graphics Processor	Graphics Processor	Graphics Processor	GPU	
	1996	1997	1998	1999	

Figur 5.2: Evolutionen af 3D grafikpipeline i GPU'er (Billede fra NVIDIA 'Transform and Lighting' [2])

I 2001 kom NVIDIA med GeForce 3 serien af grafik kort, der som de første havde programmerbare vertex- og pixelshaders [34]. Dette betød at man nu på visse punkter kunne omprogrammere GPU'ens pipeline til at udføre ekstra operationer på objekter og skærmpixels. Sådanne programmer (kaldet shaders) var dog stærkt begrænset i længde og tilladte operationer, bl.a. var det ikke muligt at lave løkker. Kort tid efter i 2002 introducerede ATI Radeon 9700 der nu kunne understøtte løkker og flere instruktioner i dens shaders [36].

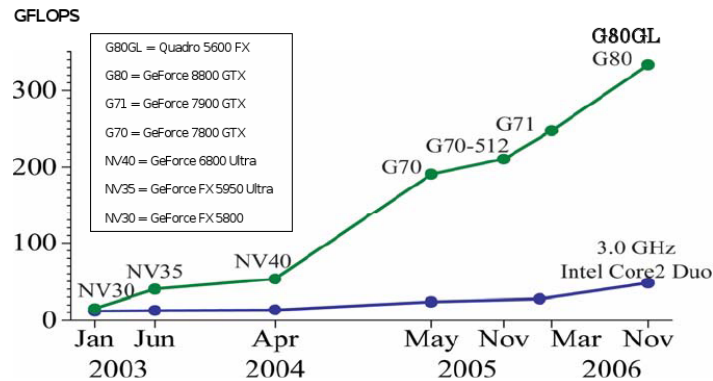
Programmerbare shaders bruges normalt til at give spil og grafikapplikationer specielle effekter. Vertex shaders bruges til bl.a. deformation af objekter, realistiske vandoverflader og animationer [4]. Pixel shaders bruges til lyseffekter, refleksioner, skygger mm [3].

5.3 Arkitektur

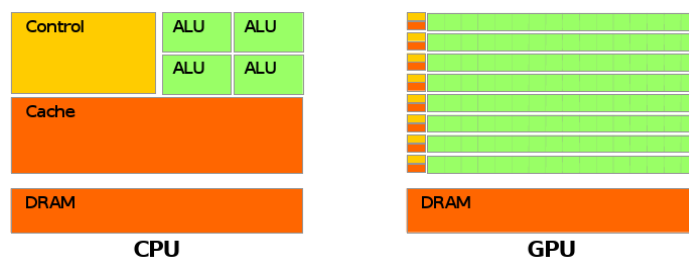
Interessen har været stor for GPU'er grundet deres store floating-point ydelse der har udviklet sig kraftigt over årene som vist på fig 5.3.

I modsætning til CPU'er der skal udføre mange forskelligartede opgaver, er GPU'er specielt optimeret til aritmetik og 'spilder' ikke megen plads (se figur 5.4) på flow control og caching. De har derfor høj aritmetik ydelse i forhold til det anvendte chipareal. GPU'er er derfor velegnede til beregningstunge opgaver på store datasæt [10].

GPU'er er stærkt parallelle arkitekturer der beregner en output stream af pixels ud fra en input stream af 3D grafikprimitiver og sceneinformationer. GPU'er gør dette ved at have en stort antal parallelle



Figur 5.3: Udvikling af GPU'ers ydeevne kontra CPU'er (Kilde: [25])



Figur 5.4: Udnyttelse af chipareal af CPU vs. GPU (Kilde: [25])

programmerbare processorer der udfører en kerne¹ på streamelementerne [10]. Denne stream processing egner sig i særdeleshed til grafik da der skal regnes på et stort antal primitiver og beregningerne i stor udstrækning er uafhængige.

5.4 General-Purpose computation

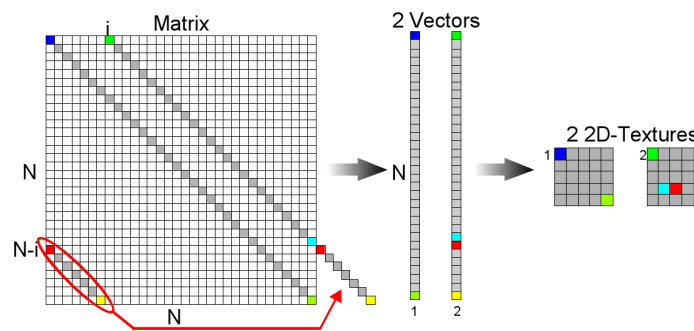
De fleksible shaders understøtter matrix og vektor operationer i floating point præcision. Sammen med GPU'ernes store regnekraft har dette tiltrukket mange ikke-grafiske formål. Dette kaldes for GPGPU² (engelsk for 'General-Purpose computation on GPUs').

Traditionelt er det nødvendigt at omsætte GPGPU-applikationer til grafik, dvs. at repræsentere programmets data som billedteksturer og omsætte algoritmer til et billedbehandlingproblem.

Dette er langt fra trivelt og kræver ud over et godt kendskab til GPU'ens grafikpipeline en vis mængde kreativitet. Men hvis problemet f.eks. er udtrykt over 2D grid er dette naturligt repræsenteret på en GPU som teksturer [10]. Grafikkortets maksimale skærmløsning sætter imidlertid en grænse for hvor højtopløst grid³ et kan være. Programmerings-API'er, pipelines og hukommelsestilgang er typisk meget restriktive. I API'et kan være et betragteligt overhead i f.eks. udlæsning af data og initialisering af grafikkortet. Hukommelsestilgangen er normalt heller ikke enkel da vertex- og pixelshaderne ikke understøtter random-access som man er vant til. Branching er ligeledes et problem da mange grafikkort ikke understøtter det. De der gør mister typisk meget performance.

¹En række af operationer der skal udføres for hvert element i input streamen

²<http://www.gpgpu.org>



Figur 5.5: Båndmatrice i en GPU. Et eksempel på en datastruktur repræsenteret som grafik (Kilde: <http://www.gpgpu.org>).

5.4.1 Unified Shader og Pipeline

I den klassiske GPU pipeline som vist på figur 5.1, er dataflowet i stor grad fastlagt. Vertex shading, rasterisering og pixel shaders følger f.eks. lineært efter hinanden, og mellemresultater i en blok kan ikke genbruges i en anden. Dette har medført en række begrænsninger for GPGPU som beskrevet i forrige afsnit.

Hvor tidligere GPU'er anvendte separate processorer til hver type af grafikoperation, anvender unified shader arkitekturen i stedet mange general-purpose, floating-point processorer. De forskellige vertex, raster og pixel shader operationer skeduleres så til at køre.

En unified shader pipeline giver en stor fordel mht. balancering af load i grafikapplikationer, da der ikke er en fast partitionering af GPU'ens regnekraft imellem f.eks. vertex og pixel shader operationer. Endnu mere interessant er det at GPU'en bliver langt bedre egnet til GPGPU applikationer.

5.5 Compute Unified Device Architecture

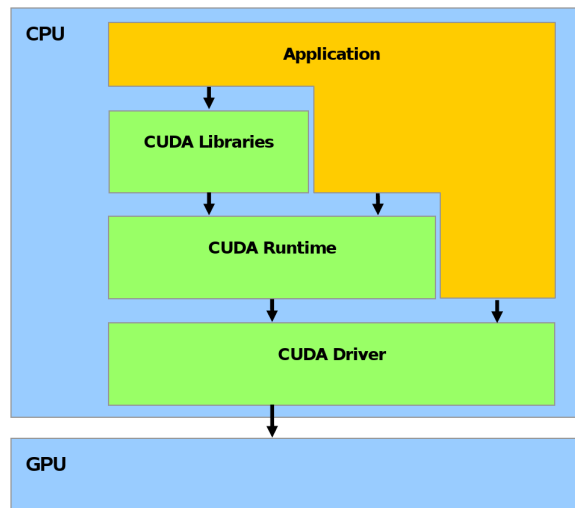
Inden for det sidste år har der været øget fokus på GPGPU formål og hardwareproducenterne er begyndt at udgive værktøjer til det. Et af disse værktøjer er CUDA fra NVIDIA der forsøger at løse nogle af de ovennævnte begrænsninger. CUDA er en forkortelse for *Compute Unified Device Architecture* og er en hardware/software arkitektur der lader GPU'er³ fra GeForce 8 serien og opefter fungere som generelle stream processorer. CUDA forbedrer datastruktur abstraktionen i forhold til tidligere så programmøren ikke længere skal oversætte sit problemområde til grafik.

CUDA er designet til at lade programmer køre med deres kontrol del på en generel CPU og lade tilsluttede NVIDIA GPU'er fungere som co-processorer for acceleration af SIMD parallel opgaver. Disse opgaver er karakteriseret ved at være uafhængige på den måde at de kan eksekveres af en række parallel tråde. Parallele opgaver gives til GPU'en og eksekveres vha. RPC⁴ [27].

CUDA består på softwaresiden af en række lag som vist på figur 5.6: En hardware driver samt et API bestående af runtime og matematiske biblioteker bl.a. FFT og BLAS [25]. Ud over API'et er der lavet udvidelser til C programmeringssproget. Dette er bl.a. funktions-qualifiers, variabel-qualifiers, eksekveringsdirektiver og globale variable til styring af grid/blokke. Disse er understøttet af en compiler inkluderet i toolkittet.

³Ud over GPU'er findes der også dedikerede beregningskort fra NVIDIA, bl.a. Tesla

⁴Remote Procedure Call



Figur 5.6: CUDA software lag (Kilde: [25]).

Selvom det umiddelbart ser ud til at man programmerer direkte til GPU'en, ligger der et virtualiseringslag kaldet PTX [27]. Dette lag består af en virtuel maskine med tilhørende instruktionssæt som skal abstrahere fra den underliggende hardware. Dette giver en række fordele bl.a.

- Stabilitet i instruktionssættet (lettere at lave høj-niveausprog compilere)
- Uafhængighed af specifik hardwareimplementation
- Skalerbarhed
- Bedre abstraktion

5.5.1 Eksekveringsmodel

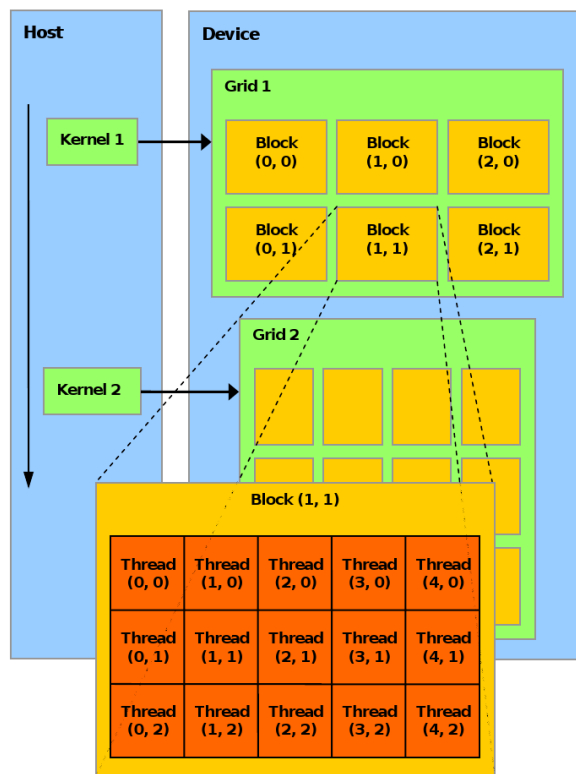
I CUDA betragtes GPU'en som en "parallel thread execution virtual machine- der kan operere med et stort antal tråde, hvori der foretages beregninger på datasæt. Disse tråde er organiseret som et sæt af blokke i grids som vist på figur 5.7.

En blok er en samling tråde der har mulighed for at kommunikere med hinanden igennem delt hukommelse. De har ligeledes mulighed for at synkronisere deres kørsel. Tråde er indekseret inde i en blok vha. et tråd-ID. Dette tråd-ID er en 1-, 2- eller 3-dimensionel vektor, der indekserer ind i blokken. Dette giver mulighed for at have en mere naturlig repræsentation af problemdomænet.

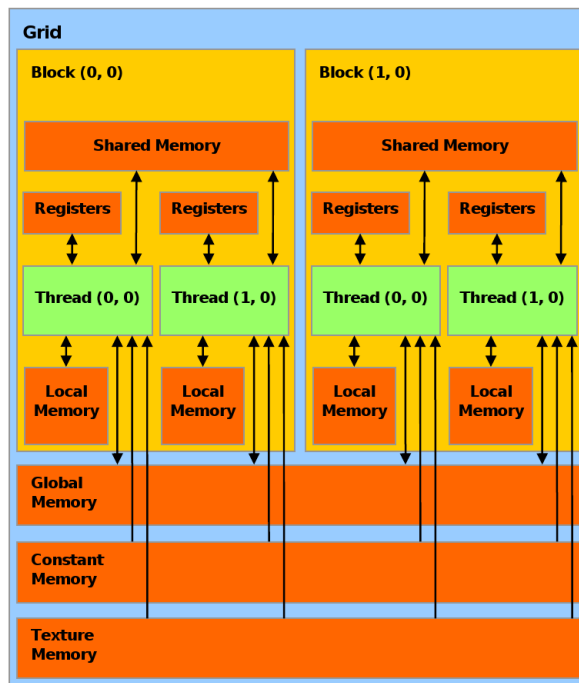
Blokke der har de samme dimensioner og eksekverer den samme kerne, kan samles i et grid. Dette gør at et større antal tråde end der er plads til i en blok, kan startes af en enkelt kerne. Tråde i forskellige blokke kan imidlertid ikke kommunikere og synkronisere f.eks. med hinanden. Blokkene i et grid er indekseret vha. et blok-ID der kan være en 1-,2- eller 3-dimensionel vektor.

5.5.2 Hukommelsesmodel

På trods af CUDA's fleksibilitet er der stadig en række begrænsninger mht. hukommelsestilgang. De tilladte hukommelsestilgange som vist på figur 5.8 er som [25] følgende:



Figur 5.7: CUDA trådningsmodel, her med 2D grid- og blokindeksering (Kilde: [25]).



Figur 5.8: CUDA hukommelsesmodel (Kilde: [25]).

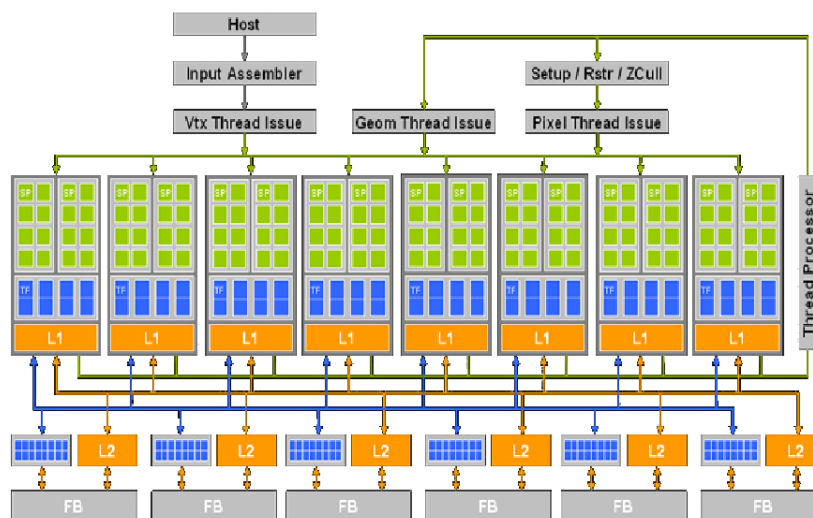
- Læse/Skrive per-tråd *registre*
- Læse/Skrive per-tråd *local memory*
- Læse/Skrive per-block *shared memory*
- Læse/Skrive per-grid *global memory*
- Læse per-grid *constant memory*
- Læse per-grid *texture memory*

Shared memory og registre er on chip. Tekstur, konstant, lokal og global ligger i hovedhukommelsen. Læsning fra tekstur og konstant hukommelse er cachet. Læsninger fra og skrivninger til lokal og global hukommelse er ikke cachet [27, kap. 3.2].

Den globale-, konstant- og teksturhukommelse kan læses og skrives fra host-CPU'en og er persistent imellem flere kerne-kørsler af samme applikationer. For en mere detaljeret beskrivelse af hukommelsen henvises der til [27, kap. 5.1] og [25].

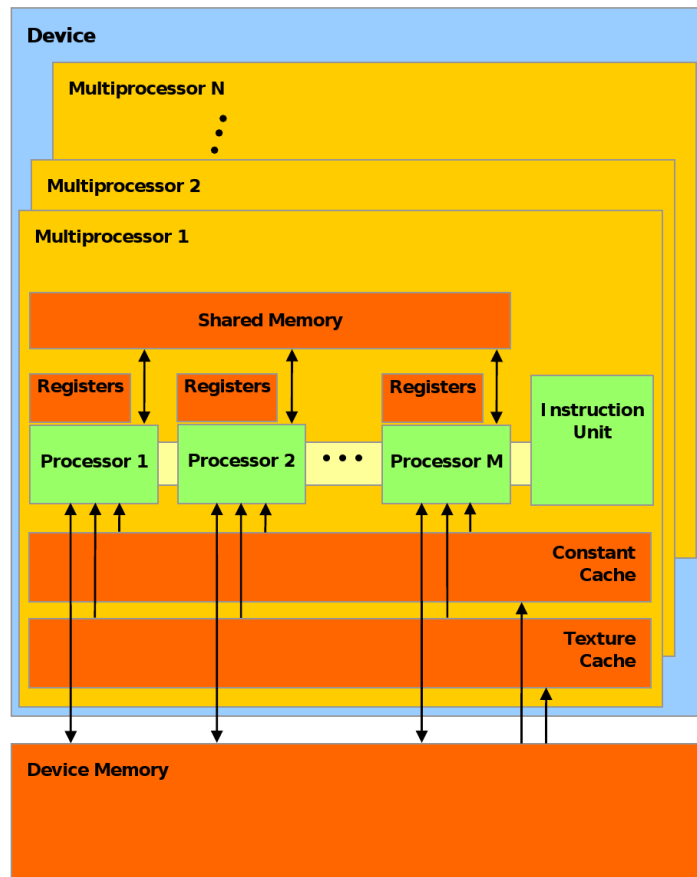
5.5.3 Hardware

Som set fra CUDA består GeForce 8 grafikortet af en række skalar stream processorer (SP'er) der sammen er i stand til at kører et højt antal tråde. SP'erne er højtydende single-precision, streaming-orienterede, floating-point beregningsenheder. På GeForce 8800 GTX modellen er der 128 SP'er, fordelt på grupper af 16 som vist på figur 5.9.



Figur 5.9: GeForce 8800 GTX blokdiagram(Gengivet fra [24]).

Hver gruppe af SP'er kører som en SIMD multiprocessor (MP). På hver SP i en MP gruppe udføres den samme instruktion men på forskellige data. En MP har fire typer hukommelse on-chip: Et sæt af 32-bit registre, en lokal læse/skrive cache (shared memory på figur 5.10) der kan tilgås fra alle SP'er i en MP, en lokal read-only cache af konstanthukommelsen og en lokal read-only cache af teksturhukommelsen. MP'en kan ligeledes tilgå forudtildelte områder af hukommelsen.



Figur 5.10: GeForce 8 multiprocessor (MP) hardware model (Gengivet fra [25]).

Eksekvering af tråde

En multiprocessor kan eksekvere et grid af tråd-blokke (se figur 5.7) ved at skedulere de enkelte blokke til eksekvering. En blok er eksekveret af én enkelt MP samtidig med andre blokke i batches. Formålet med denne batch-tilgang er at optimere ressource-tilgang som f.eks. delt hukommelse og udnyttelse af beregningskraften i SP'erne. Ved at skifte til en anden blok imens at der ventes på hukommelse, kan hukommelseslatency i mange tilfælde gemmes. Dette håndteres af en scheduler som beskrevet i [25, kap. 3.2]. Hvor mange blokke der kan køres i en batch afhænger af, hvor mange registre og hukommelse en givet kerne (dvs. en algoritme) har behov for.

Da der kan være flere tråde i en blok end der er SP'er i en MP, skeduleres trådene også. Trådene organiseres i SIMD grupper kaldet "warps", der kan køres i en single-instruction, multiple-data facon⁵. For eksekveringen af warps i en blok og blokke i et grid er det vigtigt at bemærke at rækkefølgen er udefineret. Warps i en blok kan dog synkroniseres hvorimod blokke i et grid ikke kan.

5.6 GPU platform og ydelse

GeForce 8800 GTX (som vist på figur 5.11) er sammen med den lidt højere clockede version 8800 Ultra de kraftigste GPU'er i GeForce 8 serien. I skrivende stund er 8800 GTX/Ultra blandt de hurtigste grafikkort på markedet [35].



Figur 5.11: GeForce 8800 GTX grafikkort (Gengivet fra [35]).

8800 GTX har følgende specifikationer

- **Kerne:** 128 stream-processorer @ 1.35 GHz
- **Businterface:** PCI Express x16
- **Busbåndbredde:** 4 GiB/sekund i hver retning, Full-Duplex
- **Hukommelse:** 768 MiB GDDR3 RAM⁶ @ 900 MHz (384 bits interface)

⁵Som vist på figur 5.10 modtager alle SP'er i en multiprocessor den samme instruktion

⁶Graphics Double Data Rate 3 Random Access Memory

- **Hukommelsesbåndbredde:** 86.4 GiB/sekund [26]

I peak kan hver SP udføre én Fused-multiply-add (2 FP operationer) per clock. Den teoretiske, floating-point ydelse bliver dermed $2 \text{ op.} \cdot 1350 \text{ MHz} \cdot 128 \text{ SP'ere} = 345.6 \text{ GFLOPS}$. Den reelle ydelse i applikationer vil dog ofte være noget lavere da hukommelsestilgangen sætter en grænse for, hvor hurtigt at SP'erne kan fødes med data. Tabel 5.1 giver et overblik af approksimative tilgangstid for de forskellige hukommelsestyper.

Space	Time	Notes
Registers	0	
Shared	0	
Constant	0	Læsning er cachet
Local	> 100 clocks	
Global	> 100 clocks	
Texture	> 100 clocks	Læsning er cachet

Tabel 5.1: Approksimativ tilgangstid for hukommelsestyper i CUDA (Kilde: [27, kap 6.6]).

Med et større antal tråde end der er SP'er vil skeduleringen være i stand til at starte en hukommelsesoverførsel og derefter skifte til en anden tråd for at udnytte SP'eren. Ligeledes kan man som programmør sørge for at loade variable så hurtigt som muligt i instruktionsrækkefølgen.

8800 GTX er specificeret til at have "compute capability 1.0-i CUDA, se [25, kap 3.3]. Dette betyder blandt andet atomiske operationer i hovedhukommelsen ikke er understøttet, hvilket gør multi-programmering sværere.

I opsætningen af tråde, blokke og grids er der en række begrænsninger i hardwaren der skal overholdes. Følgende tabel fra [25, A.1] indeholder specifikationerne for enheder der understøtter compute capability 1.x:

- Maksimalt antal tråde per blok: 512
- Maksimal størrelse af X-, Y- og Z-dimensioner af trådblokke: 512, 512 og 64
- Maksimal størrelse af alle dimensioner på et grid af trådblokke: 65535
- Warp størrelse: 32 tråde
- Antal registre per multiprocessor: 8192
- Størrelse af shared memory per multiprocessor: 16 KiB
- Samlet constant memory: 64 KiB
- Størrelse af cachen af constant memory i en multiprocessor: 8 KiB
- Størrelse af cachen af texture memory i en multiprocessor: 8 KiB
- Maksimalt antal aktive blokke per multiprocessor: 8
- Maksimalt antal aktive warps per multiprocessor: 24
- Maksimalt antal aktive tråde per multiprocessor: 768
- Maksimal kernestørrelse: 2 millioner instruktioner ⁷

⁷GPU'ens egne instruktioner, ikke PTX

- Hver multiprocessor består af 8 SP'er så en MP er i stand til at processere de 32 tråde i en warp på 4 clock cycles.

5.7 Toolchain

CUDA er som nævnt er udviklet af NVIDIA og er pt. understøttet på 32/64 bit Linux og 32 bit Windows. CUDA består af *CUDA Toolkit*, der indeholder compilere, libc, assemblere, linker, profiler samt BLAS og FFT rutiner samt *CUDA SDK* der indeholder makefiler og kodekesempler. I dette projekt har version 1.0 af toolkittet samt SDK'et til Linux været anvendt.

5.7.1 Compiler

CUDA applikationer består af almindelig C/C++ kode til CPU-plattformen samt specielt C kode til GPU'en. CUDA Compiler Driver (*nvcc*) sørger for at separere CPU- og GPU-kode og give den til henholdsvis CPU-plattformens egen compiler (f.eks. *gcc*) og NVIDIA's C-compiler, *nvopencc*. Når begge er kompileret sørger *nvcc* for at indlejre GPU programmet i CPU'ens så det kan indlæses ved kørsel af CUDA applikationen.

5.7.2 Debugging

CUDA indeholder ikke en debugger der kan køre på GPU'en, men har dog mulighed for at kunne køre i en "emulation"tilstand. I denne tilstand kompileres GPU-koden til og eksekveres på systemets almindelige CPU. Dette gør det muligt at bruge systemets almindelige debugger (f.eks. *gdb*) samt at udføre printf-debugging der ellers ikke er muligt.

Denne funktion er nyttig til at finde fejl i i bl.a. algoritmer, men kommer til kort i andre situationer (bl.a. timing, floating-point præcision og pointer-problemer, se [25]).

5.7.3 Profiler

CUDA indeholder en simpel profiler der gør det muligt at analysere tiden for kørsler af CUDA applikationer. Det er muligt at se tiden for hukommelsesoverførsler imellem CPU og GPU og tiden det tager på både CPU og GPU at eksekvere en kerne.

5.7.4 Testplatform

Projektgruppen har haft adgang til en PC med et GeForce 8800 GTX kort. De vigtigste specifikationer for maskinen er

- Intel Core2 Duo E6750 @ 2.66 GHz
- 2 GiB DDR 2 RAM @ 667 MHz

Denne maskine er blevet anvendt til alle tests udført på CUDA, samt i386 referenceimplementationen.

Kapitel 6

Cell Broadband Engine

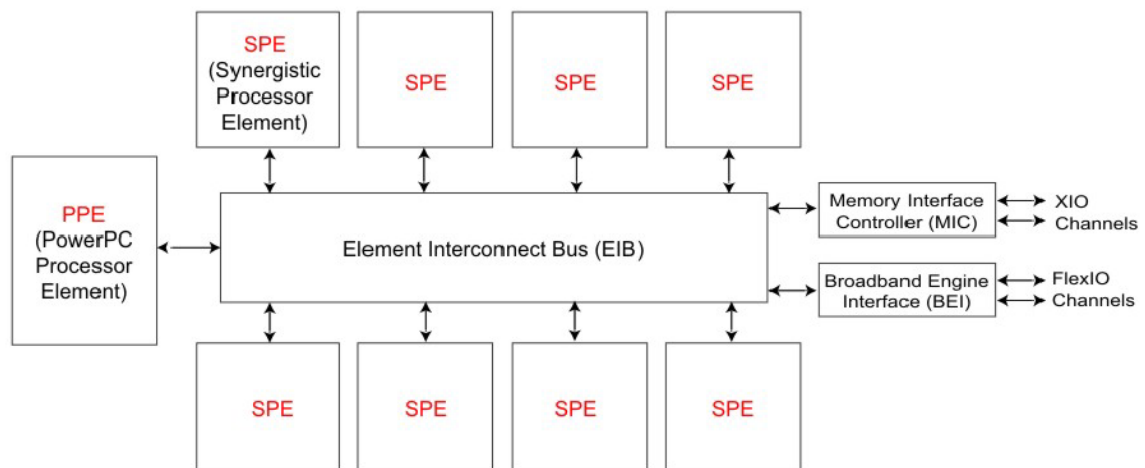
6.1 Introduktion

Cell Broadband Engine (Cell BE) er betegnelsen for en mikroprocessor arkitektur udviklet af en gruppe firmaer, bestående af Sony Computer Entertainment, Toshiba og IBM. Udviklingen af Cell BE har primært været motiveret af den kraftige udvikling som der er forgået indenfor 3D spil industrien. Denne udvikling har betydet at behovet for kraftigere og kraftigere processorer har været støt stigende. I PC verdenen har den udvikling medført frembringelsen af kraftige GPU'er, mens udviklingen på CPU siden har været bundet af kompatibilitetskrav, og derfor ikke har været så innovativ. Spillekonsol markedet er ikke i samme grad bundet af krav om bagud kompatibilitet, og der har derfor været mere fokus på at udvikle konsollernes CPU. Samtidig er en konsol også i sigens natur en mere dedikeret maskine end en PC, hvilket har givet mulighed for yderligere at optimere CPU'en. Samlet set har udviklingen betydet at Cell BE processoren har et radikalt anderledes design, end moderne PC processorer.

Den første applikation der benytter en Cell BE processor er som bekendt Sony's Playstation 3. Men processoren er så kraftig og universel at IBM har et par bladeserver under udvikling, disse er pt. på prototype stadiet. Det er ligeledes muligt at købe specielle indstikskort til almindelige PC'er. Disse er tænkt som accelerator kort til forskellige, primært videnskabelige, applikationer. Prisen på indstikskortene er stadig ret høj, på grund den begrænsede efterspørgelse, så de fleste der benytter Cell BE processorer idag, benytter Playstation 3 platformen. Dette hænger også sammen med at Sony, har valgt at gøre det enkelt at installere et alternativt operativsystem, som f.eks. Linux på Playstation 3.

6.2 Arkitektur

Cell BE processoren er designet som en parallel arkitektur, hvor flere processor elementer arbejder i parallel. I modsætning til de fleste andre parallel arkitekture er alle processor elementer dog ikke "lige", forstået på den måde, at de ikke har lige adgang til ressourcer, som f.eks. hukommelse. Yderligere har de også forskellige egenskaber, udtrykt ved forskellige opbygninger og instruktionssæt. Figur 6.1 viser et overblik over Cell BE arkitekturen.

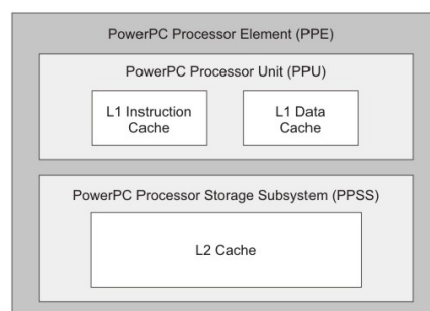


Figur 6.1: Overblik over Cell BE arkitekturen (Billede gengivet fra [14]).

Cell BE består, som vist, af to forskellige processor elementer: et *PowerPC Processor Element* (PPE) og et antal *Synergistic Processor Elements* (SPE). Disse elementer er forbundne via en bus kaldet *Element Interconnect Bus* (EIB). Bussen forbinder ligeledes Cell BE processoren med omverdenen, det vil sige med hovedhukommelsen og ekstern I/O (eller andre Cell BE processorer). I det efterfølgende beskrives de enkelte del nærmere.

6.2.1 PowerPC Processor Element

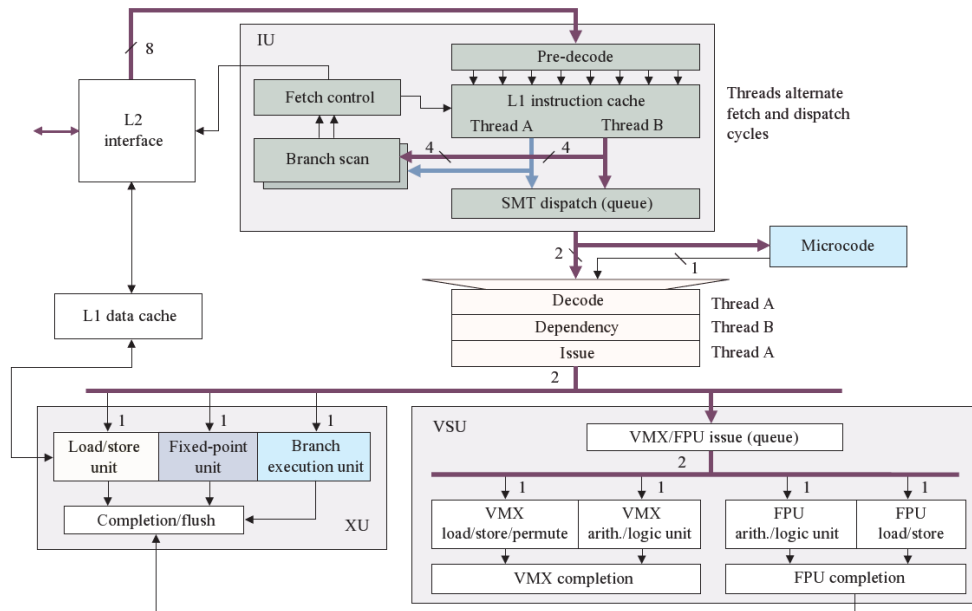
PPE'en, er som navnet antyder, en traditionel PowerPC CPU. PowerPC er en moderne superskalar RISC processor [39], oprindeligt baseret på IBM's POWER arkitektur. PowerPC processoren kommer i en række forskellige udgaver, bestemt for vidt forskellige applikationer, der spænder lige fra embedded systemer til high end server systemer. PPE'en er baseret på PowerPC arkitekturen version 2.02 [12, 13, 11], hvilket indebærer at det er en 64 bit processor, med to kerner (PPE'en er altså i stand til at afvikle to tråde samtidigt). Derudover er PPE'en udstyret med VMX¹ SIMD instruktionssættet med tilhørende registre. Figur 6.2 viser et principdiagram over PPE'en.



Figur 6.2: Overblik over PowerPC Processor Element (Billede gengivet fra [14]).

¹VMX er IBM's betegnelse for AltiVec [31], der er et flydene komma og heltals SIMD/vektor instruktionssæt.

Som det ses indeholder PPE'en to niveau 1 (L1) caches, en til instruktioner og en til data. Disse er begge på 32 KiB². Desuden indeholder PPE'en en niveau 2 (L2) cache til både data og instruktioner på 512 KiB. Figur 6.3 viser et mere detaljeret overblik over PPE'en. Her ses det hvordan PPE'en er delt op i tre blokke.



Figur 6.3: Blokdiagram over en PPE (Billede gengivet fra [20]).

Instruktions enheden (IU) står for at hente instruktioner, dekode dem og sende dem afsted til eksekvering. Den henter 4 instruktioner per tråd af gangen, kontrollerer dem for afhængigheder og sender dem afsted to ad gangen (såkaldt dual issue) til en eksekverings enhed. Desuden indeholder enheden en del branch logik, der benyttes til at prøve at forudsige udfaldet af forgreninger i programmet. PPE'en har to eksekverings enheder. Den ene (XU) tager sig af alle fixed-point instruktioner, samt af alle load/store instruktioner. Den anden (VSU) tager sig af alle floating-point instruktioner, og alle VMX instruktioner. XU enheden indeholder 2 sæt registre (et sæt per kerne), hvert sæt består af 32 registre der er 64 bit brede. VSU enheden indeholder 4 sæt registre (to sæt per kerne). Det ene sæt består af 32 registre, alle 64 bit brede, og disse benyttes ved floating-point beregninger. Det andet sæt består af 32 stk. 128 bit registre, der benyttes af SIMD instruktionerne.

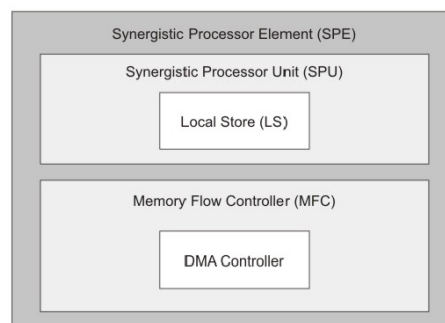
Alt i alt er PPE'en i sig selv en ganske potent allround processor, med omfattende understøttelse af vektor processing via de indbyggede VMX instruktioner. Men det primære formål med PPE'en er at være "orkesterleder", og styre data- og programflow til SPE'erne. I denne rolle har PPE'en desuden den fordel, at den er bygget på en standard arkitektur, hvortil der findes flere operativsystemer, herunder Linux. Det betyder at standard software kan afvikles på PPE'en, og de beregningstunge opgaver kan uddelegeres til SPE'erne.

²1 KiB = 1 Kilo binary Byte = 1024 bytes

6.2.2 Synergistic Processor Element

SPE elementerne er ikke ligesom PPE'en, designet til at være allround processorer. De har eksempelvis en meget stor registerfil (128 stk. 128 bit registre), hvilket ville gøre kontekst skift meget langsomme. Til gengæld er de optimeret til beregningsintensive SIMD applikationer, hvor der skal arbejdes på store datasæt i et forudsigeligt mønster. SPE'erne er opbygget som 128 bit RISC processorer. SPE'rens instruktionssæt er ikke baseret på et eksisterende instruktionssæt, men er i stedet lavet specifikt til formålet. Instruktionssættet, der blandt andet indeholder SIMD instruktioner, er nærmere beskrevet i [17].

Det der helt centralt adskiller en SPE fra en PPE, er dens håndtering af hukommelsen. PPE'en har et traditionelt hukommelses hierarki, bestående af cache (level 1 og level 2), samt hovedhukommelse. SPE'en derimod, har ikke cache, den kan heller ikke tilgå hovedhukommelsen direkte. I stedet har hver SPE sin egen lokale hukommelse, på 256 KiB (foruden sin register fil). Denne hukommelse arbejder ved samme hastighed som selve SPE kernen, der derfor ikke behøver en cache. Netop fraværet af en cache, simplificerer SPE på en række områder. Bla. er der ikke brug for kompliceret logik til at spekulere i hvilke instruktioner og data det kan betale sig at cache. Fraværet af en cache betyder også at SPE'erne opnår en høj grad af determinisme, idet der så ikke længere skal tages højde for eventuelle cache misses. Figur 6.4 viser et overblik over indmaden i en SPE.

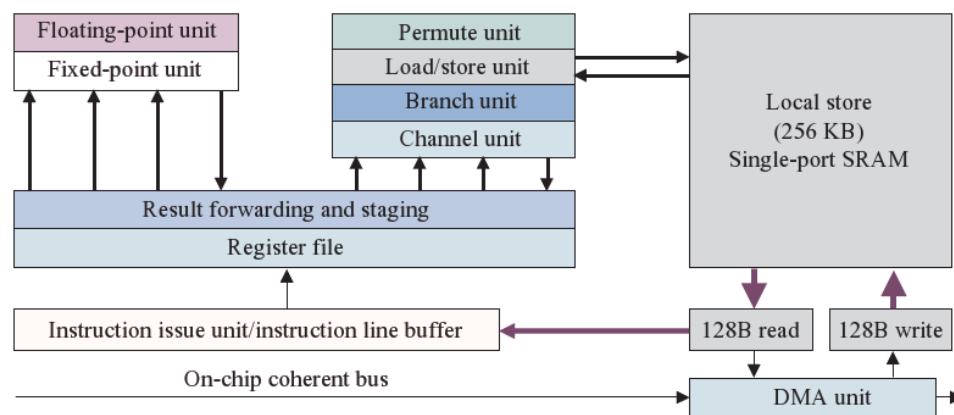


Figur 6.4: Overblik over Synergistic Processor Element (Billede gengivet fra [14]).

Den lokale hukommelse (på figuren kaldet LS) er forbundet med Cell processorens interne bus (Element Interconnect Bus), via en DMA controller. Via denne DMA kan SPE'en asynkront læse og skrive data (og hente instruktioner) fra hovedhukommelsen, men den kan også kommunikere direkte med de andre SPE'er og med PPE'en.

Denne NUMA lignende tilgang til multiprocessing, har en række fordele, men giver også en række udfordringer når det kommer til programudvikling. Den asynkrone DMA enhed betyder at det er muligt effektivt at holde gang i mange hukommelses transaktioner ad gangen. Dette gør det muligt at holde SPE'erne beskæftiget stort set hele tiden. Der opstår med andre ord ikke situationer, hvor processorene skal vente på data fra hovedhukommelsen³. Denne tilgang betyder tilgængæld at det er op til programmøren at styre hukommelsen, i et langt større omfang end det er tilfældet med andre mere konventionelle processorer. Det er f.eks. op til programmøren at overføre data og instruktioner til de enkelte SPE'er, via DMA overførelser. Det er ligeledes op til programmøren at sikre at SPE'erne ikke "løber tør" for data, ved løbende at benytte DMA overførelser. Figur 6.5 gengiver et mere detaljeret indblik i en SPE.

³På en moderne konventionel processor kan denne "memory latency" antage størrelser på helt op til 1000 CPU cykler! Denne performance barriere kaldes populært for "the memory wall".



Figur 6.5: Blokdiagram over en SPE (Billede gengivet fra [20]).

SPE'erne er organiseret omkring deres lokale hukommelse. Denne er implementeret som en single-port SRAM, med en 128 byte bred læse bus og en 128 byte bred skrive bus. DMA overførelserne foregår med 128 byte af gangen, mens kernen læser og skriver 16 bytes (128 bit) af gangen. Denne opdeling af busbredderne, sikre at 7 ud af 8 CPU cykler kan bruges til at skrive/læse data og hente instruktioner fra lokal hukommelsen [20, s. 595]. Som tidligere nævnt er selve kernen bygget op omkring en register fil med 128 registre med en bitbredde på 128. Der kan udstedes op til to instruktioner af gangen, da der findes to eksekveringsenheder. Den ene kan udføre fixed-point og single-precision floating-point instruktioner, og den anden kan udføre load/store, byte permutationer og branch instruktioner. Processoren kan også udføre double-precision floating-point beregninger, men disse er forholdsvis langsomme i denne version af Cell BE arkitekturen. En senere version skulle dog rette op på denne begrænsning. SPE'erne indeholder også en begrænset logik til at forudse udfaldet af forgreninger i programmet, denne logik er dog helt afhængig af hints fra programmøren eller compileren.

6.2.3 Element Interconnect Bus

En forudsætning for at Cell processoren overhovedet kan virke, og at den enorme regnekraft i de enkelte processor elementer kan udnyttes, er naturligvis at data kan transporteres til og fra de enkelte elementer hurtigt nok. Til dette formål implementerer Cell BE arkitekturen en bus kaldet EIB. Oprindeligt var det planen at alle elementer skulle forbindes via en crossbar switch. Men denne plan lod sig ikke gøre, idet en crossbar switch ville optage for meget plads på selve chippen [33]. Den løsning der blev valgt i stedet er baseret på cirkulær bus, bestående af 4 envejs ringe hver med en bredde på 16 bytes. Hver ring kan have 3 transaktioner i gang samtidigt, hvilket gør at bussen kan overføre $3 * 4 * 16 = 192$ bytes samtidigt. Bussen opererer ved det halve af chippens clock frekvens, det betyder at bussen har en peak overførelses hastighed på $192/2 = 96$ bytes per clock.

Bussen forbinder i alt 11 elementer: PPE'en, de 8 SPE'er, *Memory Interface Controlleren* (MIC) og *Broadband Engine Interface* (BEI). Se figur 6.1. MIC'en forbinder Cell processoren med hovedhukommelsen via et såkaldt XIO⁴ interface. Hovedhukommelsen består af Rambus XDR DRAM⁵, som er en RAM type der har en meget lav latency og en meget høj overførelse hastighed. I Cell processoren der har to 32 bit XDR DRAM kanaler, er den teoretiske overførsels hastighed på 25.6 GiB/s.

⁴eXtreme Data Rate Input/Output

⁵eXtreme Data Rate Dynamic Random Access Memory

Cell processoren indeholder også et system interface, kendt som FlexIO, der styres af BEI elementet. Dette interface benyttes til at forbinde Cell processoren med omverdenen, med det kan også benyttes til at forbinde flere Cell processorer i et multiprocessor system.

6.3 Playstation 3

Den nemmeste og billigste måde at få fat i et system baseret på en Cell BE processor, er pt. at købe en Sony Playstation 3 (PS3). Som tidligere beskrevet understøtter denne konsol Linux, hvilket gør den nærmest ideel som platform for eksperimenter med Cell. Projektgruppen har haft adgang til en PS3 med følgende specifikationer:

- **Processor:** Cell BE @ 3.2 GHz.
- **Hukommelse:** 256 MiB XDR DRAM
- **Video:** NVIDIA RSX⁶ m. 256 MiB GDDR3 video RAM.
- **Harddisk:** 40 GiB, 2.5"SATA.
- **Optisk drev:** Blue-ray.
- **Forbindelser:** Gigabit ethernet, Bluetooth 2.0, USB 2.0, HDMI 1.3a, A/V multi out.

En ting der er værd at bemærke, når en PS3 benyttes sammen med linux er, at selv om Cell processoren har 8 SPE'er, så er der kun 6 SPE'er tilgængelige. Dette skyldes at Linux afvikles ovenpå en såkaldt *hypervisor*, et software lag der virtualiserer hardwaren. Denne hypervisor bruger en af SPE'erne, der derfor ikke er tilgængelig. Derudover er en af SPE'erne afbrudt af hensyn til produktionsomkostningerne.



Figur 6.6: Sony Playstation 3.

⁶NVIDIA "Reality Synthesizer", speciel udviklet grafik kort til PS3, baseret på en G71 GPU @ 550 MHz.

6.4 Teoretisk ydelse

Som tidligere beskrevet er en SPE en vektor processor, med en registerbredde på 128 bit. Registerene kan deles op på forskellige måder: 2×64 bit, 4×32 bit, 8×16 bit eller 16×8 bit. Da single precision (SP) floating-point⁷ svare til 32 bit, kan en SPE altså operere på 4 SP floating-point tal af gangen. SPE kernen understøtter såkaldte *Fused-multiply-add* (FMA) instruktioner der kan færdigøres på en clock cycle (når pipelinen er fuld). Den er med andre ord i stand til at udføre 2 SP floating-point operationer per element i vektoren per clock cycle. I alt svare dette til en peak performance på $4 * 2 * 3.2 = 25.6$ GFLOP/s ved en processor hastighed på 3.2 GHz. SPE'erne er uafhængige, idet de har deres egen hukommelse, det betyder at de alle sammen har mulighed for at opnå denne peak performance, så den samlede ydelse bliver $6 * 25.6 = 153.6$ GFLOP/s.

PPE'en understøtter også vektor beregninger via dens VMX instruktionssæt. Dette instruktionssæt operere ligeledes på 128 bit vektorer, og har også FMA instruktioner. Så performance for PPE'en er identisk med en SPE, altså 25.6 GFLOP/s.

Samlet set kan en Cell processor (i en PS3) opnå en samlet performance ved SP floating-point beregninger på $7 * 25.6 = 179.2$ GFLOP/s ved en clock hastighed på 3.2 GHz. Dette tal er dog mere teoretisk end noget andet. Først og fremmest vil man normalt ikke benytte PPE'en som rå regnekraft, men nærmere til at styre SPE'erne og memory flow. Derudover forudsætter beregninger at man kan holde SPE'erne fuldt beskæftiget med at udføre FMA operationer, næppe realistisk eller ønskværdigt. Et realistisk bud på performance ligger derfor noget lavere. Præcist hvor meget lavere, afhænger af den aktuelle applikation og dennes implementation.

Tabellen på figur 6.7 viser en sammenligning mellem en enkelt SPE, en Cell processor, en enkelt processor fra en Cray X1E Supercomputer, en AMD64 PC processor og en Intel IA64 server processor. Disse processorer dækker et bredt udsnit af forskellige typer af arkitekturer. Som det ses har Cell processoren en klar fordel når den kommer til SP floating-point beregninger. Mens den, som forventet, ikke klarer sig helt så godt når det gælder DP floating-point beregninger. En anden ting der kan læses ud af tabellen er at Cell processoren er meget power effektiv. En meget vigtig egenskab, da effektforbrug typisk er en begrænsende faktor for skalerbarheden i en moderne processor. Det skal i øvrig bemærkes at Cell processoren på figuren har 8 SPE'er, og derfor er tallene lidt højere end angivet ovenfor.

6.5 Toolchain

IBM har udviklet et meget omfattende sæt af værktøjer til udvikling på Cell BE processoren, kaldet *Software Development Kit For Multicore Acceleration* (herefter benævnt SDK). Alle disse værktøjer er baseret på Linux platformen. Ydremere er mange af værktøjerne bygget på standard værktøjer, så deres funktion og brug, burde være velkendt for enhver der kender til software udvikling på Linux platformen. I dette projekt er SDK version 3.0 benyttet. Herunder er de vigtigste værktøjer opremset.

6.5.1 Compiler

IBM har udviklet to forskellige compilere til Cell processoren. Denne ene er IBM's egne `xlc` compiler, mens den anden er en Cell udgave af den velkendte `gcc` compiler. I dette projekt er der udelukkende blevet brugt `gcc`.

⁷Da Cell processoren, som tidligere nævnt, i sin nuværende udgave er langsom til double precision floating-point beregninger, er der i projektet udelukkende blevet benyttet single precision floating-point beregninger.

	Cell		X1E	AMD64	IA64
	SPE	Chip	(MSP)		
Architecture	SIMD	Multi-core SIMD	Multi-chip Vector	Super scalar	VLIW
Clock (GHz)	3.2	3.2	1.13	2.2	1.4
DRAM (GB/s)	25.6	25.6	34	6.4	6.4
SP Gflop/s	25.6	204.8	36	8.8	5.6
DP Gflop/s	1.83	14.63	18	4.4	5.6
Local Store	256KB	2MB	—	—	—
L2 Cache	—	512KB	2MB	1MB	256KB
L3 Cache	—	—	—	—	3MB
Power (W)	3	~40	120	89	130
Year	—	2006	2005	2004	2003

Figur 6.7: Sammenligning mellem Cell og forskellige andre arkitekture (Billede gengivet fra [41]).

Da Cell processoren reelt set benytter to instruktionssæt (PPE og SPE), er der ikke kun tale om en compiler, men faktisk to. Et typisk Cell program består derfor af to separate programmer, et til PPE'en og et til SPE'erne. Det er dog muligt at indlejre det linkede SPE program i det linkede PPE program, ved at benytte værktøjet `embedspu`.

For en nærmere beskrivelse af compileren og tilhørende værktøjer henvises til [14, 16].

6.5.2 Debugger

SDK'et indeholder en debugger, der er baseret på den velkendte `gdb`. Denne debugger er i stand til at debugge både PPE kode og SPE kode. Dette fungerer ved at kode der kører på SPE'erne opfattes som separate tråde, og når debuggeren stepper ind i sådan en tråd, skifter den automatisk til en modus der understøtter SPE'ernes instruktionssæt og registre. På denne måde er det muligt at debugge Cell BE kode på en fuldstændig transparent måde.

6.5.3 Profiler

Til at analysere performance indeholder SDK'et en række værktøjer. Til at estimere SPU timing spørgsmål kan værktøjet `spu_timing` benyttes. Dette værktøj giver et billede af hvordan instruktionsstrømmen i en SPU ser ud, ud fra en assembler fil. Der er således tale om en statisk kode analyse.

Til at give et billede af hvordan et program opføre sig runtime, findes der værktøjet `Oprofile`. Dette værktøj laver en profil af programmet, ved at benytte indbyggede hardware performance counters.

Endeligt findes der værktøjet `Performance Debugging Tool (PDT)`, der er i stand til at logge events mens programmet kører. Hvilke events der skal logges bestemmes inden programmet køres, og efterfølgende kan kørslen analyseres ved hjælp af en såkaldt `Visual Performance Analyser`.

6.5.4 Simulator

Har man ikke adgang til en Cell baseret maskine, eller vil man bare gerne udvikle Cell programmer på en almindelig PC, kan man benytte Cell simulatoren `systemsim`. Denne simulator simulerer et fuldt Cell baseret system. Med til simulatoren følger et image af et Fedore Core 7 Linux system. Når

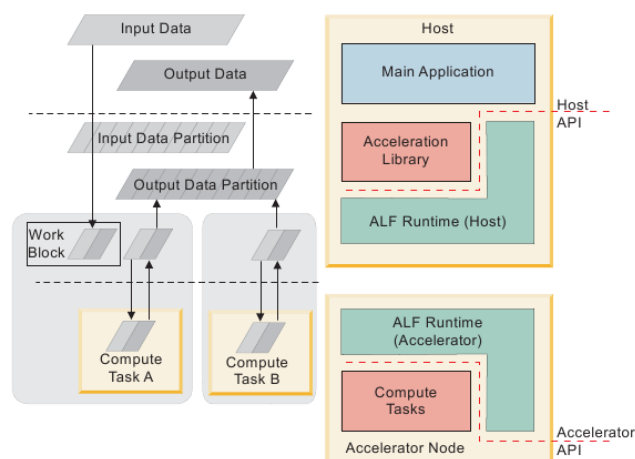
simulatoren kører med dette image indlæst, har man et fuldt funktionsdygtigt Cell Linux miljø til sin rådighed. Desuden indeholder Linux systemet det fulde SDK, så alle værktøjer er tilgængelige. Simulatoren indeholder en mængde faciliteter, f.eks. er det muligt at lave diverse traces af de kørende programmer. Der er også muligt at følge med i hvad de enkelte processor elementer laver, helt ned på register niveau.

6.5.5 IDE

Til at binde alle de forskellige værktøjer sammen har IBM udviklet et IDE⁸ baseret på Eclipse⁹ platformen. Via dette IDE er det muligt at udvikle kode, compilere den og køre det færdige program enten via simulatoren, eller på et fysisk system via en fjern opkobling. IDE'et indeholder desuden mulighed for at debugge/profilere koden. Alt i alt skulle det være muligt at udføre stort set alt udviklingsarbejdet fra IDE'et. Projektgruppen har dog ikke benyttet IDE'et, da der har været problemer med at få det til at virke efter hensigten. Desuden kan SDK'et værktøjer fint benyttes som selvstændige komponenter, sammen med en egnet editor som f.eks. emacs eller vi.

6.5.6 Accelerated Library Framework

Accelerated Library Framework (herefter benævnt ALF), er et framework er der beregnet på at højne abstraktions niveauet når der arbejdes på CELL platformen. Figur 6.8 giver et overblik over ALF.



Figur 6.8: Overblik over ALF (Billede gengivet fra [15]).

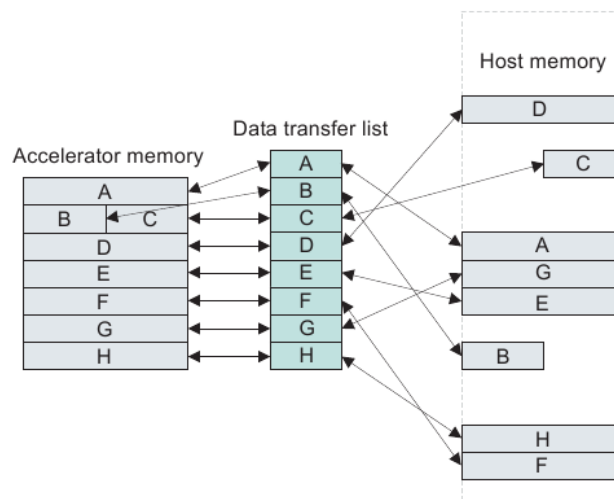
ALF er et generelt framework og skelner som sådan mellem en host og en eller flere accelerator noder I CELL platformenes tilfælde er hosten det samme som PPU'en, mens accelerator noderne er det samme som SPU'erne. Host siden opretter en eller flere compute tasks og knytter en række workblocks til hver task. Hver workblock beskriver et subset af input data og output data som skal processeres, samt eventuelle parametre. Input og output data beskrives ved hjælp af Data Transfer Lists (DTL). Disse beskriver hvordan data skal overføres mellem host hukommelsen og accelerator hukommelsen (se figur 6.9). Herefter udføres disse compute tasks på accelerator noderne. Under

⁸Integrated Development Environment

⁹<http://www.eclipse.org>

afviklingen kaldes en `task` for en `task instance`. Output fra accelerator noderne samles herefter for at danne det færdige resultat. ALF terminologien er opsummeret herunder:

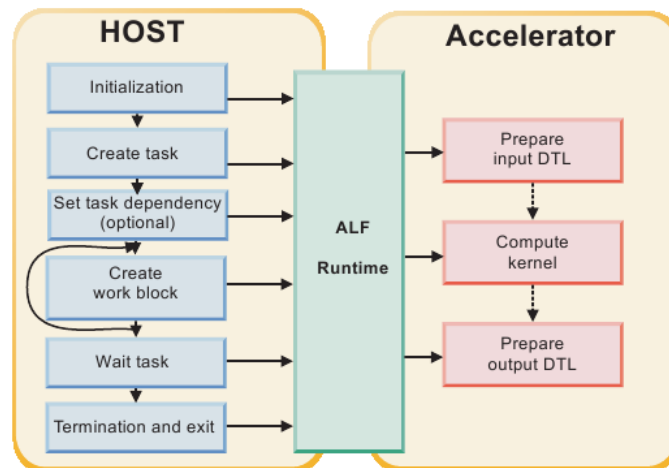
- `Compute task`: En type opgave der ønskes udført på en mængde input data. Eksempelvis en matrix-multiplikation, eller en trekant-trekant test.
- `Workblock`: En del af input data til en compute task.
- `Data Transfer List`: Liste der beskriver hvordan data skal overføres fra host til accerlator noderne.
- `Task instance`: En compute task under afvikling på en accelerator node.
- `Input data partition`: Opdeling af den samlede mængde input data til en compute task i en række workblocks
- `Output data partition`: Indsamling af output data fra en række task instances, sådan det samlede resultat opnåes.



Figur 6.9: Data Transfer Lists (Billede gengivet fra [15]).

Som det ses giver ALF et ganske fornuftigt abstraktion niveau for en lang række af almindelige forekommende problemer. Ydermere har ALF også en række andre fordele. Eksempelvis giver ALF mulighed for automatisk brug af double buffering af data overførsler mellem host og accelerator siden. Dette gør at overførslen af data og beregninger på acceleratorene overlappes, hvilket bidrager væsentligt til udnyttelsen af CELL platformen. Brugeren skærmes også fra direkte kontakt med DMA'erne i CELL processoren. Men de memory krav som DMA'erne stiller er stadig gældende! Derudover har ALF en række mere eller mindre avancerede features. F.eks. er det muligt at lave workblocks der bearbejdes af flere omgange på accelerator noderne. Det er også muligt at specificere inter-task afhængigheder, således at en task ikke køres før alle de andre tasks den afhænger af er blevet afviklet. Endeligt er der også muligt at uddelegere selve partitioneringen til accelerator noder, idet der er mulighed for at disse selv kan opbygge de data transfer lists, der bestemmer hvilke data der skal overføres fra hosten.

Figur 6.10 giver et overblik over hvordan en ALF applikation typisk er sat sammen. På host siden opretten en task, og der tilføjes en række workblocks til denne. Efter dette eksekveres tasken på accelerator noderne. Dette forgår ved at deres `compute` kernel funktion bliver kaldt. Denne funktion står for den egentlige data behandling. Figuren indikere at funktionerne `Prepare input DTL` og `Prepare output DTL` kaldes på accelerator siden. Dette sker dog kun hvis accelerator noderne selv står for at sammesætte deres data transfer list (accelerator side partitionering).



Figur 6.10: ALF applikation (Billede gengivet fra [15]).

ALF er et ganske potent framework, til visse typer af problemer. For at undersøge ALF nærmere er det blevet besluttet at prøve at inddrage det i projektet, ved at implementere massive trekant-trekant og OBB-OBB test med ALF. For en længere og mere uddybende diskussion af ALF henvises til [15].

Del II

Implementation

Kapitel 7

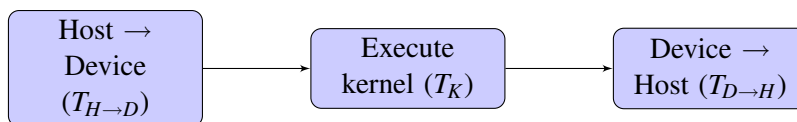
Timing tests af platforme

For at kunne hjælpe evalueringen af algoritmer under designfasen er det ønskeligt at have et bedre kendskab til de forskellige platformes ydeevne. Ydeevnen kan blandt andet være den tid det tager at overføre data til en beregningsenhed, starte den op og få resultater tilbage igen. Følgende afsnit analyserer ydeevnen og forventede kørelstider for de to platforme i projektet, GPU og Cell.

7.1 GPU

7.1.1 Beregningsmæssig model

Der ønskes for GPU'en et lower bound på den tid det tager at overføre data til enheden, køre en kerne og overføre resultaterne tilbage til hovedhukommelsen. Dette flow er vist på figur 7.1.



Figur 7.1: Typisk flow for kørsel på GPU

Modellen for, hvor lang tid ovenstående tager er formuleret som

$$[N_{H \rightarrow D} \times T_{H \rightarrow D}(x_1)] + [N_K \times T_K] + [N_{D \rightarrow H} \times T_{D \rightarrow H}(x_2)] \quad (7.1)$$

x_1 og x_2 er transfer-størrelse i bytes for henholdsvis host-device og device-host overførsler og $[N_{H \rightarrow D}$, N_K , $N_{D \rightarrow H}$ er antal gange at host-device overførsel, kerne kørsel samt device-host overførsel udføres.

7.1.2 Memory

Formålet med de efterfølgende tests er at måle den tid det tager for at overføre data fra og til GPU'en såvel som interne overførsler.

Frem for selv at skrive et program er `bandwidthTest` fra CUDA SDK 1.1 blevet anvendt.

`bandwidthTest` måler båndbredde i MB/sekund for device-host, host-device og device-device overførsler. Programmet gør dette ved at kopiere testdata 10 gange i den ønskede tilstand og derefter tage

gennemsnittet af tiden per kørsel. Ud fra datastørrelsen og tiden per kørsel beregnes båndbredde og rapporteres til brugeren.

Fra programmets kildekode er den relevante sektion

```

1 CUT_SAFE_CALL( cutStartTimer( timer));
2 for( unsigned int i = 0; i < MEMCOPY_ITERATIONS; i++ )
3 {
4     CUDA_SAFE_CALL( cudaMemcpy( h_odata, d_idata, memSize,
5                               cudaMemcpyDeviceToHost ) );
6 }
7
8 CUT_SAFE_CALL( cutStopTimer( timer));
9 elapsedTimeInMs = cutGetTimerValue( timer);
10
11 bandwidthInMBs = (1e3 * memSize * (float)MEMCOPY_ITERATIONS) /
12    (elapsedTimeInMs * (float)(1 << 20));

```

Som det kan ses af koden, beregnes båndbredden i MB/sekund vha. følgende ligning

$$bandwidthinMBs = \frac{1000 \cdot memSize \cdot memcpyiterations}{elapsedTimeMs \cdot 1048576} \quad (7.2)$$

tiden det tog er derfor

$$elapsedTimeMs = \frac{1}{bandwidthinMBs} \cdot \frac{1000 \cdot memSize}{1048576} \quad (7.3)$$

For device-device overførsler ganger programmet båndbredden med 2, da der i timings-løkken både udføres read og write. Dette skal der tages hensyn til når tiden skal isoleres.

bandwidthTest programmet blev kørt med følgende parametre

```

1 ./bandwidthTest --memory=pinned --noprompt --dtoh --mode=range --start=32 --end
   =2048 --increment=32

```

Dette eksempel giver et gennemløb af en range fra 32 til 2048 bytes i 32 bytes increments med device-host overførsler og brug af pinned (non-pageable) hukommelse¹. Pinned hukommelse er blevet brugt til alle tests.

Resultater

For hver af de forskellige typer overførsler er det valgt at køre med tre forskellige ranges af overførsler, for at sikre den bedste detaljeringsgrad.

Til low-range overførsler bruges en høj opløsning (32 bytes increment). Dette svarer godt til størrelse af en trekant repræsenteret med floating point (36 bytes). Low-range kører fra 32 til 2048 bytes.

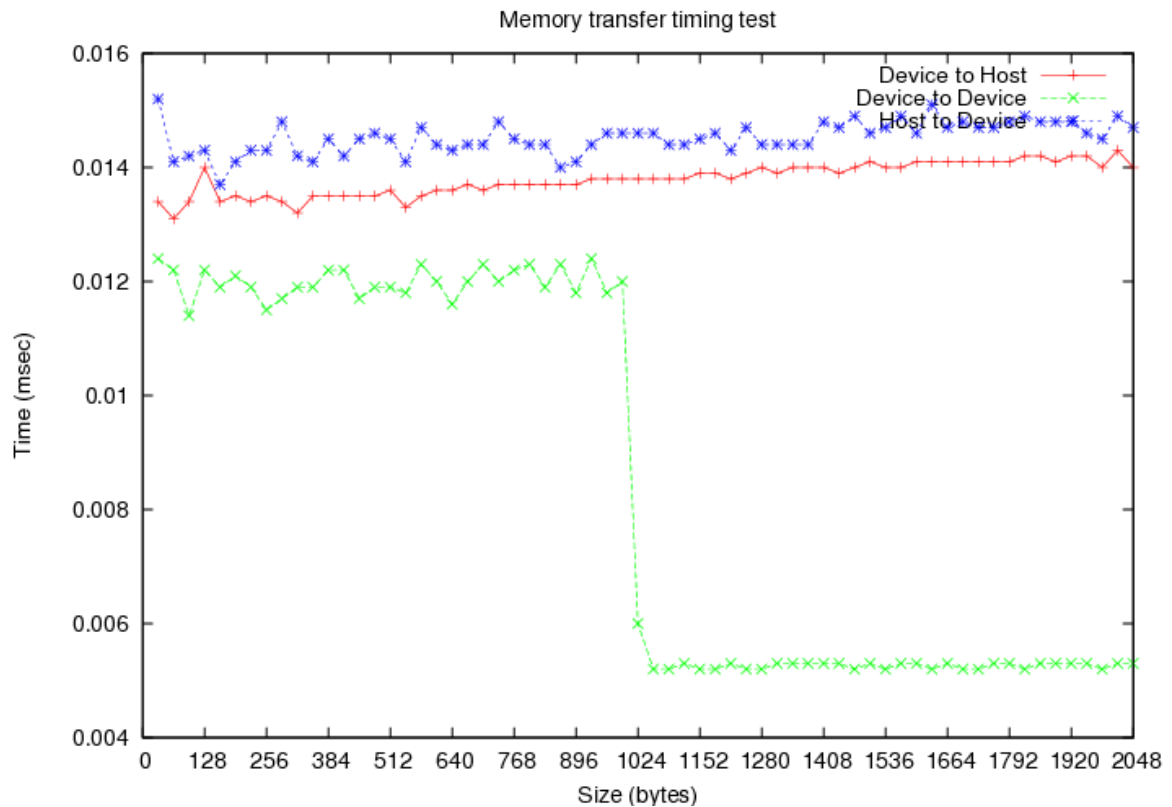
Mid-range overførslerne foretages med 512 bytes increment (svarende til ca. 14 trekanter per step). Mid-range forløber fra 512 til 51200 bytes (14 til 1422 trekanter).

High-range er med 102400 bytes increment fra 102400 til 10240000 bytes (2844 til 284444 trekanter). For at automatisere tests og databehandling er der blevet lavet en række scripts. Disse kan findes i appendix D.1.

¹Pinned hukommelse allokeret med `cudaMallocHost()`

Low-range

Figur 7.2 viser et run fra 32 til 2048 bytes in 32 bytes increment. Tiderne for host-device og device-host er rimeligt symmetriske. Device-host er dog en anelse hurtigere.



Figur 7.2: Memory timing test, fra 32 til 2048 bytes i 32 bytes increments.

En analyse af dataene kan findes i appendix D.2.1.

Mid-range

En analyse af dataen kan findes i appendix D.2.2.

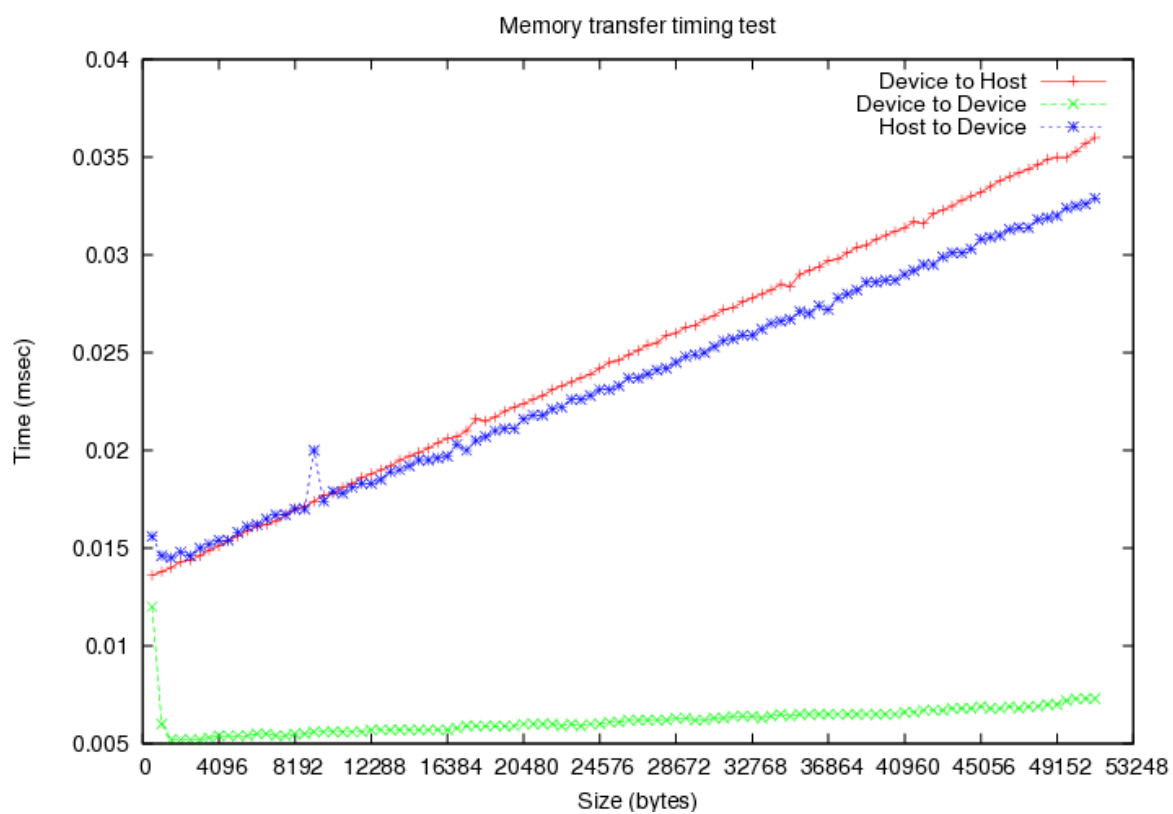
High-range

En analyse af dataen kan findes i appendix D.2.3.

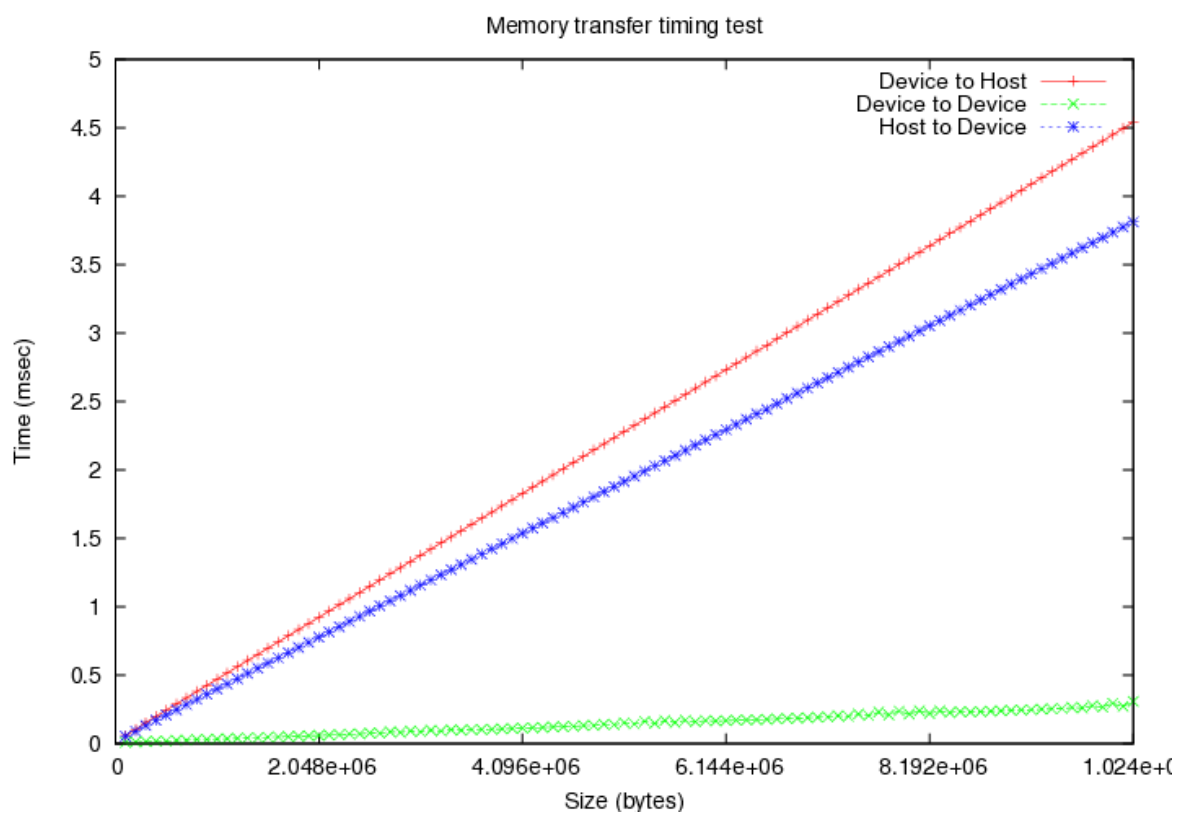
Opsummering af resultater

Ud fra de ovenstående timingstest og analyserne i appendix D er stykvisse lineære approksimationer blevet udformet for den tid $T(x)$ i millisekunder det tager at overføre x bytes.

Device-host



Figur 7.3: Memory timing test, fra 512 til 51200 bytes i 512 bytes increments.



Figur 7.4: Memory timing test, fra 102400 til 10240000 bytes i 102400 bytes increments.

$$T_{d \rightarrow h}(x) = \begin{cases} 4.327 \cdot 10^{-7}x + 0.0133 & \text{if } 32 \leq x \leq 2048 \\ 4.412 \cdot 10^{-7}x + 0.0133 & \text{if } 512 \leq x \leq 51200 \\ 4.437 \cdot 10^{-7}x + 0.005 & \text{if } 102400 \leq x \leq 10240000 \end{cases} \quad (7.4)$$

Host-device

$$T_{h \rightarrow d}(x) = \begin{cases} 2.647 \cdot 10^{-7}x + 0.0142 & \text{if } 32 \leq x \leq 2048 \\ 3.669 \cdot 10^{-7}x + 0.0140 & \text{if } 512 \leq x \leq 51200 \\ 3.724 \cdot 10^{-7}x + 0.005 & \text{if } 102400 \leq x \leq 10240000 \end{cases} \quad (7.5)$$

Device-device

$$T_{d \rightarrow d}(x) = \begin{cases} 0.012 & \text{if } 32 \leq x \leq 1024 \\ 0.005 & \text{if } 1024 \leq x \leq 2048 \\ 2.796 \cdot 10^{-8}x + 0.0055 & \text{if } 512 \leq x \leq 51200 \\ 2.691 \cdot 10^{-8}x + 0.0055 & \text{if } 102400 \leq x \leq 10240000 \end{cases} \quad (7.6)$$

7.1.3 Kerne

Formålet med testene i dette afsnit er at måle den tid det minimum tager for at starte en CUDA kerne og vente på at den terminerer (synkronisering). Dette overhead findes ved at at køre en tom kerne et antal gange og derefter midle resultaterne. På denne måde negligeres den tid det tager at starte/stoppe timerne og initialisere løkken (henholdsvis $T_{timers\text{setup}}$ og $T_{loop\text{setup}}$). Pseudokoden for dette er som følger

```

1 timerStart()
2 for (i = 0; i < N; i++)
3     kernel()
4     sync()
5 timerStop()
6 doAveraging()

```

Tiden det tager for at køre ovenstående er

$$T_{timers\text{setup}} + T_{loop\text{setup}} + N \times (T_{kernel} + T_{looptest}) \quad (7.7)$$

hvor $T_{looptest}$ er tiden det tager at udføre sammenligningen for hver iteration af for-løkken. En mere simpel model af 7.7 er

$$T_{setup} + N \times (T_{kernel}) \quad (7.8)$$

Disse vil blive bestemt eksperimentelt det efterfølgende afsnit.

Resultater

T_{kernel} og T_{setup} findes ved at sammenligne en enkelt kørsel af en kerne $N = 1$ med gennemsnittet af et stort antal kørsler $N = 1000$ hvor setup tiden kan negligeres. Kernen er implementeret så simpelt som muligt

```

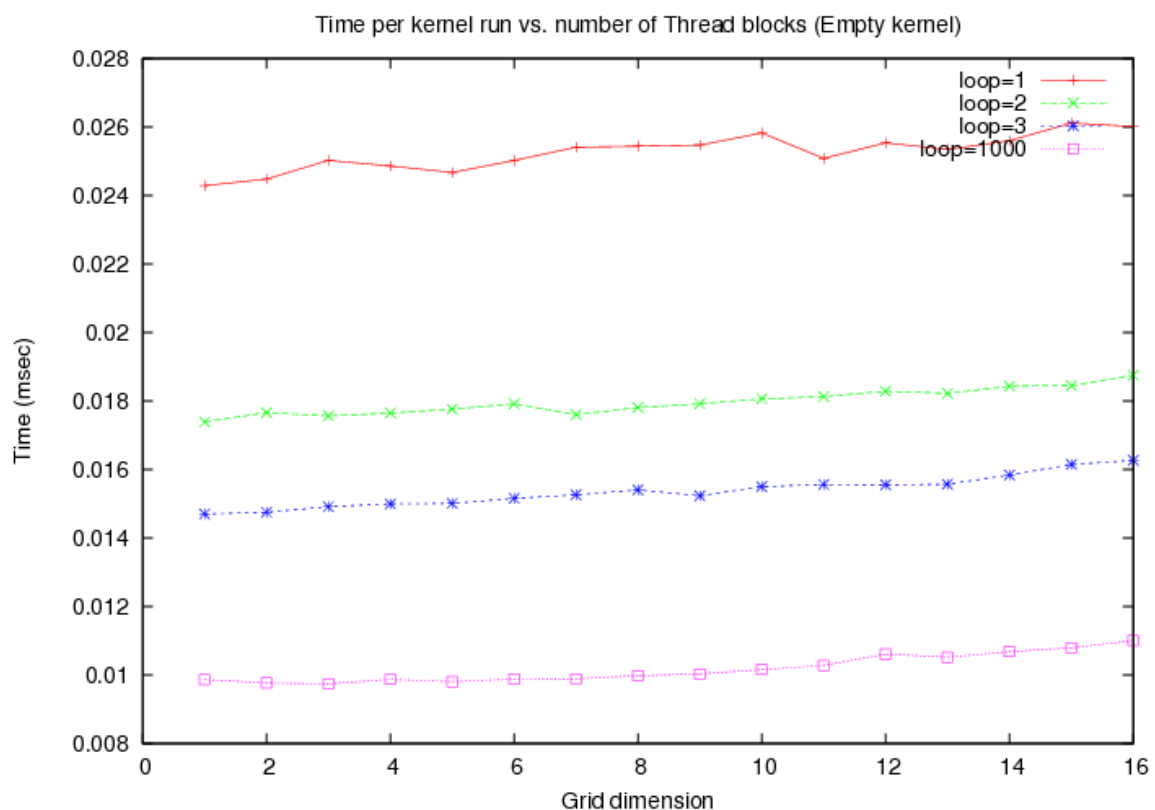
1 /*
2  * Empty kernel function (for invocation test)
3  */
4 __global__ void emptyKernel(void) {
5     return;
6 }

```

CUDA event management API'et i CUDA SDK 1.1 blev anvendt til at udføre timingen.

Figur 7.5 viser tiden det tager per kerne kørsel som funktion af grid-dimensionen.

Grunden til at antal trådblokke varierer i testen, er at det er interessant at vide om der er et ekstra overhead ved at køre flere trådblokke. Jævnfør afsnit 5.5.1, vil der ved et grid på f.eks. 16x16 blive afviklet 256 trådblokke. Disse skal afvikles af de 8 multiprocessorer på Geforce 8800 GTX GPU'en . Da alle trådblokke ikke kan køres samtidig, er det derfor forventeligt at se en stigning i eksekveringstiden som funktion af antal trådblokke.



Figur 7.5: Tid per kerne kørsel ifht. antal tråd blokke. Der køres fra et 1x1 til et 16x16 grid.

Kørslen af et enkelt loop giver $T_{setup} + 1 \times T_{kernel}$, hvilket ud fra figur 7.5 er omkring $24\mu s$.

Den nedre grænse for kerne kørslen fås ved at midle de 1000 kørsler, dette giver $T_{kernel} = 10\mu s$.

Med $T_{setup} = 24\mu s - T_{kernel} = 14\mu s$ er modellen så

$$0.014 + N \times 0.010 \quad (7.9)$$

7.1.4 Delkonklusion for GPU

Ud fra ligning 7.1 og resultaterne fra de foregående afsnit kan et lower bound for den samlede beregningstid beregnes. Sættes $N_{H \rightarrow D} = N_K = N_{D \rightarrow H} = 1$ og overførselsstørrelserne til $x_1 = x_2 = 32$ bytes fås der

$$0.014ms + 0.010ms + 0.012ms = 36\mu s \quad (7.10)$$

Dette svarer til omkring 28000 kørsler per sekund.

7.2 CELL/BE

I dette afsnit præsenteres en række målinger foretaget på CELL processoren. Målingerne har til formål at tilvejebringe realistiske tal til at supplere de teoretiske tal. Håber er hermed at givet et bedre grundlag for forståelsen af denne processor og dens karakteristika.

7.2.1 DMA båndbredde

I teori afsnittes nævnes det at den rå båndbredde på den interne bus i CELL processoren (EIB'en) antager 96 bytes per clock cykle. Da PlayStationen har en clock frekvens på 3.2 GHz , betyder det at den teoretiske båndbredde er på hele $307,2 \text{ GB/s}$. Næppe et tal man skal forvente at se i praksis. Specielt ikke i lyset af at der køre Linux på maskinen, og en del af båndbredden vil derfor gå til operativsystemets og dets programmer, når disse f.esk. kommunikerer med hovedhukommelsen. For at få et mere realistisk bud på den tilgængelige båndbredde, er der derfor blevet foretaget en række målinger. Resultaterne af disse kan ses i de efterfølgende tabeller. Alle målinger er blevet foretaget med programmet `dmabench`, der følger med SDK'et. Herunder er vist hvilke parametre der er benyttet til alle målingerne (den aktuelle måling er en løsning fra en SPU til PPU'en).

```
./dmabench --numspe 6 --numreqs 1 --maxsize 16384 --entrysize 128 seqdmar
```

Programmet laver en række overførelser, med et stigende antal bytes. En overførelse kan enten gå fra PPU'en og til SPU'erne ², eller mellem de enkelte SPU'er ³. En overførelse kan enten være en læsning, en skrivning eller en skrivning efterfulgt af en læsning. Yderligere er der to forskellige metoder der kan benyttes til udføre overførelserne via DMA'erne. Den første af disse er den såkaldte sekventielle metode, hvor en overførelse udføres som en række sekventielle kald til DMA'erne. Med denne metode kan der maksimalt overføres 16384 bytes med et kald. Den anden metode er baseret på DMA lister, hvor der oprettes en liste der beskriver hvad der skal overføres, og et enkelt kald til en DMA udføre så alle overførelserne på listen. Hver liste element er i stand til at overføre op til 16384 bytes, og DMA listerne er således ideelle hvis der skal overføres større mængder data. Desuden er det muligt at udføre „gather“ operationer med DMA lister, hvor data f.esk. hentes forskellige steder i hovedhukommelsen og placeres kontinuert i lokalhukommelsen på en SPU.

²Mere præcist fra hovedhukommelsen til de enkelte SPU'ers lokalhukommelse.

³Imellem de involverede SPU'ers lokale hukommelser.

Tabellen 7.1 viser en række målinger foretaget for læsning, skrivning og læsning/skrivning fra PPU'en til SPU'erne ved brug af den sekventielle metode. Den båndbredde er der angivet i tabellen er den samlede båndbredde for alle seks SPE'er. Det skal endvidere bemærkes at alle målinger er enkeltstående, der er altså ikke tale om gennemsnit eller ligende. Dog er det observeret at målingerne udgør et typisk udsnit blandt flere ens målinger. Som det ses skal der en vis mængde data til før ydelsen begynder at stige markant. Dette skyldes uden tvivl at der er en vis latency forbundet med at sætte DMA overførelserne op. Denne latency overskygges ved større datamængder.

Størrelse	read		write		read/write	
	bytes	μS	GB/s	μS	GB/s	μS
8	0.17	0.2871	0.24	0.2038	0.11	0.4526
16	0.17	0.5759	0.10	0.9193	0.09	1.1095
32	0.17	1.1503	0.13	1.5278	0.08	2.2772
64	0.17	2.3028	0.12	3.2149	0.08	4.5793
128	0.17	4.5925	0.12	6.5535	0.09	8.9209
256	0.17	8.9198	0.13	12.2491	0.09	17.4647
512	0.18	16.6294	0.13	22.8397	0.10	30.4214
1024	0.20	30.0056	0.14	43.0899	0.12	49.1849
2048	0.25	49.7505	0.18	67.2553	0.23	53.6962
4096	0.58	42.6201	0.54	45.4306	0.84	29.0996
8192	1.58	31.0516	1.71	28.7091	2.25	21.8370
16384	3.97	24.7858	4.03	24.3657	4.46	22.0294

Tabel 7.1: PPU til SPU overførelse ved sekventiel brug af DMA'erne

Det ses endvidere at de opnåede båndbredder ligger langt fra den teoretiske båndbredde. Dette er dog forventeligt, selv om det måske var forventet at få en lidt højere ydelse. Desuden ser det ud til at den højeste ydelse for overførelse mellem PPU og SPU med den sekventielle metode, opnåes ved en blokstørrelse på omkring 2KiB.

Tabel 7.2 viser den samme måling som før, her er der blot tale om overførelse SPU'erne imellem. Igen ses at der er en vis latency, som dominerer ved små overførelser. Men i modsætning til den sekventielle PPU-SPU overførelse fra før, falder performance ikke ved større blokke end 2KiB. Generelt er båndbredden også højere end før. Dette mønster går igen i senere målinger: Tilsyneladende er båndbredden mellem SPU'erne større end mellem PPU og SPU'erne, selv om de sidder på den samme fysiske bus. Hvad forklaringen er på dette er svært at sige, men en forklaring kan være at PPU har andet at lave end blot at styre DMA overførelserne (f.eks kontekst skift osv.).

Tablel 7.3 viser målinger foretaget mellem PPU'en og SPU'erne under anvendelse af DMA lister. Hver element i DMA listerne er i denne måling på 128 bytes, dette betyder at den mindste størrelse der kan overføre netop er 128 byte (et element). Hvis båndbredden for denne størrelse sammenlignes med båndbredden for samme størrelse i tabel 7.1, ses det at der tilsyneladende ikke er noget særligt overhead forbundet med at benytte DMA lister i forhold til at benytte DMA'erne sekventielt. Desuden ser det ud til at performance ligeledes falder når blok størrelse bliver større end 2KiB når der benyttes DMA lister. DMA lister kan altså benyttes uden nævneværdig nedgang i performance. Det skal dog nævnes at ovenstående tal bygger på at data ligger kontinuert i hukommelsen, de siger altså ikke noget om hvordan en evt. „gather“ operation vil påvirke resultatet.

Anderledes forholder det sig med SPU-SPU overførelser ved brug af DMA lister. Tabel 7.4 viser en måling af dette. Igen er et element på 128 bytes. Som det ses er læse og skrive hastigheden for de små

Størrelse	read		write		read/write	
	<i>bytes</i>	μS	GB/s	μS	GB/s	μS
8	0.08	0.5767	0.08	0.6117	0.04	1.0714
16	0.08	1.1540	0.08	1.2603	0.04	2.2291
32	0.08	2.3086	0.08	2.5214	0.04	4.4303
64	0.08	4.6173	0.08	5.0455	0.04	8.8821
128	0.08	9.2345	0.08	10.0828	0.04	17.8417
256	0.09	17.3044	0.08	18.6944	0.05	33.2927
512	0.10	31.0901	0.09	33.2176	0.05	58.3912
1024	0.12	51.6005	0.11	53.5155	0.09	66.1067
2048	0.16	76.0436	0.17	72.9872	0.16	77.7315
4096	0.25	99.6865	0.29	83.7389	0.30	81.9029
8192	0.43	114.7383	0.56	87.4804	0.58	84.5148
16384	0.77	127.8743	1.09	90.1012	1.15	85.2989

Tabel 7.2: SPU til SPU overførelse ved sekventiel brug af DMA'erne

Størrelse	read		write		read/write	
	<i>bytes</i>	μS	GB/s	μS	GB/s	μS
128	0.18	4.2170	0.12	6.2410	0.09	8.3801
256	0.20	7.6872	0.15	10.3963	0.10	15.3825
512	0.24	13.0270	0.18	17.0457	0.13	24.5432
1024	0.31	20.1421	0.25	24.8437	0.18	34.1587
2048	0.47	26.3385	0.40	30.3555	0.29	42.9986
4096	0.99	24.7497	1.01	24.3987	0.79	31.2445
8192	2.00	24.5225	2.02	24.3711	2.21	22.2163
16384	3.94	24.9466	4.05	24.2777	4.44	22.1585

Tabel 7.3: PPU til SPU overførelse ved brug af DMA lister (128 byte element størrelse)

overførelser at sammenligne med den sekventielle overførelse (se tabel 7.2), mens de for de større blokstørrelser er noget lavere. Dette gælder dog mærkeligt nok ikke for den kombinerede læse og skrive test. Det er ikke lykkedes at finde en forklaring på dette resultat.

Størrelse	read		write		read/write	
	bytes	μS	GB/s	μS	GB/s	μS
128	0.10	7.5943	0.10	8.0200	0.05	14.5286
256	0.12	13.1305	0.11	13.7336	0.06	25.7957
512	0.15	20.1159	0.15	20.9139	0.08	38.2741
1024	0.22	27.6272	0.22	28.2806	0.12	52.6440
2048	0.36	33.7569	0.36	34.2601	0.19	65.9880
4096	0.64	38.1921	0.64	38.5058	0.33	74.4511
8192	1.21	40.7557	1.20	40.9415	0.62	79.5497
16384	2.33	42.1847	2.32	42.2839	1.19	82.7132

Tabel 7.4: SPU til SPU overførelse ved brug af DMA lister (128 byte element størrelse)

Ovenstående målinger på DMA liste overførelser, er som nævnt alle begået med en element størrelse på 128 bytes. For at se hvordan DMA'erne opføre sig med varierende element størrelser, er der blevet fortaget to sæt af målinger, et sæt mellem PPU'en og SPU'erne og et sæt mellem SPU'erne.

Tabel 7.5 viser resultatet af målingerne mellem PPU'en og SPU'erne. Det ses at der er lidt performance at hente ved at benytte større elementer, specielt ser 2KiB ud til at være en god element størrelse.

Tabel 7.6 vise samme måling udført mellem SPU'erne. Her ses igen at overførelserne mellem SPU'erne er hurtigere, end overførelse mellem PPU'en og SPU'erne. Desuden ser det ud til at 4KiB element størrelse er et godt kompromis. Dette giver anledning til nogle særdeles høje båndbredder, helt op til omkring 185 GB/s , hvilket må siges at være særdeles imponerende.

Alt i alt må det siges at de målte båndbredder ligger et godt stykke fra de teoretisk mulige, selv om enkelt af dem er ganske imponerende. Generelt ser det ud til at man kan forvente en overførelse-hastighed på en 20-30 GB/s for de mest almindelige forkommende overførelser, nemlig mellem PPU og SPU'erne. Kan man indrette sit problem så de fleste overførelser sker mellem SPU'erne alene, kan man forvent en større båndbredde. Dette er dog svært i praksis. I stedet skal der nok satses på at overlappes overførelser og beregninger på SPU'erne, for at maksimere throughput i applikationen.

7.2.2 Signalering mellem PPU og SPU

Ud over at måle diverse DMA overførelses hastigheder, er det også blevet målt hvor hurtigt det er muligt at signalerer imellem PPU'en og de enkelte SPU'er. Dette mål er væsentligt idet det er nødvendigt at signalerer til SPU'erne når der er arbejde der skal udføres. Omvendt skal SPU'erne også kunne signalerer til PPU'en når de er færdige med et stykke arbejde. Den typiske metode til denne form for signalerering i CELL processoren er at benytte mailboxes. Disse tillader at sende korte besked-er (32 bit) imellem PPU'en og SPU'erne. En anden nyttig feature ved mailboxes er at de kan blokere indtil der ankommer en meddelelse, hvilket gør dem velegnede til den form for signalerering der er omtalt ovenfor. Mailboxes fungerer lige ud af landevejen, en besked lægges i en mailbox på den ene side (f.eks PPU siden), og kan derefter læses af den anden side (f.eks SPU siden).

Herunder er vist et udpluk af den testkode der er anvendt på PPU siden for at måle signalerings hastigheden. Funktionen `_spe_in_mbox_write`, skriver en besked i en SPU mailbox ⁴ Dernæst

⁴Den her benyttede funktion skriver direkte til den enkelte SPU's såkaldte Control Area. Dette er hurtigere end at

<i>Elementstørrelse: 2048 bytes</i>						
Størrelse	read		write		read/write	
<i>bytes</i>	μS	<i>GB/s</i>	μS	<i>GB/s</i>	μS	<i>GB/s</i>
2048	0.26	47.7285	0.20	62.7774	0.23	54.4214
4096	0.44	55.8498	0.43	56.6947	0.77	31.9938
8192	0.95	51.7720	1.52	32.3905	2.22	22.1584
16384	3.03	32.4430	4.04	24.3497	4.44	22.1328
<i>Elementstørrelse: 4096 bytes</i>						
Størrelse	read		write		read/write	
<i>bytes</i>	μS	<i>GB/s</i>	μS	<i>GB/s</i>	μS	<i>GB/s</i>
4096	0.66	37.0078	0.42	58.7615	0.65	38.0538
8192	1.96	25.0167	1.62	30.3555	1.78	27.6724
16384	3.95	24.8880	4.03	24.3671	4.45	22.0973
<i>Elementstørrelse: 8192 bytes</i>						
Størrelse	read		write		read/write	
<i>bytes</i>	μS	<i>GB/s</i>	μS	<i>GB/s</i>	μS	<i>GB/s</i>
8192	1.58	31.1312	1.52	32.2520	1.76	27.9292
16384	3.93	25.0252	4.05	24.2500	4.43	22.1740
<i>Elementstørrelse: 16384 bytes</i>						
Størrelse	read		write		read/write	
<i>bytes</i>	μS	<i>GB/s</i>	μS	<i>GB/s</i>	μS	<i>GB/s</i>
16384	3.51	27.9704	3.50	28.0788	3.97	24.7799

Tabel 7.5: PPU til SPU overførelse ved brug af DMA lister (varierende element størrelse)

<i>Elementstørrelse: 2048 bytes</i>						
Størrelse	read		write		read/write	
<i>bytes</i>	μS	<i>GB/s</i>	μS	<i>GB/s</i>	μS	<i>GB/s</i>
2048	0.18	67.0560	0.19	66.4126	0.16	74.4935
4096	0.26	94.7499	0.30	81.6358	0.31	79.9823
8192	0.44	112.4521	0.56	87.3830	0.58	84.7247
16384	0.75	130.4580	1.07	91.8865	1.15	85.1215
<i>Elementstørrelse: 4096 bytes</i>						
Størrelse	read		write		read/write	
<i>bytes</i>	μS	<i>GB/s</i>	μS	<i>GB/s</i>	μS	<i>GB/s</i>
4096	0.26	93.4778	0.30	81.3874	0.17	148.3857
8192	0.43	114.3148	0.58	85.1785	0.27	184.5513
16384	0.77	127.2210	1.10	89.5969	0.56	175.0714
<i>Elementstørrelse: 8192 bytes</i>						
Størrelse	read		write		read/write	
<i>bytes</i>	μS	<i>GB/s</i>	μS	<i>GB/s</i>	μS	<i>GB/s</i>
8192	0.61	80.2495	0.43	113.0735	0.58	84.5998
16384	1.22	80.7770	0.83	118.2137	1.14	86.3078
<i>Elementstørrelse: 16384 bytes</i>						
Størrelse	read		write		read/write	
<i>bytes</i>	μS	<i>GB/s</i>	μS	<i>GB/s</i>	μS	<i>GB/s</i>
16384	0.84	116.6290	1.10	89.1320	1.15	85.1693

Tabel 7.6: SPU til SPU overførelse ved brug af DMA lister (varierende element størrelse)

kaldes funktionen `_spe_out_mbox_read` der blokere indtil der ankommer et svar fra SPU'en. Det skal bemærkes at denne funktion blokere ved kontinuert at polle status registret i SPU'en. Dette er naturligvis ikke optimalt, ikke blot benyttes CPU tid på denne polling, men den spilder også båndbredde da det er PPE'en der poller SPU'ens status register.

```
1 clock_gettime(CLOCK_THREAD_CPUTIME_ID, &t0);
2 _spe_in_mbox_write(psa, 0);
3 result = _spe_out_mbox_read(psa);
4 clock_gettime(CLOCK_THREAD_CPUTIME_ID, &t1);
```

Den tilhørende SPU kode er vist herunder. Først kaldes funktionen `spu_read_mbox`, der blokere indtil PPU skriver til mailboxen. Dernæst skrives med det samme et svar til PPU'en med funktionen `spu_write_out_mbox`.

```
1 result = spu_read_in_mbox();
2 spu_write_out_mbox(0);
```

Med ovenstående kode er den begået en række målinger. Resultatet af disse er en smule overraskende. Hvis koden udføres som vist måles en roundtrip tid på omkring 20-25 μ s, hvilket må siges at være meget. Hvis man derimod modificere koden så der udføres flere på hindanden følgende roundtrips, viser det sig at det kun er første kald der har en usædvanlig høj latency. På de efterfølgende kald måles en roundtrip tid på omkring 1-2 μ s. Undersøges den høje latency på det første kald viser det sig at det primært skyldes kaldet til `_spe_in_mbox_write`, men hvorfor det forholder sig sådan er det ikke lykkedes at finde en forklaring på.

Som nævnt tidligere basere det blokerende kald i PPU koden vist ovenfor sig på polling. Det kan dog undgås ved at benytte en anden funktion, kaldet `spe_out_intr_mbox_read` der istedet for polling benytter et interrupt. For at undersøge denne metode er følgende PPU kode blevet benyttet til at udføre en timing test.

```
1 clock_gettime(CLOCK_THREAD_CPUTIME_ID, &t0);
2 _spe_in_mbox_write(psa, 0);
3 spe_out_intr_mbox_read(ctx, &result, 1, SPE_MBOX_ALL_BLOCKING);
4 clock_gettime(CLOCK_THREAD_CPUTIME_ID, &t1);
```

Den tilsvarende SPU kode er vist herunder.

```
1 result = spu_read_in_mbox();
2 spu_write_out_intr_mbox(0);
```

Igen viser det sig at det første roundtrip er langt dyrere end de efterfølgende. Første roundtrip tager omkring 70 μ s, mens de efterfølgende roundtrips tager omkring 6-7 μ s. Det er altså dyrere at benytte den interrupt baserede model, til gengæld sparer man CPU tid og båndbredde.

7.2.3 Delkonklusion for CELL

I dette afsnit er der blevet målt på nogle af de rå parametre for en CELL processor. Det er blevet fundet at den praktiske båndbredde for overførelse af data mellem PPU'en og SPU'erne er del-lavere end den teoretiske båndbredde. Men det overrasker alligevel at der er en faktor 10 til forskel. En anden overraskelse er at der er en ret høj latency på den første besked der sendes igennem en mailbox, mens roundtrip tiden falder med en faktor 10 for de efterfølgende beskeder. Ud over de målinger der beskrevet her, er det også blevet fortaget en række målinger på bla. Libspe. Disse er dog ikke medtaget her, idet det blev vurderet at de var knap så relevante.

benytte Libspe funktionerne der skriver via en context (f.eks `spe_in_mbox_write`).

Kapitel 8

Framework og library

8.1 Indledning

Fokus i specialet har fra starten af ikke blot været at afprøve nogle kollisions algoritmer på parallelle arkitekturer, men i lige så høj grad at skabe noget kode, der kan danne basis for en praktisk anvendelse. Det har derfor været nødvendigt at bruge tid på at skabe den hel del utility kode, foruden selve kerne algoritmerne. Desuden er der blevet brugt en del tid på at integrere alle dele i et samlet kode hierarki. I dette kapitel vil organiseringen af den samlede kodebase blive beskrevet.

8.2 Organisering af kode

Den centrale komponent i kodebasen er et library kaldet LibOPP ¹. Dette library implementere to forskellige typer kollisions tests. Den første type test er i stand til at udføre en udtømmende (exhaustive) test mellem to „supper“ af enten trekanter eller OBB'er. Den anden type test kan teste en model af en scene for kollision. Scenen er opbygget af en række objekter, der hver især er repræsenteret ved deres OBB'er og trekanter samt et træ der beskriver sammenhængen mellem disse. Selve library'et er kendetegnet ved dets interface, men det findes faktisk i tre forskellige implementationer, nemlig en der er beregnet på I386 platformen, en der er beregnet på CELL platformen, og endelig en der er beregnet på CUDA platformen. Hvordan den enkelte platform implementerer funktionaliteten beskrevet ved interfacet er vidt forskelligt, men funktionaliteten er ikke desto mindre den samme for alle platforme. Ud over selve LibOPP, indeholder koden også en del test kode, samt en masse funktionalitet til at understøtte hele projektet. Herunder er vist hvordan projektet er struktureret på biblioteksniveau.

```
|-- algorithm
|   |-- obb_tests
|   |-- other_tests
|   `-- triangle_tests
|-- glut_viewer
|-- include
|   |-- cell
|   |-- cuda
```

¹OPP er en forkortelse for Offensus Probatur Parallelus.

```

|   `-- i386
|-- libopp
|   |-- cell
|   |   |-- include
|   |   |-- spu_model_test
|   |   `-- spu_primitive_test
|   |       `-- tools
|   |-- cuda
|   |   |-- cuda_model_test
|   |   |-- cuda_primitive_test
|   |   |-- include
|   |   |-- symlinks
|   |   `-- tools
|   `-- i386
|-- testdata
|-- tools
`-- util

```

Herunder følger en kort gennemgang af kildekode filerne i projektet, delt op efter hvilke bibliotek de tilhøre.

Rod biblioteket

Dette bibliotek udgøre roden i projektet, og indeholder som sådan en række filer den binder projektet sammen.

Fil	Notes
main.c	Projektets hoved fil
run_container_test.c	Test kode til containere (queue, list, stack og tree)
run_libopp_cell_test.c	Test kode til CELL implementation af udtømmende tests
run_obb_test.c	Test kode til OBB test algoritmerne
run_thread_test.c	Test kode til CELL, CUDA og i386 implemetationen af model tests
run_tri_test.c	Test kode til trekant test algoritmerne

algoritm/obb_tests

Under dette bibliotek findes der to typer af OBB test algoritmer. Den første er en udtømmende test der tester alle kanter i den ene OBB mod alle flader i den anden. Den anden algoritme er den såkaldte Gottschalk algoritme, som er mindre ressourcekrævende end den første.

Fil	Notes
exhaustive.c	Exhaustive udgaven af OBB test algoritmen
exhaustive.h	Header til exhaustive.c
gottschalk.c	Gottschalk udgaven af OBB test algoritmen
gottschalk.h	Header til gottschalk.c
obb_obb_test.c	Public interface til OBB test algoritmen
obb_obb_test.h	Header til obb_obb_test.c
test_obbs.h	Header indeholdende test data til OBB test algoritmerne

algorithm/other_tests

Dette bibliotek indeholder en række under algoritmer, der benyttes af de visse af de andre algoritmer.

Fil	Notes
line_obb_face_test.c	Algoritme til at teste om en linie går igennem et plan i rummet
line_obb_face_test.h	Header til line_obb_face_test.c
line_tri_test.c	Algoritme til at teste om en linie går gennem en trekant
line_tri_test.h	Header til line_tri_test.c
segment_obb_face_test.c	Algoritme til at teste om et linie segment går igennem en OBB face
segment_obb_face_test.h	Header til segment_obb_face_test.c
segment_plane_test.c	Algoritme til at teste om et linie segment går igennem et plan i rummet
segment_plane_test.h	Header til segment_plane_test.c
segment_tri_test.c	Algoritme til at teste om et linie segment går igennem en trekant
segment_tri_test.h	Header til segment_tri_test.c

algorithm/triangle_tests

I dette bibliotek findes kildekoden til de forskellige trekant test algoritmer. De to hoved algoritmer er segment-piercing og devillers. Den første af disse er en naiv og simpel algoritme, mens devillers er mere kompliceret. Devillers findes i to udgaver, en der læner sig op ad den originale implementation, samt en der er modificeret så den er mindre krævende mht. antal branches. Desuden findes der også i dette bibliotek kode til at håndtere de koplanare tilfælde.

Fil	Notes
devillers.c	Modificeret udgave af devillers trekant test algoritmen
devillers.h	Header til devillers.c
devillers_coplanar.c	Algoritme til at håndtere det koplanare tilfælde for devillers algoritmerne
devillers_coplanar.h	Header til devillers_coplanar.c
devillers_orig.c	Devillers trekant test algoritmen, implementeret efter den originale kode
devillers_orig.h	Header til devillers_orig.c
segment_piercing.c	Segment piercing trekant test algoritmen
segment_piercing.h	Header til segment_piercing.c
test_triangles.h	Header indeholdende test trekanter til test af trekant algoritmerne
tri_tri_test.h	Header med public interface til trekant algoritmerne

glut_viewer

Glut_viewer er en simpel test applikation der er i stand til at vise en scene, med tilhørende objekter i GLUT (OpenGL). Selve scenen gengives som et triangle mesh, hvor kollisioner markeres med rødt.

Fil	Notes
main.c	Hoved fil til GLUT viweren
zpr.h	Header til zpr.h
zpr.h.c	Zoom, Pan and Rotate library til GLUT (se: http://www.nigels.com/glt/gltzpr/)

include

Include biblioteket er fælles for alle platforme. Men indeholder også en række underbiblioteker der er platformsafhængige.

Fil	Notes
common_conf.h	Fælles indstillinger for alle platformene

include/cell

Dette include bibliotek indeholder CELL specifik kode og indstillinger. Disse omfatter blandt andet geometriske definitioner der udnytter CELL platformes vektor typer. Desuden er en række matematiske funktioner implementeret så de udnytter disse vektor funktioner. Visse datastrukturer er ligeledes tilpasset i størrelse så de lettere kan bruges på CELL platformen.

Fil	Notes
config.h	Diverse indstillinger og defines til CELL platformen
geometry.h	Geometriske definitioner til CELL platformen
matrix_math.h	Matrix operationer og homogene transformations operationer til CELL platformen
opp_types.h	Diverse type definitioner til CELL platformen
vec_math.h	Vektor matematik funktioner og almindelige matematiske funktioner til CELL platformen

include/cuda

De CUDA specifikke includes ligger i dette bibliotek. Filerne og indholdet er essentielt det samme som include/cell, dog med CUDA specifikke optimeringer.

Fil	Notes
config.h	Diverse indstillinger og defines til CUDA platformen
geometry.h	Geometriske definitioner til CUDA platformen
matrix_math.h	Matrix operationer og homogene transformations operationer til CUDA platformen
opp_types.h	Diverse type definitioner til CUDA platformen
vec_math.h	Vektor matematik funktioner og almindelige matematiske funktioner til CUDA platformen

include/i386

I lighed med de to andre platforme findes der i dette bibliotek includes til i386 platformen. Disse includes er de samme som for de andre platforme, men disse er de mest generiske.

Fil	Notes
config.h	Diverse indstillinger og defines til I386 platformen
geometry.h	Geometriske definitioner til I386 platformen
matrix_math.h	Matrix operationer og homogene transformations operationer til I386 platformen
opp_types.h	Diverse type definitioner til I386 platformen
vec_math.h	Vektor matematik funktioner og almindelige matematiske funktioner til I386 platformen

libopp

Kernen i projektet ligger i biblioteksstrukturen under dette bibliotek. I selve biblioteket ligger interfacet til LibOPP library'et, med diverse definitioner og funktions prototyper.

Fil	Notes
libopp.h	Public interface til LibOPP biblioteket
libopp_errno.h	Definition af fejlkoder brugt af LibOPP biblioteket

libopp/cell

Dette bibliotek og dets underbiblioteker indeholder CELL implementationen af LibOPP. Selve implementationen er beskrevet andetsteds i rapporten.

Fil	Notes
alf_sys.c	ALF implementation af exhaustive OBB og trekant tests
alf_sys.h	Header til alf_sys.c
cell_timing.c	Funktioner til timings formål på CELL platformen
cell_timing.h	Header til cell_timing.c
libopp_ppu.c	CELL implementation af LibOPP interfacet
ppu_mem.c	Diverse funktioner til at håndtere hukommelsen på CELL platformen
ppu_mem.h	Header til ppu_mem.c
spu_pgm.c	Hjælpe funktioner til model tests
spu_pgm.h	Header til spu_pgm.c

libopp/cell/include

Filer der skal deles imellem PPU og SPU siden af LibOPP CELL implementationen ligger i dette bibliotek.

Fil	Notes
obb_parm.h	Parametre der skal være fælles for PPU og SPU'ern ved model tests
wb_parm.h	Parametre der skal være fælles for PPU og SPU'ern ved exhaustive tests

libopp/cell/spu_model_test

Dette bibliotek indeholder SPU koden til model testen.

Fil	Notes
spu_model_test.c	SPU implementation af model tests

libopp/cell/spu_primitive_test

De udtømmende trekant og OBB tests er implementeret i SPU koden i dette bibliotek.

Fil	Notes
spu_primitive_test.c	SPU implementation af exhaustive primitiv tests

libopp/cuda

Denne biblioteksstruktur indeholder CUDA implementationen af LibOPP.

Fil	Notes
dbg_cuda.cu	Hjælpe kode til at debugge CUDA kerner
libopp_cuda.cu	CUDA implementation af LibOPP
libopp_cuda.h	Header til libopp_cuda.cu
util.cu	Diverse utility funktioner til CUDA

libopp/cuda/cuda_model_test

Model testen i CUDA udgave ligger i dette bibliotek. Dette bibliotek indeholder både device (GPU) og host (CPU) siden af koden.

Fil	Notes
cuda_model_kernels.cu	CUDA kerne til model tests
cuda_model_test.cu	Host side CUDA kode til model tests
cuda_model_test.h	Header til cuda_model_test.cu

libopp/cuda/cuda_primitive_test

CUDA implementationen af trekant og OBB testene er indeholdt i dette bibliotek. Som med model testen, ligger både host side og device side kode i biblioteket.

Fil	Notes
cuda_primitive_test.cu	Host side CUDA kode til exhaustive primitiv tests
cuda_primitive_test.h	Header til cuda_primitive_test.cu
obbovb_test_kernel.cu	CUDA kerne til exhaustive tests af OBB's
tritri_test_kernel.cu	CUDA kerne til exhaustive tests af trekanter

libopp/cuda/include

Dette bibliotek indeholder includes der skal benyttes af både host side og device side i CUDA.

Fil	Notes
debug_cuda.h	Header til device side debug på CUDA platformen

libopp/i386

Den sidste inplementation af LibOPP library'et er til I386 platformen, og findes i denne bibliotekstruktur. Denne implementation er medtaget primært som reference implementation. Men den benyttes også som udgangspunkt for hastighedstest af de andre implementationer af LibOPP.

Fil	Notes
libopp_i386.c	Implmentation af LibOPP på I386 platformen

util

Dette bibliotek indeholder en del forskellig funktionalitet. Her findes bla. debug funktionalitet, diverse output funktioner, implementation af en række standard datastrukturer og meget mere. Alt i alt i masse utility kode der er nødvendig for at det hele til at hænge sammen.

Fil	Notes
custom_printf.c	Diverse udvidelser til printf funktionen, så den kan udskrive custom typer
custom_printf.h	Header til custom_printf.c
debug.c	Funktioner til at udskrive debug meddelelser
debug.h	Header til debug.c
emem.c	Hjælpe funktioner til memory allocation
emem.h	Header til emem.c
file_util.c	Funktioner til at indlæse diverse data fra filer
file_util.h	Header til file_util.c
geomview.c	Funktioner til at visualisere primitiver med programmet geomview
geomview.h	Header til geomview.c
graphviz.c	Funktioner til at visualisere træer med programmet graphviz
graphviz.h	Header til graphviz.c
list.c	Implementation af doubly linked og singly linked lister
list.h	Header til list.c
queue.c	Implementation af en kø
queue.h	Header til queue.c
random.c	Diverse funktioner til at generer f.eks. tilfældige trekanter
random.h	Header til random.c
result.c	Funktioner til at udskrive og håndtere resultater fra exhaustive primitiv tests
result.h	Header til result.c
rotation.c	Funktioner til at udføre rotationer og translationer
rotation.h	Header til rotation.c
stack.c	Implementation af en stak
stack.h	Header til stack.c
timing.c	Funktioner til at udføre tidsmålinger
timing.h	Header til timing.c
tree.c	Implementation af et generelt træ
tree.h	Header til tree.c
vec3d_triple.c	Implementation af trippel skalar produkt
vec3d_triple.h	Header til vec3d_triple.c

8.3 LibOPP API

Som tidligere nævnt er LibOPP den centrale komponent i koden. Dette library består basalt set af et API og en række implementationer af dette til de forskellige platforme. Selve API'et delt op i to dele, som hver især håndtere hhv. udtømmende tests af primitiver (trekanter og OBB'er) og tests af hele modeller. I afsnit 8.3.2 gennemgås de funktioner og datastrukturer der knytter sig til de udtømmende tests, mens afsnit 8.3.3 beskriver API'et der omhandler model tests.

8.3.1 Generelt API

Det generelle API benyttes til at initialisere LibOPP og til at nedlægge LibOPP igen. Da API'et som nævnt kan udføre to forskellige former for tests, skal LibOPP initialiseres til at udføre den ønskede form for tests. Herunder er de generelle API funktioner dokumenteret.

libopp_init*C-prototype:*

```
int libopp_init(enum LIBOPP_MODE mode);
```

Parametere: mode: angiver hvilken mode LibOPP skal bruges i.

Kan være MODE_EXHAUSTIVE, eller MODE_MODEL

Retur værdi: LIBOPP_ERR hvis kaldet fejler, ellers LIBOPP_OK

Beskrivelse: Initialisere LibOPP til enten at blive brugt til udtømmende primitiv tests, eller model test

Noter: Denne funktions skal kaldes før LibOPP kan benyttes.

libopp_finalize*C-prototype:*

```
int libopp_finalize(void);
```

Parametere: Ingen

Retur værdi: LIBOPP_ERR hvis kaldet fejler, ellers LIBOPP_OK

Beskrivelse: Afslutter LibOPP og frigiver eventuelle ressourcer

Noter: Efter kald til denne funktion, skal LibOPP initialiseres igen før det kan bruges

8.3.2 Udtømmende primitiv test API

Denne del af LibOPP API'et benyttes til at udføre udtømmende test af enten trekanter eller OBB'er. En udtømmende test består af to supper bestående af enten trekanter eller OBB'er. Hvert element i den ene suppe testes mod alle andre elementer i den anden suppe, hvilket resulterer i et resultat matrix, hvor et "1" indikere en kollision imellem de to tilsvarende elementer i supperne. Mens et "0" i kollisions matrixen indikere at der ikke er kollision imellem de to elementer.

Kollisions resultatet er repræsenteret ved en struct, som det ses herunder:

```
1  /** Struct representing a result from a exhaustive test */
2  struct result_matrix {
3      opp_result_t *r; /**< Result matrix */
4      int rows; /**< Number of columns in result matrix */
5      int cols; /**< Number of rows in result matrix */
6  };
```

Som sagt testes kollisionen mellem to supper, enten en trekant suppe defineret som:

```
1  /** Struct representing a triangle soup */
2  struct triangle_soup {
3      triangle_t *t; /**< array of triangle in soup */
4      int n_tri; /**< Number of triangles in soup */
5  };
```

Eller en OBB suppe defineret som:

```
1  struct obb_soup {
2      obb_t *o; /**< array of OBB's in the soup */
3      int n_obb; /**< number of OBB's in the soup */
4  };
```

En test består af to supper og den tilhørende resultat matrice. For en trekant test er dette repræsenteret således:

```

1  /** Struct representing a exhaustive triangle test */
2  struct triangle_test {
3      struct triangle_soup *ts1; /**< First triangle soup */
4      struct triangle_soup *ts2; /**< Second triangle soup */
5      struct result_matrix r; /**< Result of the test */
6      void *p_data; /**< Placeholder for private data */
7  };

```

På tilsvarende vis er en OBB test repræsenteret som følger:

```

1  /** Struct representing a exhaustive OBB test */
2  struct obb_test {
3      struct obb_soup *os1; /**< First OBB soup */
4      struct obb_soup *os2; /**< Second OBB soup */
5      struct result_matrix r; /**< Result of the test */
6      void *p_data; /**< Placeholder for private data */
7  };

```

Disse strukturer udgør kernen i de udtømmende tests. Herunder er de funktioner der operer på disse strukturer beskrevet nærmere.

libopp_create_triangle_soup

C-prototype:

```
int libopp_create_triangle_soup(struct triangle_soup **ts, int n);
```

Parametere: ts: Pointer til pointer til triangle_soup struktur
n: Antal trekanter der skal være plads til i suppen

Retur værdi: LIBOPP_ERR hvis kaldet fejler, ellers LIBOPP_OK

Beskrivelse: Opretter en trekant suppe og allokerer plads til n trekanter i denne

Noter: Denne funktion er nødvendig idet visse platforme har specielle krav til f.eks. alignment af den hukommelse der benyttes

libopp_free_triangle_soup

C-prototype:

```
int libopp_free_triangle_soup(struct triangle_soup **ts);
```

Parametere: ts: Pointer til pointer til triangle_soup struktur

Retur værdi: LIBOPP_ERR hvis kaldet fejler, ellers LIBOPP_OK

Beskrivelse: Nedlægger trekant suppen og frigiver hukommelsen

Noter: Efter kaldet er *ts en NULL pointer

libopp_create_triangle_test

C-prototype:

```
int libopp_create_triangle_test(struct triangle_soup *ts1, struct triangle_soup *ts2, struct triangle_test **tt);
```

Parametere: ts1: Pointer til den første trekant suppe i testen
 ts2: Pointer til den anden trekant suppe i testen
 tt: Pointer til pointer til resulterede triangle_test struktur

Retur værdi: LIBOPP_ERR hvis kaldet fejler, ellers LIBOPP_OK

Beskrivelse: Opretter en udtømmende trekant test, bestående af to trekant supper, der skal testes mod hinanden

Noter: Denne funktion allokerer ligeledes plads til det resultat matrix, hvor test resultatet placeres når testen udføres

libopp_free_triangle_test

C-prototype:

```
int libopp_free_triangle_test(struct triangle_test **tt);
```

Parametere: tt: Pointer til pointer til triangle_test struktur

Retur værdi: LIBOPP_ERR hvis kaldet fejler, ellers LIBOPP_OK

Beskrivelse: Nedlægger en trekant test, og frigiver hukommelsen associeret hermed

Noter: Efter kaldet er *tt en NULL pointer. Kaldet frigiver ikke de to trekant supper.

libopp_tri_tri_test

C-prototype:

```
int libopp_tri_tri_test(struct triangle_test *tt);
```

Parametere: tt: Pointer til triangle_test struktur

Retur værdi: LIBOPP_ERR hvis kaldet fejler, ellers LIBOPP_OK

Beskrivelse: Udfører en udtømmende trekant test og placerer resultatet i resultat matrixen

Noter: Ingen

libopp_create_obb_soup

C-prototype:

```
int libopp_create_obb_soup(struct obb_soup **os, int n);
```

Parametere: os: Pointer til pointer til obb_soup struktur
 n: Antal OBB'er der skal være plads til i suppen

Retur værdi: LIBOPP_ERR hvis kaldet fejler, ellers LIBOPP_OK

Beskrivelse: Opretter en OBB suppe og allokerer plads til n OBB'er i denne

Noter: Denne funktion er nødvendig idet visse platforme har specielle krav til f.eks. alignment af den hukommelse der benyttes

libopp_free_obb_soup

C-prototype:

```
int libopp_free_obb_soup(struct obb_soup **os);
```

Parametere: os: Pointer til pointer til obb_soup struktur
Retur værdi: LIBOPP_ERR hvis kaldet fejler, ellers LIBOPP_OK
Beskrivelse: Nedlægger OBB suppen og frigiver hukommelsen
Noter: Efter kaldet er *os en NULL pointer

libopp_create_obb_test

C-prototype:

```
int libopp_create_obb_test(struct obb_soup *os1, struct obb_soup *os2, struct obb_test **ot);
```

Parametere: os1: Pointer til den første OBB suppe i testen
os2: Pointer til den anden OBB suppe i testen
ot: Pointer til pointer til resulterede obb_test struktur
Retur værdi: LIBOPP_ERR hvis kaldet fejler, ellers LIBOPP_OK
Beskrivelse: Opretter en udtømmende OBB test, bestående af to OBB supper, der skal testes mod hinanden
Noter: Denne funktion allokerer ligeledes plads til det resultat matrix, hvor test resultatet placeres når testen udføres

libopp_free_obb_test

C-prototype:

```
int libopp_free_obb_test(struct obb_test **ot);
```

Parametere: ot: Pointer til pointer til obb_test struktur
Retur værdi: LIBOPP_ERR hvis kaldet fejler, ellers LIBOPP_OK
Beskrivelse: Nedlægger en OBB test, og frigiver hukommelsen associeret hermed
Noter: Efter kaldet er *ot en NULL pointer. Kaldet frigiver ikke de to OBB supper.

libopp_obb_obb_test

C-prototype:

```
int libopp_obb_obb_test(struct obb_test *ot);
```

Parametere: ot: Pointer til obb_test struktur
Retur værdi: LIBOPP_ERR hvis kaldet fejler, ellers LIBOPP_OK
Beskrivelse: Udfører en udtømmende OBB test og placerer resultatet i resultat matrixen
Noter: Ingen

8.3.3 Model test API

Model test API delen af LibOPP API'et er i stand til at udføre kollisions test mellem en række objekter i en scene. Hvert objekt er opbygget som er træ af OBB'er og de tilhørende trekanten. Et træ er defineret som det ses herunder:

```
1 /** A single rigid body is representet by a object tree */
2 struct object_tree {
3     it_node_t *it; /**< implicit representation of the tree */
```

```

4  int it_nnodes; /**< Number of implicit tree nodes */
5  struct obb_soup *os; /**< Soup of OBB's covering the triangles in the object */
6  struct triangle_soup *ts; /**< Soup of triangle that make up the object */
7  };

```

Internt benyttes såkaldte Collision Test Tree nodes (ctt_nodes) til at tests for kollision, og det endelige resultat er også repræsenteret ved sådanne noder. En ctt_node beskriver en kollision ved at angive hvilke to træer (objekter i scenen) der er i kollision, samt hvilke trekanter i de to træer der specifikt er i kollision. En ctt_node er defineret som:

```

1  /** Collision Test Tree node. A central datastructure in the
2  * design. This structure represents a single test in the imaginary
3  * Collision Test Tree. This tree represents all possible collisions
4  * bwteen the objects in the scene */
5  struct ctt_node {
6  unsigned short objt1; /**< tree 1, index into array of object trees */
7  unsigned short objt2; /**< tree 2, index into array of object trees */
8  unsigned short n1; /**< node of tree 1, index into it_node_t array */
9  unsigned short n2; /**< node of tree 2, index into it_node_t array */
10 unsigned short obj1_idx; /**< the optionally resolved index of (tri/obb) object
11     1 */
12 unsigned short obj2_idx; /**< the optionally resolved index of (tri/obb) object
13     2 */
14 };

```

Selve scenen består af en række træer og deres tilhørende homogene transformationer. Desuden indeholder scenen også information om hvilke objekter der skal testes mod hinanden. Scenen er defineret som:

```

1  /** Struct describing an entire scene, consisting of a number of
2  * individual objects */
3  struct scene {
4  struct object_tree *objts; /**< array of object trees */
5  struct transformation_matrix *tfs; /**< array of transformations, one
6  transformation for each object tree */
7  int n_objts; /**< Number of objects in the scene */
8  void *p_data; /**< Pointer to private data */
9  void *arch_data; /**< Pointer to architecture specific data */
10 struct ctt_node tps[TREPAIRS_MAX]; /**< array of tree pairs to be tested for
11 collision */
12 int n_tps; /**< number of tree pairs in the array */
13 };

```

Efter denne gennemgang af de centrale datastrukturer i model API'et følger en gennemgang af funktionerne i API'et.

libopp_create_scene

C-prototype:

```
int libopp_create_scene(struct scene **scene, int n_objts);
```

Parametere: scene: Pointer to pointer til scene struktur
 n_objts: Antal af objekt træer scenen skal kunne indeholde
Retur værdi: LIBOPP_ERR hvis kaldet fejler, ellers LIBOPP_OK
Beskrivelse: Opretter en ny scene med plads til n_objts objekter
Noter: Ingen

libopp_free_scene

C-prototype:

```
int libopp_free_scene(struct scene **scene);
```

Parametere: scene: Pointer til pointer til scene struktur
Retur værdi: LIBOPP_ERR hvis kaldet fejler, ellers LIBOPP_OK
Beskrivelse: Nedlægger en scene og frigiver hukommelsen associeret hermed
Noter: Efter kaldet er *scene en NULL pointer

libopp_begin_model

C-prototype:

```
int libopp_begin_model(struct scene *scene);
```

Parametere: scene: Pointer til scene struktur
Retur værdi: LIBOPP_ERR hvis kaldet fejler, ellers LIBOPP_OK
Beskrivelse: Påbegynder opbygningen af en model for den pågældende scene
Noter: Denne funktion skal kaldes før der kan tilføjes objekter til scenen

libopp_end_model

C-prototype:

```
int libopp_end_model(struct scene *scene);
```

Parametere: scene: Pointer til scene struktur
Retur værdi: LIBOPP_ERR hvis kaldet fejler, ellers LIBOPP_OK
Beskrivelse: Færdiggøre opbygningen af en model
Noter: Efter kald til denne funktion kan der ikke længere tilføjes objekter til scenen

libopp_update_model

C-prototype:

```
int libopp_update_model(struct scene *scene);
```

Parametere: scene: Pointer til scene struktur
Retur værdi: LIBOPP_ERR hvis kaldet fejler, ellers LIBOPP_OK
Beskrivelse: Opdatere scenen med nye transformations matricer
Noter: Ingen

libopp_add_treepair

C-prototype:

```
int libopp_add_treepair(struct scene *scene, int objt1, int objt2);
```

Parametere: scene: Pointer til scene struktur
 obj1: Index for det første objekt træ
 obj2: Index for det andet objekt træ

Retur værdi: LIBOPP_ERR hvis kaldet fejler, ellers LIBOPP_OK

Beskrivelse: Tilføjer et træ-par til scenen. Dette betyder at der kontrolleres for kollision mellem de to objekter

Noter: Ingen

libopp_do_col_check

C-prototype:

```
int libopp_do_col_check(struct scene *scene);
```

Parametere: scene: Pointer til scene struktur

Retur værdi: LIBOPP_ERR hvis kaldet fejler, ellers antallet af kollisioner i scenen

Beskrivelse: Udføre en kollision test på en scene.

Noter: Scenen skal først opbygges og træ-par skal tilføjes før denne funktion kaldes

libopp_get_result

C-prototype:

```
int libopp_get_result(struct ctt_node **nodes);
```

Parametere: nodes: Pointer til pointer til de rå ctt_nodes strukturer

Retur værdi: LIBOPP_ERR hvis kaldet fejler, ellers LIBOPP_OK

Beskrivelse: Returnere et array af ctt_node strukturerer, der hver især beskriver en enkelt kollision i scenen

Noter: Ingen

8.3.4 Kodeeksempler

Som en afslutning på gennemgangen af LibOPP API'et følger her et par kode eksempler på dets brug. Det første eksempel indlæser to trekant supper og tester dem for kollision. Slutteligt skrives resultatet ud. Bemærk at er ikke fortages fejl check for overskuelighedens skyld.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "geometry.h"
4 #include "libopp.h"
5 #include "file_util.h"
6 #include "result.h"
7
```



```

8 int main(int argc, char **argv) {
9
10     struct triangle_soup *ts1 = NULL;
11     struct triangle_soup *ts2 = NULL;
12     struct triangle_test *tt = NULL;
13     int o1s, o2s;
14
15     /* Initialiser LibOPP */
16     libopp_init(MODE_EXHAUSTIVE);
17
18     o1s = 16; /* Antal trekkanter i objekt 1 */
19     o2s = 16; /* Antal trekkanter i objekt 2 */
20
21     /* Opret trekant supper */
22     libopp_create_triangle_soup(&ts1, o1s);
23     libopp_create_triangle_soup(&ts2, o2s);
24
25     /* Hent trekkanter fra fil */
26     load_object_triangle("testdata/object0.txt", ts1->t, o1s);
27     load_object_triangle("testdata/object1.txt", ts2->t, o2s);
28
29     /* Opret trekant test */
30     libopp_create_triangle_test(ts1, ts2, &tt);
31
32     /* Eksekver trekant testen */
33     libopp_tri_tri_test(tt);
34
35     /* Udskriv resultater */
36     result_print(tt->r.r, tt->r.cols, tt->r.rows);
37
38     /* Ryd op */
39     libopp_free_triangle_test(&tt);
40     libopp_free_triangle_soup(&ts1);
41     libopp_free_triangle_soup(&ts2);
42     libopp_finalize();
43
44     return EXIT_SUCCESS;
45 }

```

Et kodeeksempel for tests mellem to OBB supper er praktisk talt identisk med ovenstående og vil derfor ikke blive vist.

At teste en scene for kollision er lidt mere indviklet end at teste primitiver. Herunder er et eksempel på hvordan dette gøres. Først oprettes scenen og objekterne læses ind i den fra disken. Dernæst placeres objekterne i scenen ved at angive deres homogene transformationer. Dernæst specificeres det hvilke objekter der skal kontrolleres for kollision. Og endeligt udføres selve testen, hvorefter scenen og objekterne nedlægges igen. Bemærk at resultaterne ikke benyttes i dette eksempel, her gemmes antallet af kollisioner i scenen blot. Bemærk yderligere at det er muligt at flytte objekterne ved at ændre deres homogene transformationer. Herved kan en serie af kollisions test udføres, hvor objekterne flyttes mellem hver test.

```

1 #include <stdlib.h>
2 #include "file_util.h"

```

```

3 #include "matrix_math.h"
4 #include "libopp.h"
5 #include "rotation.h"
6
7 int main(int argc, char **argv) {
8
9     struct scene *scene;
10    int nres;
11
12    /* Initialiser LibOPP */
13    libopp_init(MODE_MODEL);
14
15    /* Opret scene med plads til 11 objekter */
16    libopp_create_scene(&scene, 11);
17
18    /* Start med at opbygge scene */
19    libopp_begin_model(scene);
20    /* Hent objekter fra filer */
21    load_object_tree("testdata/base_3615_tree.txt", "testdata/base_3615_obbs.txt"
22        , "testdata/base_3615_tris.txt", scene->objts + 0);
23    load_object_tree("testdata/fingerbase_3953_tree.txt", "testdata/
24        fingerbase_3953_obbs.txt", "testdata/fingerbase_3953_tris.txt", scene->
25        objts + 1);
26    load_object_tree("testdata/fingermid_3902_tree.txt", "testdata/
27        fingermid_3902_obbs.txt", "testdata/fingermid_3902_tris.txt", scene->
28        objts + 2);
29    load_object_tree("testdata/fingertip_3806_tree.txt", "testdata/
30        fingertip_3806_obbs.txt", "testdata/fingertip_3806_tris.txt", scene->
31        objts + 3);
32    load_object_tree("testdata/fingerbase_3953_tree.txt", "testdata/
33        fingerbase_3953_obbs.txt", "testdata/fingerbase_3953_tris.txt", scene->
34        objts + 4);
35    load_object_tree("testdata/fingermid_3902_tree.txt", "testdata/
36        fingermid_3902_obbs.txt", "testdata/fingermid_3902_tris.txt", scene->
37        objts + 5);
38    load_object_tree("testdata/fingertip_3806_tree.txt", "testdata/
39        fingertip_3806_obbs.txt", "testdata/fingertip_3806_tris.txt", scene->
40        objts + 6);
41    load_object_tree("testdata/fingerbase_3953_tree.txt", "testdata/
42        fingerbase_3953_obbs.txt", "testdata/fingerbase_3953_tris.txt", scene->
43        objts + 7);
44    load_object_tree("testdata/fingermid_3902_tree.txt", "testdata/
45        fingermid_3902_obbs.txt", "testdata/fingermid_3902_tris.txt", scene->
46        objts + 8);
47    load_object_tree("testdata/fingertip_3806_tree.txt", "testdata/
48        fingertip_3806_obbs.txt", "testdata/fingertip_3806_tris.txt", scene->
49        objts + 9);
50    load_object_tree("testdata/ship_tree.txt", "testdata/ship_obbs.txt", "
51        testdata/ship_tris.txt", scene->objts + 10);
52
53    /* Det foelgende kode opbygger homogene transformationer, der placere
54       objekterne i scenen */
55    /* BASE */
56    rotation_matrix_identity(scene->tfs[0].rot);
57    scene->tfs[0].trans[0] = 0.0;
58    scene->tfs[0].trans[1] = 0.0;
59    scene->tfs[0].trans[2] = 0.0;

```

```
39
40     /* finger 1 */
41     rotation_matrix_r3(scene->tfs[1].rot, 0);
42     scene->tfs[1].trans[0] = 0.0;
43     scene->tfs[1].trans[1] = 0.0;
44     scene->tfs[1].trans[2] = 0.0;
45
46     rotation_matrix_r3(scene->tfs[2].rot, 0);
47     scene->tfs[2].trans[0] = 0.0;
48     scene->tfs[2].trans[1] = 0.0;
49     scene->tfs[2].trans[2] = 0.0;
50
51     rotation_matrix_r3(scene->tfs[3].rot, 0);
52     scene->tfs[3].trans[0] = 0.0;
53     scene->tfs[3].trans[1] = 0.0;
54     scene->tfs[3].trans[2] = 0.0;
55
56     /* finger 2 */
57     rotation_matrix_r3(scene->tfs[4].rot, 2.094);
58     scene->tfs[4].trans[0] = 0.0;
59     scene->tfs[4].trans[1] = 0.0;
60     scene->tfs[4].trans[2] = 0.0;
61
62     rotation_matrix_r3(scene->tfs[5].rot, 2.094);
63     scene->tfs[5].trans[0] = 0.0;
64     scene->tfs[5].trans[1] = 0.0;
65     scene->tfs[5].trans[2] = 0.0;
66
67     rotation_matrix_r3(scene->tfs[6].rot, 2.094);
68     scene->tfs[6].trans[0] = 0.0;
69     scene->tfs[6].trans[1] = 0.0;
70     scene->tfs[6].trans[2] = 0.0;
71
72     /* finger 3 */
73     rotation_matrix_r3(scene->tfs[7].rot, -2.094);
74     scene->tfs[7].trans[0] = 0.0;
75     scene->tfs[7].trans[1] = 0.0;
76     scene->tfs[7].trans[2] = 0.0;
77
78     rotation_matrix_r3(scene->tfs[8].rot, -2.094);
79     scene->tfs[8].trans[0] = 0.0;
80     scene->tfs[8].trans[1] = 0.0;
81     scene->tfs[8].trans[2] = 0.0;
82
83     rotation_matrix_r3(scene->tfs[9].rot, -2.094);
84     scene->tfs[9].trans[0] = 0.0;
85     scene->tfs[9].trans[1] = 0.0;
86     scene->tfs[9].trans[2] = 0.0;
87
88     /* grabbing object */
89     rotation_matrix_r3(scene->tfs[10].rot, -1.4999);
90     scene->tfs[10].trans[0] = 0.0;
91     scene->tfs[10].trans[1] = 0.0;
92     scene->tfs[10].trans[2] = -0.015;
93
94     /* Scenen er opbygget */
95     libopp_end_model(scene);
```

```

96  /* Opdater transformationerne */
97  libopp_update_model(scene);
98
99  /* Definer mellem hvilke objekter der skal kontrolleres for kollision */
100 /* Indekserne refererer til den rækkefølge som objekterne blev indlaest i
    */
101 libopp_add_treepair(scene, 3, 6);
102 libopp_add_treepair(scene, 3, 9);
103 libopp_add_treepair(scene, 6, 9);
104 libopp_add_treepair(scene, 0, 10);
105 libopp_add_treepair(scene, 1, 10);
106 libopp_add_treepair(scene, 2, 10);
107 libopp_add_treepair(scene, 3, 10);
108 libopp_add_treepair(scene, 4, 10);
109 libopp_add_treepair(scene, 5, 10);
110 libopp_add_treepair(scene, 6, 10);
111 libopp_add_treepair(scene, 7, 10);
112 libopp_add_treepair(scene, 8, 10);
113 libopp_add_treepair(scene, 9, 10);
114
115 /* Udfoer kollisions testen */
116 nres = libopp_do_col_check(scene);
117
118 /* Ryd op */
119 libopp_free_obb_soup(&(scene->objts[0].os));
120 libopp_free_obb_soup(&(scene->objts[1].os));
121 libopp_free_triangle_soup(&(scene->objts[0].ts));
122 libopp_free_triangle_soup(&(scene->objts[1].ts));
123 free((scene->objts[0].it));
124 free((scene->objts[1].it));
125 libopp_free_scene(&scene);
126 libopp_finalize();
127
128 return EXIT_SUCCESS;
129 }

```

8.4 Build environment

Eftersom udviklingen er forgået på flere forskellige platforme, med flere forskellige platforme som target for koden, har der været brugt en del energi på at sætte et fornuftigt build environment op. De primære udviklings maskiner har været baseret på standard PC'er med Gentoo Linux (en 32 bit og en 64 bit). Derudover er CUDA target maskinen også en standard PC med 64 bit Gentoo Linux. Mens Target for CELL udviklingen har været en PlayStation 3 med Fedora Core 7 linux i 64 bit udgave. (Se appendix B og C for installationsvejledninger.) Yderligere har IBM system simulator også været brugt som target for CELL.

8.4.1 Værktøjer

På I386 platformen har de primære værktøjer der er blevet brugt været standard Linux værktøjer. Disse omfatter: GNU GCC version 4.1.2, binutils version 2.18 og GNU libc version 2.6.1. Derudover er debug værktøjer blevet brugt: valgrind version 3.3.1, GNU gdb version 6.7.1 og GNU DDD version 3.3.11.

På CELL platformen er der ligeledes blevet anvendt standard Linux værktøjer. Disse er alle blevet installeret som en del af IBM SDK for Multicore Acceleration Version 3.0, hvorfor de enkelte værktøjer ikke vil blive opremset her. Blot skal det nævnes at GCC og ikke IBM egen compiler er blevet anvendt.

På CUDA platformen er der blevet anvendt de samme værktøjer som på I386 platformen, samt CUDA Toolkit version 2.0 BETA. Der er desuden brugt kode fra CUDA SDK version 2.0 BETA. Video driveren der er anvendt er NVIDIA Accelerated Graphics Driver for Linux-x86_64 version 177.67. Desuden er der anvendt CUDA visual profiler version 2.0.

8.4.2 Makefiler

Til at bygge koden på en ensartet måde, benyttes standard GNU make. Til denne brug er der blevet udviklet en række Makefiler, samt et shell script. Dette script (tools/settarget.sh) benyttes til at vælge til hvilken platform der skal kompileres når make kaldes. Scriptet skifter ganske enkelt mellem de forskellige platformes makefiler med et symbolsk link. De forskellige filer der udgør det samlede make system for projektet er beskrevet herunder:

Fil	Placering	Noter
Makefile	./	Symbolsk link til platform specifik make fil
Makefile.cell	./	Hoved make fil til CELL platformen
Makefile.cuda	./	Hoved make fil til CUDA platformen
Makefile.i386	./	Hoved make fil til I386 platformen
objs.mk	./	Fælles definition for alle platforme af objekt filer
common.mk	./	Fælles definitioner for alle platforme
common_tg.mk	./	Fælles targets for alle platforme
settarget.sh	./tools	Shellscript til at specificere target platform
Makefile	./glut_viewer	Selvstændig makefil til glut_viewer applikationen
Makefile	./libopp/cell	Hoved Makefile til CELL udgaven af LibOPP
cell.mk	./libopp/cell	CELL specifikke defines og makroer
Makefile	./libopp/cell/spu_model_test	Makefile til at bygge SPU kode til model test
Makefile	./libopp/cell/spu_primitive_test	Makefile til at bygge SPU koden til primitiv test
Makefile	./libopp/cuda	Hoved Makefile til CUDA udgaven af LibOPP
cuda.mk	./libopp/cuda	CUDA specifikke defines og makroer
Makefile	./libopp/i386	Hoved Makefile til I386 udgaven af LibOPP
i386.mk	./libopp/i386	I386 specifikke defines og makroer

CELL make filen har et target til at installere de binære filer via ssh på en PlayStation 3 fra udviklings maskinen. For at benytte denne feature skal man enten søgere for at hostnavnet ps3 resolver til PlayStationens IP-adresse (f.eks. via /etc/hosts filen). Eller også kan IP-adressen indsættes di-

rekte i `cell.mk` filen. Dernæst skal PlayStationen sætte op til at acceptere passwordless login ² på den ønskede brugerkonto (som ligeledes sættes i filen `cell.mk`).

8.4.3 Defines og indstillinger

Koden indeholder en del defines til at kontrollere en mængde forskellige indstillinger. Mange af disse indstillinger er specifikke for en lille del af kode, og disse vil blive behandlet i de forskellige implementations afsnit hvor de har relevans. Der er også en række globale indstillinger som er beskrevet i tabellen herunder:

Define	fil	Noter
<code>_ENABLE_DEBUG</code>	<code>common_conf.h</code>	Slå debug output til/fra
<code>_STORAGE_TYPE_EXTENDED_PRECISION</code>	<code>common_conf.h</code>	Benyt extended (80 bit float) præcision
<code>_STORAGE_TYPE_DOUBLE_PRECISION</code>	<code>common_conf.h</code>	Benyt double (64 bit float) præcision
<code>_STORAGE_TYPE_SINGLE_PRECISION</code>	<code>common_conf.h</code>	Benyt double (32 bit float) præcision
<code>_TRI_ALG_DEVILLERS</code>	<code>common_conf.h</code>	Anvend devillers algoritmen til trekant tests
<code>_TRI_ALG_DEVILLERS_ORIG</code>	<code>common_conf.h</code>	Anvend den originale devillers algoritme til trekant tests
<code>_TRI_ALG_SEGMENT_PIERCING</code>	<code>common_conf.h</code>	Anvend segment piercing algoritmen til trekant tests
<code>_OBB_ALG_GOTTSCHALK</code>	<code>common_conf.h</code>	Anvend Gottschalk algoritmen til OBB tests
<code>_OBB_ALG_EXHAUSTIVE</code>	<code>common_conf.h</code>	Anvend Exhaustive algoritmen til OBB tests
<code>_DEVILLERS_USE_TOLERANCES</code>	<code>common_conf.h</code>	Benyt tolerancer i devillers algoritmen
<code>_CELL_USE_VECTOR</code>	<code>cell/config.h</code>	Benyt vektor (SIMD) på CELL platformen
<code>_CUDA_RUNTIME_EMULATION</code>	<code>cuda/config.h</code>	Emulerer GPU'en på CUDA platformen

Debug indstillingerne kan ud over den ovenfor nævnet define, desuden finindstilles yderligere via variabelen `debug_modules` i files `util/debug.c`. Denne variabel bestemmer hvilke moduler der kan udskrive debug output. Modulerne er defineret i filen `util/debug.h`.

8.5 Delkonklusion

Udviklingen af et fælles API og library for alle platformene har vist sig at være en stor udfordring. Ikke mindst at skulle vedligeholde en kodebase til tre forskellige platforme har vist sig at være mere

²Se f.eks. <http://www.debian-administration.org/articles/152> for en guide til dette.

besværligt end først antaget. Faktisk i et sådant omfang at hvis det skulle gøre om, ville resultatet nok blive separate kodebaser til hver platform. Den nuværende organisering af koden kunne endvidere godt trænge til en revision. Eksempelvis benytter både LibOPP og dele af test koden de samme debug funktioner. Det kunne med fordel overvejes at lave en mere skarp opdeling mellem LibOPP og resten af koden.

Kapitel 9

Algoritme implementation

Dette kapitel beskæftiger sig med selve implementationen af de anvendte algoritmer. Selve algoritmerne er beskrevet i teori afsnittet af rapporten. Desuden vil det også blive beskrevet hvorledes implementationen af massive tests af hhv. trekanter og OBB'er er fortaget på de forskellige platforme.

9.1 Trekant-trekant tests

9.1.1 Generelt

Kernen i alle trekant algoritmerne er en datatstruktur der beskriver en trekant. Herunder er vist hvordan denne datastruktur ser ud i sin grundform.

```
1 typedef struct triangle_s
2 {
3     point3d a;
4     point3d b;
5     point3d c;
6 #ifdef _ENABLE_DEBUG
7     unsigned int signature;
8 #endif
9 } triangle_t;
```

Hver hjørne er angivet som et `point3d`. Hvordan et 3D punkt er defineret er forskelligt fra platform til platform. På CUDA og I386 platformene er det defineret som et array af tre flydende kommatall (hvis præcision afhænger af en global indstilling). På CELL platformen derimod består et 3D punkt af en vektor på 4 flydende kommatall, idet dette giver mulighed for at benytte denne platforms SIMD egenskaber. Den angivne `signature` variabel benyttes ved debug til at sikre at en trekant ikke bliver korrupt, og hvis den gør så opstår der en fejlsituation ¹.

De forskellige trekant test algoritmer er implementeret med de samme interface, så det er muligt at skifte mellem dem. Interfacet er udformet så et kald til en trekanter test algoritme udføres netop en test imellem to trekanter. Det antages at de to trekanter er i samme reference ramme (angivet i samme koordinat frame).

Alle grundlæggende matematiske funktioner som eksempelvis vektor subtraktion, prik produktet og kryds produkter, er implementeret som makroer. Dette muliggør at disse operationer kan optimeres til

¹Normalt indeholde signaturen værdien `0xDEADBEEF`, og denne værdi kontrolleres med mellemrum, og hvis værdien ikke længere passer udskrives en fejlbesked.

de enkelte platforme, for at give maksimal ydelse. Denne fremgangsmåde gør også at alle algoritmer kan skrives generelt, og således ikke skal implementeres specielt til den enkelte platform. Til gengæld bliver det sværere at „toptune“ en bestemt algoritme til en given platform. Metoden er dog alligevel blevet valgt idet det sikrer at algoritmerne er klart beskrevet i koden, og generelt gør koden mere overskuelig. Der er dog ingen tvivl om at der vil være en performance forbedring at hente, ved at skræddersy algoritmerne til de enkelte platforme.

9.1.2 Segment-piercing

Segment-piercing trekant-trekant testen består på overordnet plan af seks segment-trekant tests. Hvis en af disse rapporterer at der er kollision terminerer algoritmen. Hvis alle seks tests derimod rapporterer ikke-kollision er de to trekanter ikke i kollision.

Hver segment-trekant test består som beskrevet af en linie-trekant og segment-plan test. Der testes først for, om linien dannet af segmentet er inden for trekantens sider. Hvis dette er tilfældet skal det afgøres om segmentet kan nå at røre planet som trekanten ligger i vha. en segment-plan test.

Når en linie-trekant test udføres, ses der på om to af de udregnede trippel skalar produkt går hen og bliver nul. Dette svarer til at linien ligger i samme plan som trekanten. Dette rapporteres til segment-trekant testen.

I selve trekant-trekant testen opsamles resultaterne af de seks segment-trekant tests. Hvis to af dem rapporterer at segmenterne har ligget i samme plan som trekanten, kaldes koden der håndterer det koplanare tilfælde (beskrevet i afsnit 9.1.3).

9.1.3 Devillers

Devillers algoritmen er implementeret i to udgaver. Den første af disse er baseret på den originale kode, mens den anden er en omskrivning der forsøger at gøre algoritmen mindre krævende ved at reducere antallet af branches.

Den originale Devillers algoritme

Den originale Devillers algoritme er implementeret ved først at teste trekant T_1 hjørner mod det plan som T_2 ligger i. Dette gøres som tidligere beskrevet vha. tre trippel skalar produkter. Hvis T_1 ikke penetrerer T_2 plan terminere algoritmen med besked om at de to trekanter ikke er i kollision. Ellers testes T_2 mod planet som T_1 ligger i, og igen termineres hvis T_2 ikke penetrerer T_1 's plan. Hvis algoritmen på dette tidspunkt har fundet ud af at de to trekanter skærer hinandens planer, går den videre til at lave permutationen for at kunne sammenligne skæringspunkterne på linien hvor de to planer skærer hinanden (se afsnit 4.6). Dette fortages ved eksplicit at beskrive alle de mulige veje som permutationen kan resultere i. Dette giver anledning til en stor mængde branches i koden, og gør den desuden svært overskuelig. Efter permutationen er gennemført testet de to trekanter for overlap ved at udføre to trippel skalar produkter.

De grundlæggende matematiske funktioner anvendes på samme måde som beskrevet for segment-piercing.

Den modificerede Devillers algoritme

Som nævnt ovenfor er den originale devillers implementation ganske tung i forhold til branches, der er derfor blevet implementeret en alternativ udgave som beskrives her. Denne algoritme benytter samme

fremgangsmåde som den originale, og starter derfor først med at teste de to trekanter, T_1 og T_2 , mod hinandens planer som beskrevet ovenfor. Dernæst udføres permutationen som en egentlig permutation, vha. en række pointer operationer. Derved undgås det store antal branches. Efter permutationen er udført kontrolleres om de to trekanter overlapper på linien mellem de to planer, vha. to trippel skaler produkter.

Det koplanare tilfælde

Det koplanare tilfælde håndteres på to forskellige måder. Segment-piercing algoritmen benytter en metode som beskrevet i afsnit 4.5.2, mens de to devillers implementationer deles om implementation af algoritmen som er beskrevet i afsnittet 4.6.3.

Segment-piercing håndterer det koplanare tilfælde ved hjælp af en funktion der er i stand til at afgøre om et punkt ligger inden i eller udenfor en trekant. Dette gøres, som beskrevet i afsnit 4.5.2 med tre 2D krydsprodukter. Denne funktion kaldes herefter seks gange, først tester den trekant T_1 hjørner mod trekant T_2 (tre kald), og dernæst T_2 's hjørner mod T_1 (igen tre kald). Hvis blot et af disse kald rapporterer kollision er de to trekanter i kollision.

Devillers algoritmerne håndterer det koplanare tilfælde med en implemenation af den i artiklen foreslåede metode. Denne algoritme fortager først en orthografisk projektion, for at bringe trekanterne til \mathbb{R}^2 . Herefter er algoritmen praktisk talt en implementation af de to beslutnings træer som er vist i afsnit 4.6.3. Dette betyder naturligvis at algoritmen indeholder en hel del branches. I praksis forventes det ikke at være en ret stor del af testene der er koplanare, og der er derfor ikke gjort noget for at optimere dette.

9.1.4 Massive trekant tests på CELL

I dette afsnit vil implementationen af en massiv trekant-trekant tester blive diskuteret. Formålet med den massive trekant tester at at kunne teste to trekant supper for kollision. Det vil sige at alle trekanter i den ene suppe skal testes for kollision med alle andre trekanter i den anden suppe. Til implementationen er der benyttet standard C, med de udvidelser som IBM har lavet for at understøtte Cell platformen. For at undersøge de muligheder som CELL platformen giver for at løse denne type opgaver, er der blevet implementeret to forskellige udgaver af den massive test. Den ene udgave baserer sig på Libspe, som er et library der tillader mere eller minder lowlevel access til SPU'erne. Den anden udgave baserer sig på IBM's ALF² library. ALF arbejder på et højere abstraktions niveau end Libspe, og kan derved være lettere at benytte i en række sammenhænge. Netop dette er grunden til at der er valgt at lave to implementationer. Hermed er håbet at finde ud af hvilke abstraktions niveau der er det rette at benytte til implementation af det endelige system.

Input og output datastrukturer

For at få maksimal performance ud af Cell processoren er det helt afgørende, at memory flowet bliver styret på en hensigtsmæssig måde. For at opnå dette tager designet af de benyttede datastrukturer udgangspunkt i netop hvordan de er repræsenteret i hukommelsen. Da de to objekter der skal testes for kollision er bygget op af en række trekanter, skal alle trekanter fra den ene objekt testes mod alle trekanter fra det andet objekt. Hvis objekt 1 antages at indeholde N trekanter og objekt 2 antages at indeholde M trekanter, giver dette anledning til et $M \times N$ resultatmatrix, som vist på figur 9.1.

²Accelerated Library Framework.

		Objekt 1				
		┌───────────┐				
		T_{11}	T_{12}	...	T_{1N}	
Objekt 2	{	T_{21}	?	?	?	?
		T_{22}	?	?	?	?
		...	?	?	?	?
		T_{2M}	?	?	?	?

Figur 9.1: $M \times N$ resultat matrix

Til at starte med er resultat matrixen tom, men efterhånden som algoritmen får testet de enkelte trekanter mod hinanden udfyldes det tilsvarende element i matrixen, med enten et "0" hvis der ikke er kollision, eller et "1" hvis der er kollision. De to objekter er i kollision, hvis resultat matrixen har et eller flere elementer indeholdende "1", når algoritmen terminerer. De to objekter gemmes som to arrays, hvor hvert element i et array består af en enkelt trekant. En trekant er repræsenteret som en `struct` af tre punkter, hvor hver punkt er repræsenteret som vektor af 4 floats (en speciel indbygget `Cell` type). En trekant fylder altså $3 * 4 * 4 = 48$ bytes, se afsnit 9.1.1 for definitionen.

Resultat matrixen er repræsenteret som et array af størrelse $M * N$ ³.

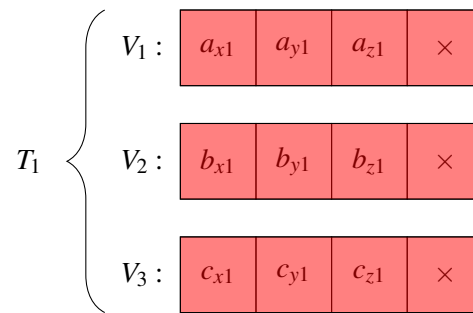
Arrayet ligger altså som en lang række bytes i hukommelsen, dette gør det mere effektivt for SPE'erne at skrive resultater i arrayet. I øvrigt stiller DMA controllerne også krav om alignment af arrayet, idet dette som minimum skal alignes til en 16 byte boundary. Det skal bemærkes at større alignment kan benyttes, så længe dette er et multiplum af 16. Faktisk benyttes 128 byte alignment i praksis, idet denne størrelse passer med en cahceline og derfor teoretisk giver en bedre performance. DMA controllerne stiller ligeledes krav om at blokstørrelsen skal 1, 2, 4 eller 8 bytes, eller et multiplum af 16. For at tilgå element (x, y) i resultat matrixen kan formelen $y * N + x$ benyttes til at omsætte koordinatet til et array index.

Array of structures vs. Structure of Arrays

I afsnittet ovenover nævnes det at en trekant på CELL platformen repræsenteres ved at hvert hjørne beskrives af en vektor (på CELL platformen er det 4 floats). Denne form for repræsentation kaldes normalt for *Array of structures* (AOS). Figur 9.2 viser hvordan en trekant T_1 gemmes i fire vektorer V_1, V_2, V_3 og V_4 . De fire hjørner a, b og c består hver af tre floats (subscript x, y , og z), og de gemmes derfor i tre komponenter i hver sin vektor. Den sidste komponent i alle vektorene benyttes ikke.

Imodsætning til AOS kan en datarepræsentation kendt som *Structure of arrays* (SOA) også benyttes. Figur 9.3 viser hvordan denne datarepræsentation tager sig ud. Det første der skal bemærkes er at der nu skal benyttes 9 vektorer, og at alle komponenter i alle vektorer benyttes. Tilgængelig er der nu blevet plads til 4 trekanter, hvor alle a hjørnernes x komponent er samlet i en vektor, y komponenterne

³I praksis kan arrayet godt være større, hvis $M * N$ ikke går op i 16. I så fald „paddes“ et antal bytes, for at få størrelsen til at gå op i 16.



Figur 9.2: Illustration af AOS.

er samlet i en anden vektor og så videre. Det ses at denne repræsentation kan føre til en mere kompakt fremstilling af en række trekanter, såfremt man har trekanter nok.

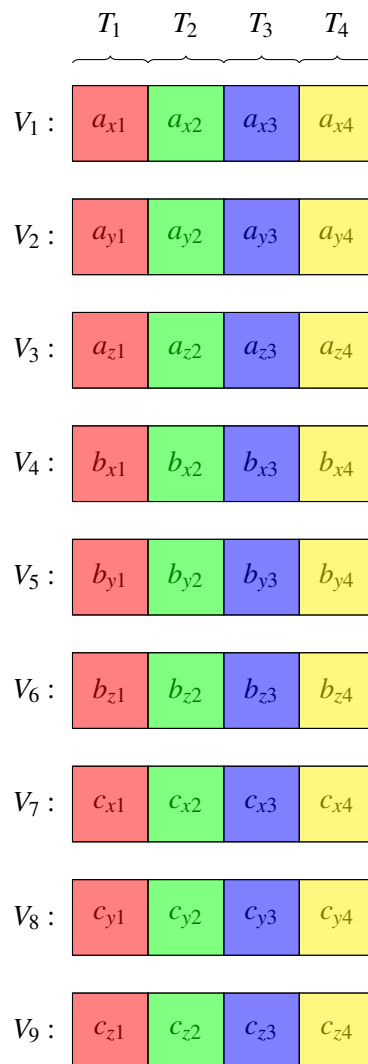
At repræsentationen er mere kompakt har naturligvis fordele, ikke mindst på en platform hvor det er vigtigt at udnytte båndbredden mellem processor elementerne bedst muligt. Desuden betyder SOA modellen også en bedre udnyttelse af hukommelsen på SPU'erne, igen fordi repræsentationene er mere kompakt.

Når så AOS er valg frem for SOA i dette projekt skyldes det en række forhold. Algoritmerne der benyttes i projektet er alle baseret på en række vektor operationer. Dette betyder at der i de fleste beregninger (f.eks krydsprodukt, eller prikprodukt) indgår 3D koordinater. Hvis AOS modellen benyttes betyder det at det er lige til at udføre sådanne beregninger, idet en vektor direkte indeholder både x , y og z koordinaterne. SOA modellen derimod nødvendiggøre at en sådan beregning udføres på 4 koordinatsæt af gangen. I praksis vil det sige at der f.eks testes for kollision mellem 4 trekanter af gangen. Dette er de nuværende algoritmer ikke beregnet på, da de er lavet så de kan benyttes på alle tre platforme. Det ville derfor være nødvendigt at omskrive algoritmerne for at kunne benytte SOA modellen. Hvad der eventuelt ville kunne vindes ved en sådan omskrivning er svært at vurdere. Det er klart at der ville kunne overføre flere data på den samme tid som der benyttes nu. Men det er også klart at det ville blive vanskeligt at udnytte SOA modellen i selve beregningerne. Hvis der eksempelvis testes for kollision på 4 trekanter ad gangen, og de to af dem hurtigt udelukkes fra at være i kollision, så skal der tages specielle hensyn for at udnytte SIMD operationerne optimalt i de resterende beregninger.

Alt i alt er det blevet vurderet at AOS modellen passer bedst ind i det samlede projekt. Det kunne dog være interessant at prøve at omskrive f.eks. Devillers algoritmen for at se om SOA modellen kunne give en performance løft. Dette har der desværre ikke været ressourcer til i dette projekt.

Partitioneringen - Libspe

For at fordele arbejdsbyrden mellem SPE'erne fordeles trekant testene til SPE'erne således at hver SPE bliver ansvarlig for at beregne resultaterne i en del af den samlede resultat matrice. Fordelingen af testene resulterer i en række såkaldte workunits. En SPE behandler en workunit, men der kan sagtens være flere workunits end SPE'er. I så fald skal de enkelte SPE'er behandle mere end en workunit. En workunit består af en pointer til en trekant fra hvert objekt, samt en angivelse af hvor mange af de efterfølgende trekanter fra hvert objekt der skal testes. Derudover indeholder en workunit en pointer til det sted i resultat matricen resultaterne skal skrives. Hvor mange resultater der skal beregnes afgøres af en konstant, den såkaldte workunit størrelse. Det er klart at jo flere tests en SPE skal udføre jo bedre. Det er dog begrænset hvor meget data der er plads til idet en SPE kun har 256 KiB lokal



Figur 9.3: Illustration af SOA.

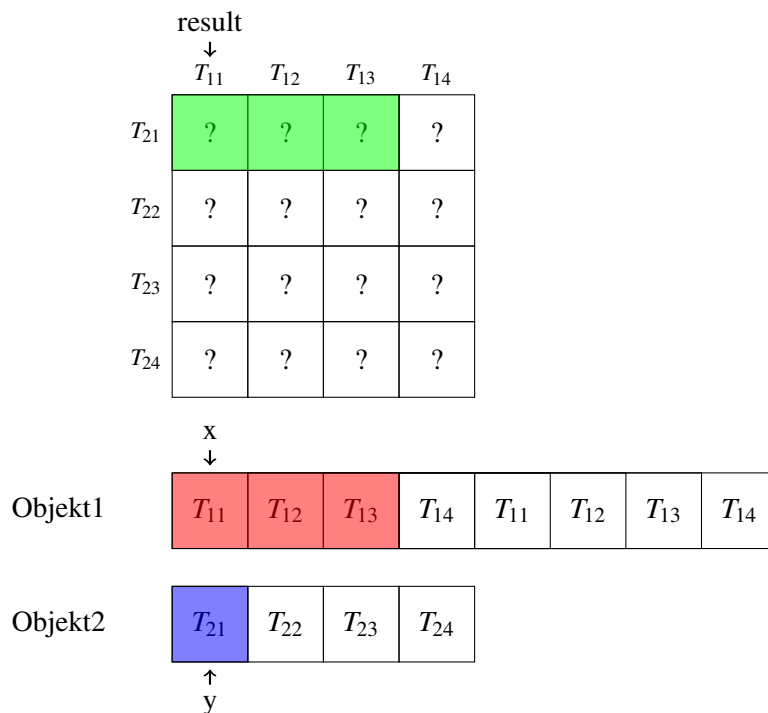
hukommelse. Denne hukommelse skal ud over data også rumme programmet og programmets stack. Så generelt skal følgende ulighed være opfyldt

$$p_s + s_s + r_s + i_s \leq 256KiB \tag{9.1}$$

Hvor p_s er programmets størrelse i bytes, s_s er stack størrelsen i bytes, r_s er resultat arrayets størrelse i bytes og i_s er input bufferes størrelse i bytes.

Det der med andre ord er ønskværdigt er at maksimere antallet af resultater (gøre r_s så stor som muligt). Dette kan naturligvis opnåes på flere måder. Program størrelsen og stack størrelsen kan mindskes, men kun til en vis grænse. Tilbage er der input størrelsen, dvs. antal trekanter der skal bruges til at beregne resultaterne. Får at maksimere antallet af resultater, ved kollisionstest mellem to sæt af trekanter, kan det let indses at hver trekant fra det ene sæt skal testes for kollision mellem alle trekanter fra det andet sæt. På den måde opnåes $m \times n$ resultater med hhv. m og n trekanter i de to sæt. Som det beskrives senere er dette dog ikke altid muligt, og antallet af resultater bliver dermed mindre end den optimale ($r_s < n \times m$).

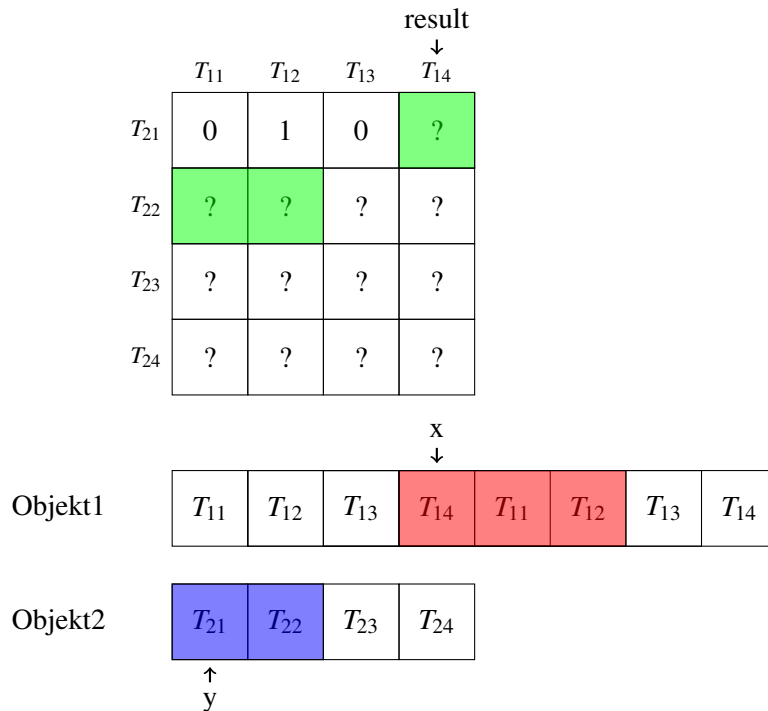
Figur 9.4 viser et eksempel på hvordan en workunit er opbygget. I dette eksempel benyttes en workunit størrelse på 3. Pointeren `result` peger på det sted i resultat matricen hvor det første resultat skal skrives, de resultater der skal udfyldes er markeret med grønt. De to objekt arrays er vist under resultat matricen. Her ses hvilke trekanter der er nødvendige for at udføre testene. Pointerene `x` og `y` peger på den første trekant fra hver objekt, og farverne rød og blå markere de trekanter der skal benyttes. Bemærk at trekanterne i objekt 1 er duplikeret.



Figur 9.4: Illustration af den første workunit.

Figur 9.5 viser hvordan den næste workunit ser ud. Her bemærkes det hvorfor objekt 1 er duplikeret. Når en workunit “wrapper” rundt er det nødvendigt at tage nogle af de sidste trekanter i objekt 1

med, men det er også nødvendig at tage nogle af de første med. Derfor emuleres en form for cirkulær buffer, ved at gentage objekt 1 to gange. Det ses i øvrigt at den første workunit har rapporteret kollision mellem trekant T_{12} og trekant T_{21} .



Figur 9.5: Illustration af den anden workunit.

Herunder er vist den struct der definerer en workunit. De forskellige pointere genkendes fra figur 9.4 og 9.5. Derudover skal det bemærkes at det er nødvendigt at padde 8 dummy byte, igen for at tilfredsstille DMA'erne. Mode feltet er pt. ikke i brug, men det er tænkt som en option til at bestemme om resultat matricen skal udfyldes, eller om der kun skal rapporteres om der er kollision eller ej. Det er også nødvendigt at overføre det antal resultater der skal beregnes, idet det ikke blot kan antages at antallet af resultater er $xsize$ gange $ysize$. Antallet af trekanter i objekt 1 ($olsize$) og start indexet i objekt 1 ($olidx$), er nødvendige at kende på SPU'erne, da de skal kunne beregne hvornår resultatet wrapper rundt.

```

1 typedef struct workunit_s
2 {
3     int id;           /**< Unique id identifying the workunit */
4     wu_modes_t mode; /**< Mode of the workunit */
5     char *x;         /**< Pointer to the wrapped triangles from object 1 this
6                      * workunit is to process */
7     size_t xsize;    /**< Number of triangles from object 1 */
8     char *y;         /**< Pointer to the wrapped triangles from object 2 this
9                      * workunit is to process */
10    size_t ysize;    /**< Number of triangles from object 1 */
11    char *result;    /**< Pointer to the place in the result matrix where
12                      * this work unit is supposed to store its
13                      * results */
14    int nres;        /**< Number of results to obtain */

```

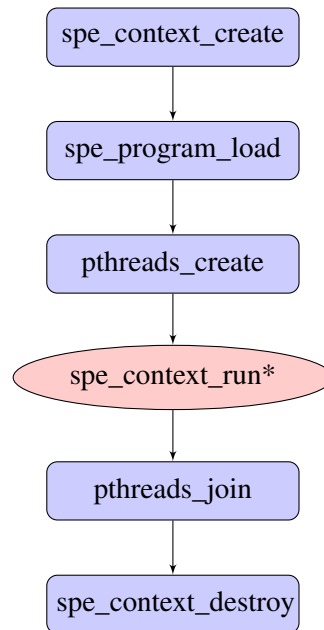
```
15  int  olsize;      /**< Number of triangles in object1 */
16  int  olidx;      /**< Start index into object 1 */
17  char __pad[8];   /**< Pad 8 dummy bytes to make struct size a
18                      * multiple of 16 (required by DMA) */
19  } workunit_t;
```

En workunit overføres til en SPE, og det er derefter op til denne at hente de nødvendigt trekanter over i den lokale hukommelse. SPE'en gør dette ved at initiere en række DMA overførsler fra de 2 objekt arrays der ligger i hovedhukommelsen. Som tidligere nævnt stiller DMA kontrolleren en række krav, så der er en del arbejde i udføre disse overførsler. Det vil føre for vidt at komme ind på alle detaljer her, så her skal blot nævnes at data skal alignes korrekt i både lokal- såvel som hovedhukommelsen, samtidig skal længderne af overførslerne passe. For flere detaljer henvises til kildekoden. Det er værd at bemærke at eftersom alle datastrukturer i hovedhukommelsen er opbygget som kontinuerlige regioner, så kan DMA overførslerne forgå effektivt.

Når en SPE har en kopi af de trekanter den behøver kan den begynde at udføre trekant-trekant tests. Programmet er opbygget så der kan vælges mellem de to algoritmer ved hjælp af en `define`. Når en SPE er færdig med at teste en workunit, kopierer den resultatet til resultat matricen i hovedhukommelsen ved hjælp af en DMA overførsel.

Styring af program flow - Libspe

For overhovedet at kunne benytte SPE'erne, er det nødvendigt at overføre et program til dem og derefter starte programmet. Dette forgår ved at oprette en såkaldt kontekst og derefter loade SPE programmet ind i denne kontekst. Dernæst kan konteksten køres på en SPE. Det er muligt at overføre parametre til SPE programmet når dette startes, denne facilitet benyttes til at overføre en pointer til den workunit som SPE'en skal behandle. Da kaldet til funktionen der kører en kontekst blokerer indtil SPE programmet er færdigt, oprettes der et antal tråde (en for hver workunit) på PPE'en. I hver af disse tråde startes der så et SPE program, og når alle tråde er færdige er alle workunits behandlet.



* = Eksekveres i separat tråd.

Figur 9.6: Flowchart over trekant-trekant test på CELL med Libspe.

På figur 9.6 ses et forenklet flow diagram over hvordan koden afvikles under Libspe.

Partitioneringen - ALF

Partitioneringen af problemet forgår på samme måde i ALF implementationen, som i Libspe implementationen. Det er dog nødvendigt at beskrive en arbejdsenhed⁴ på en lidt anden måde. Herunder ses definitionen af en ALF workbock.

```

1 struct workblock {
2     int id;
3     int obj1_size;      /**< Total number of triangles in object 1. */
4     int obj2_size;      /**< Total number of triangles in object 2. */
5     int obj1_part;      /**< Index into the original object 1 telling where
6     this workblock starts. */
7     int obj2_part;      /**< Index into the original object 1 telling where
8     this workblock starts. */
9     int obj1_part_size; /**< The number of triangles from object 1 this
10    workblock consists of. */
11    int obj2_part_size; /**< The number of triangles from object 2 this
12    workblock consists of. */
13    int result_part;     /**< Index into the result array, where this workblock
14    is to store its results. */
15    int result_part_size; /**< Number of results this workblock is to calculate.
16    */
17    char __pad[12];      /**< Padding to make struct conform to DMA
18    requirements */
  
```

⁴En arbejdsenhed er den delmængde af det samlede problem en SPU behandler på et givet tidspunkt. En arbejdsenhed kaldes en workunit i Libspe implementationen, mens den kaldes en workblock i ALF implementationen.

```
12 };
```

Som det ses er det essentielt de samme oplysninger der indgår i en workblock som der indgår i en Libspe workunit. Den største forskel består i at der benyttes indekser i stedet for pointere til at udpege data. Dette skyldes at ALF selv står for at overføre data til de enkelte SPU'er, hvor dette skal håndtere manuelt i Libspe. Desuden er der lidt ekstra oplysninger, bla. den totale størrelse af begge objekter. Disse ekstra oplysninger er taget med for at gøre workblocken mere generel, men er egentligt ikke strengt nødvendig i den nuværende implementation.

De samme overvejelser vedrørende maksimering af antallet af resultater kontra størrelsen af hhv. programmet, stacken og input data, gør sig gældende for ALF implementationen som for Libspe implementationen (se afsnit 9.1.4). Det skal dog bemærkes at eftersom ALF benytter statisk allokering af hukommelsen, skal størrelsen af eksempelvis input bufferen angives på forhånd. Det vil sige at bufferstørrelserne bliver nød til at være store nok til at håndtere det største input. Det er med andre ord ikke muligt at optimere for hver enkelt kørsel.

Ud over disse overvejelser stiller ALF også nogle krav til hukommelses forbruget på den enkelte SPE. Dette sker fordi ALF selv vælger hvilken buffering den benytter. For at få ALF til at benytte double buffering, er der altså nogle krav der skal overholdes. For at få et overblik over disse krav henvises der til [15].

Styring af program flow - ALF

Selve ALF API'et består af en række funktioner til at bla. at oprette tasks og tilføje workblocks til disse. For at gøre det nemmere at arbejde med dette API, er en stor del af funktionaliteten blevet pakket ind i en række funktioner og tilhørende datastrukturer. Den vigtigste af disse datastrukturer er vist herunder.

```
1 struct alf_sys {
2     alf_handle_t handle;           /**< Handle to the system */
3     void *config_parms;          /**< Internal pointer to config */
4 };
```

Den beskriver et ALF system, og indeholder vigtig information der skal benyttes i alle efterfølgende kald til ALF. En test (enten OBB eller trekant) beskrives ved hjælp af en ALF task, og for at holde styr på alle de parametre og data der knytter sig til en sådan benyttes en datastruktur som vist her.

```
1 struct alf_task {
2     alf_task_handle_t handle;      /**< Handle to the task */
3     struct task_parm task_parm;    /**< Per task parameters */
4     struct alf_task_desc task_desc; /**< Description of the task */
5     int n_workblocks;             /**< Number of workblocks */
6     struct workblock *workblocks;  /**< Array of workblocks */
7 };
```

Denne struktur indeholder bla. en anden datastruktur der beskriver en række parametre forbundet med tasken. Denne struktur ses her.

```
1 struct alf_task_desc {
2     char *accl_lib_name;           /**< Name of the accelerator library */
3     char *accl_img_name;           /**< Name of the accelerator image */
4     char *comp_func_name;          /**< Name of the accelerator computation
5     function */
6     char *input_prepare_func_name; /**< Name of the input prepare funcion */
7     char *output_prepare_func_name; /**< Name of the output prepare funcion */
```

```

7   int stack_size;           /**< Stack size to use on the accelerators
   */
8   int input_buffer_size;   /**< Size in bytes of the input buffer */
9   int output_buffer_size;  /**< Size in bytes of the output buffer */
10  int overlapped_buffer_size; /**< Size in bytes of the overlapped I/O
   buffer */
11  int workblock_parm_buffer_size; /**< Size in bytes of the per workblock
   parameters */
12  int task_ctx_buffer_size; /**< Size in bytes of the per task
   parameters */
13  int max_dtl_entries;      /**< Maximum number of entries in a DTL */
14 };

```

Som det fremgår indeholde denne struktur bla. navnet på det library der indeholder accelerator koden, samt navnet på funktionen der skal kaldes i dette library for at udføre testen. Desuden beskrives størrelserne på de forskellige buffere som ALF benytter, herunder input buffer og output buffer. Dette skyldes at ALF benytter en statisk allokering af hukommelsen på accelerator noderne, for at optimere performance. Derfor er det nødvendigt på forhånd at angive disse buffer størrelser, samt en mængde andre størrelser f.eks. stack størrelsen på de enkelte accelerator noder.

Hver task er altså forsynet med en sådan struktur der beskriver en række parametre. Den er desuden udstyret med en liste af workblocks, der tilsammen udgør det samlede problem der ønskes løst. I dette tilfælde vil det sige en kollisionstest mellem to OBB eller trekant supper. Med ovenstående datastrukturer in mente, kan det nu beskrives hvordan selve program flowet fungerer. Først oprettes et ALF system (`struct alf_sys`), ved et kald til funktionen `init_alf_sys`. Dernæst oprettes en ny ALF task (`struct alf_task`), dette sker ved at kalde funktionen `create_alf_task`. Når dette er gjort, kaldes `partition`, der er en funktion der partitionere det samlede problem til en række mindre problemer der han håndterer af de enkelte SPU'er. Dette sker ved at oprette en række workblocks (`struct workblock`) og tilføje dem til den ny oprettede task. Desuden fortæller `partition` funktionen også hvor stor input og output buffer der er behov for, for at kunne behandle de oprettede workblocks. Denne information benyttes til at oprette en beskrivelse af ALF tasken. Dette sker ved et kald til funktionen `set_task_desc`, der opretter en `struct alf_task_desc` og knytter denne til ALF tasken. De parametre der er angivet i `struct alf_task_desc` overføres herefter til ALF ved et kald til `prepare_alf_task`. Ved kaldet til denne funktion angives det endvidere at der er tale om en trekant test der skal udføres. Dette skyldes at det samme API benyttes til at udføre OBB tests. Det sidste trin før selve testen kan udføres er at ligge de oprettes workblocks i kø til at blive udført. Dette sker ved et kald til `enqueue_alf_workblocks`. Slutteligt er der bare tilbage at kalde funktionen `finalize_alf_task`. Denne funktion udføre selve testen og blokerer indtil alle workblocks er blevet processeret af SPU'erne.

Alle de nævnte funktioner er de omtalte wrapper funktioner der er blevet lavet for at pakke ALF API'et ind. Hver af de omtalte funktioner kalder derfor indtil flere ALF API funktioner for at udføre deres arbejde. Der henvises iøvrigt til IBM's ALF dokumentation (se [15] for en nærmere forklaring til ALF API'et. Af de ovennævnte funktioner, er `partition` den egentlige arbejdshest. Denne funktion udføre partitioneringen som beskrevet i afsnit 9.1.4, men derudover håndtere den også de memory krav som DMA'erne stiller til alignment og størrelsen på de enkelte DMA overførelser. Der er ligeledes udviklet en række funktion til at støtte håndteringen af de specielle krav som DMA'erne stiller, se kilde koden for nærmere. Desuden henvises til den omfattende doxygen dokumentation (omkring 600 sider, så husk at lave kaffe), der kan autogeneres ud fra kilde koden i både HTML og \LaTeX format. Dette gøres ved at benytte det specielle make target `doc` (`make doc`).

9.1.5 Massiv trekant test på CUDA

I dette afsnit vil implementationen af massiv trekant-trekant test på CUDA blive beskrevet. Den massive trekant-trekant test udfører kollisionstest på to trekant supper ved at tage hver trekant i den ene suppe og tester den imod alle andre trekanten i den anden suppe.

Som beskrevet i afsnit 9.1.4, består testen i at tage N trekanten fra objekt 1, teste dem imod M trekanten fra objekt 2 og derefter udfylde en $M \times N$ resultatmatrix som vist på figur 9.1.

De to objekter er repræsenteret som to arrays i hukommelsen, hvor hvert enkelt element er en trekant. En trekant er beskrevet ved en struct af tre punkter, der hver er et array af fire single-precision floats. En trekant fylder derfor $3 \cdot 4 \cdot 4 = 48$ bytes. GPU'en er i stand til med en enkelt instruktion ([25]) at tilgå 32-, 64- og 128-bits words i global memory. Ved at bruge `__align__(16)` specificieren på trekantdefinitionen, sikres det at trekanten kan tilgås med kun 3 styk 128-bits loads.

$M \times N$ resultatmatrixen er, af hensyn til dynamisk allokering allokeret som et enkelt 1-dimensionelt array.

Partitioneringen

CUDA's evne til at organisere eksekverende tråde i flere dimensioner (som beskrevet i 5.5.1) gør det muligt at lave en ligetil mapping af problemet. Partitioneringen af kollisionstest er som vist på figur 9.7.

Hver trådblok er ansvarlig for en selvstændig, kvadratisk del af grid'et, og kan køre helt uafhængigt af andre trådblokke. Disse trådblokke er identificeret ved deres position i grid'et, jævnfør figur 9.7. I CUDA er en trådbloks position tilgængelig i den globale vektor `blockIdx`.

Hver tråd i en trådblok er ansvarlig for én trekant-trekant kollisionstest. En tråd er identificeret ved dens position i trådblokken. Denne information er tilgængelig i den globale vektor `threadIdx`.

For maksimal ydelse af den enkelte MP er det interessant at have så mange samtidige tråde som muligt. Mange tråde gør det nemmere for den interne scheduler at effektivisere time-slices under kørslen af warps ([25]). Det maksimale antal der understøttes for 8800 GTX, er som beskrevet i afsnit 5.6 512 tråde. Antal tråde er dog oftere, for mere komplicerede kerner, begrænset af andre tilgængelige ressourcer. Dette er hovedsageligt antallet af registre der anvendes per tråd. Per afsnit 5.2 af [25] er antal registre der er tilgængelige per tråd

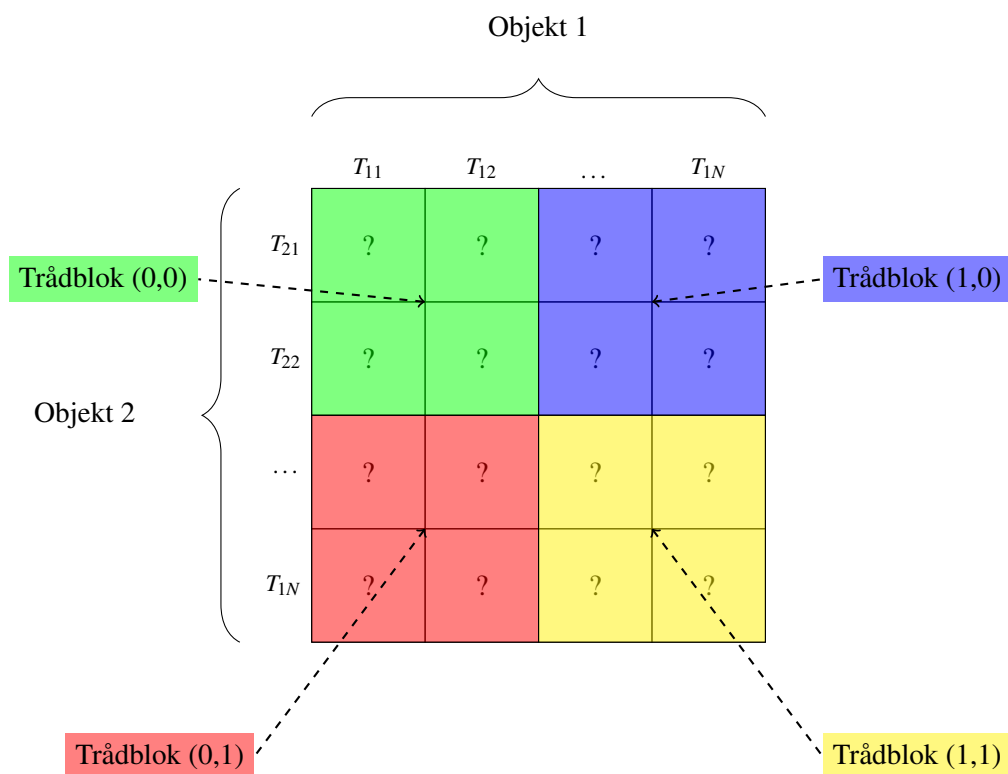
$$\frac{R}{B \cdot \lceil T \rceil_{32}} \quad (9.2)$$

Hvor R er antal registre per MP, B er antallet af aktive trådblokke og $\lceil T \rceil_{32}$ er antal registre per tråde rundet op til et multiplum af 32.

Som det blev nævnt i 5.5.3 bliver en trådblok i en MP eksekveret samtidigt med andre blokke i batches, hvis der vel og mærke er overskydende ressourcer (registre og hukommelse) til rådighed. Antallet af aktive trådblokke per MP er i ovenstående formel B . Afhængigt af det de anvendte ressourcer vil GPU'en automatisk vælge B gående fra 1 til de maksimalt 8 der er grænsen for 8800 GTX.

Givet en kerne der kræver K registre for at afvikles, kan det maksimale antal tråde i en trådblok T findes som

$$T = \frac{R}{K} \cdot 32 \quad (9.3)$$



Figur 9.7: Eksempel på et 4×4 grid indeholdende 4 trådblokke af dimension 2×2 . Hver trådblok er ansvarlig for sit eget afgrænsede område af trekant-trekant tests.

hvor der udføres heltals-division og $B = 1$.

Problemet kan udtrykkes som opgaven med at udfylde alle felter i $M \times N$ resultatmatricen. Givet et problem af en arbitrær størrelse, skal der foretages en partitionering af problemområdet i trådblokke. CUDA's eksekveringsmodel foreskriver en partitionering af problemet på tråde, grupperet i trådblokke i et grid. Dimensionen af en trådblok er i CUDA terminologi vektoren `blockDim`, og dimensionen af grid'et vektoren `gridDim`.

Konfigurationen til en eksekvering bliver gemt i en struct for hver massiv trekant-trekant test

```

1 struct cuda_exec_cfg {
2     dim3 gd;    /* grid dimension */
3     dim3 tbd;  /* threadblock dimension */
4 };

```

Da hver tråd i en trådblok tager sig af at udfylde et enkelt felt skal følgende være opfyldt

$$blockDim.x * blockDim.y = N \quad (9.4)$$

$$blockDim.y * blockDim.x = M \quad (9.5)$$

Ved at indføre logik i kernen, der kan håndtere tilfældet af at ikke alle tråde skal køre en test kan kravet afslappes til

$$blockDim.x * blockDim.y \geq N \quad (9.6)$$

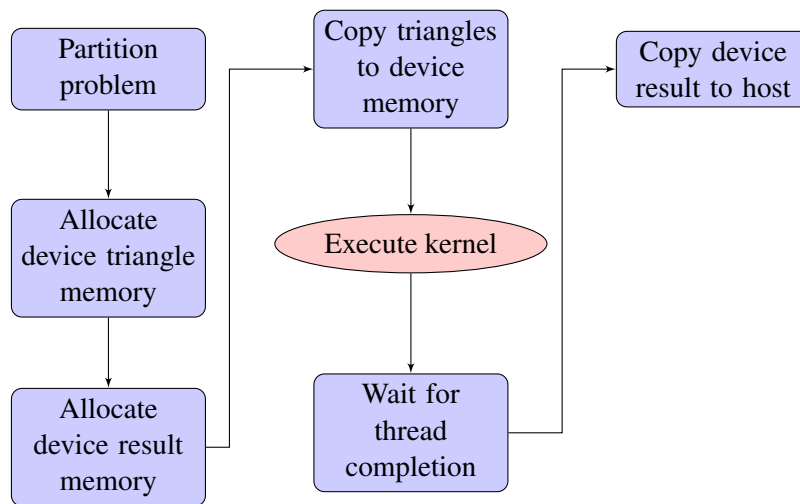
$$blockDim.y * blockDim.x \geq M \quad (9.7)$$

Antallet af tråde i en trådblok ønskes maksimeret, da det som tidligere beskrevet giver den bedste ydelse. Derfor sættes `blockDim` til en fast størrelse og `gridDim` løses til at opfylde ligning 9.6.

I praksis gøres dette ved at udføre heltalsdivision af N og M med henholdsvis `blockDim.x` og `blockDim.y`. Hvis remainderen af divisionerne ikke giver nul, udvides grid'et til at have en enkelt ekstra trådblok.

Styring af programflow

Figur 9.8 viser programflowet i massiv trekant-trekant test på CUDA.



Figur 9.8: Flowchart over trekant-trekant test på CUDA.

Efter at partitioneringen af problemet er overstået, allokeres der plads i device memory på GPU'en til resultatmatricen. Trekanterne kopieres derefter til GPU'en og beregningskernen startes. Når denne terminerer, er felterne i resultatmatricen i device memory udfyldt. Denne kopieres så til et resultatarray i systemets hovedhukommelse.

I CUDA er der specielle direktiver til at kontrollere og eksekvere en kerne. Selve kernen er i kildekode specificeret med modifieren `__global__`, som vist i følgende kildekode.

```

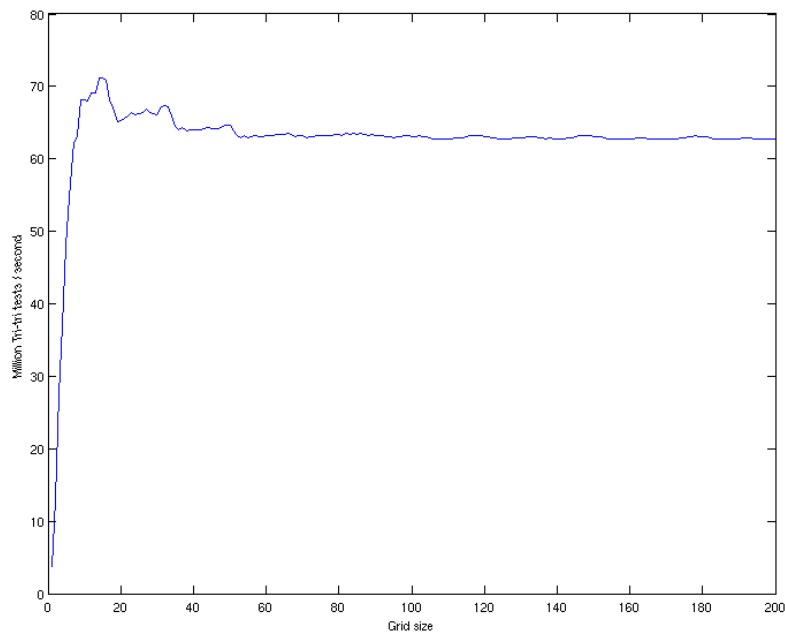
1  __global__ void cuda_tritri_test_kernel(triangle_t *t1, int t1_len, triangle_t *
2  t2, int t2_len, opp_result_t *res)
3  {
4  const unsigned int tx = threadIdx.x;          /* x thread index within
5  thread block */
6  const unsigned int ty = threadIdx.y;          /* y thread index within
7  thread block */
8  const unsigned int gx = blockIdx.x * blockDim.x + tx; /* global thread x-
9  index into triangle array */
10 const unsigned int gy = blockIdx.y * blockDim.y + ty; /* global thread y-
11 index into triangle array */
12
13 if ((gx < t1_len) && (gy < t2_len)) {
14     res[gy * t1_len + gx] = tri_tri_intersect_test_3d(&t1[gx], &t2[gy]);
15 }
16 }
  
```

Ovenstående kerne implementerer massiv trekant-trekant test som beskrevet af to trekant supper t_1 og t_2 af længde N og M . Hver enkelt tråd tager sig af en trekant-trekant test, og skriver resultatet i $M \times N$ resultatmatricen. Tråde der måtte indeksere ud over længden af trekant supperne udfører ikke noget arbejde. Dette vil dog ikke i den parallelle processing resultere i en væsentlig længere beregningstid. Under implementationen af trekant-trekant test algoritmerne har der været en række udfordringer i at implementere dem på CUDA. Disse erfaringer er beskrevet i afsnit 11.2.

9.1.6 Testresultater

Variation af grid størrelse

Figur 9.9 og 9.10 viser for henholdsvis Segment-piercing og Devillers et plot over deres ydelse som funktion af grid'ets størrelse.



Figur 9.9: Ydelse af GPU med Segment-pierce algoritme som funktion af grid størrelse ved en trådblokstørrelse på 13×13 .

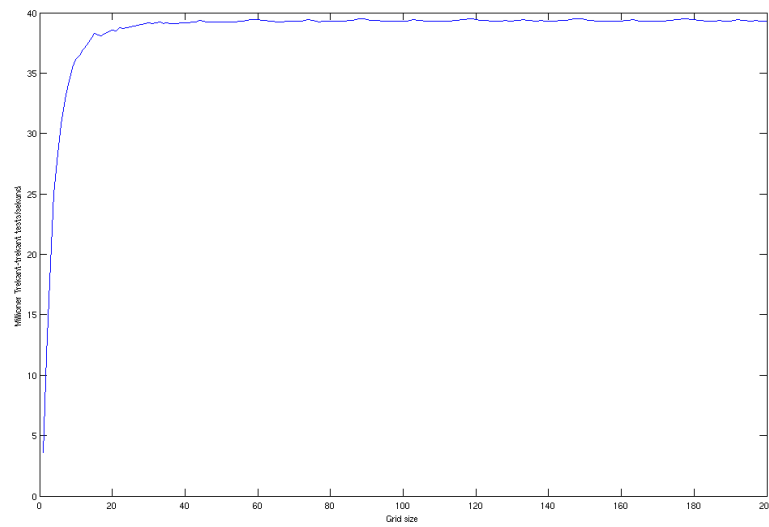
Segment-pierce algoritmen er den højstydende med en peak på 71.2 millioner trekant-trekant tests per sekund ved en grid størrelse på 14×14 . Dette svarer til 33124 trekant-trekant tests (to ens objekter på 182 trekanten). Det er værd at bemærke at ydelsen falder lidt igen ved større grids.

Devillers algoritmen har en lidt lavere ydelse der maxer ud ved ca. 39 millioner trekant-trekant tests per sekund. Interessant er det at den allerede ved en grid størrelse på 9×9 når algoritmen 90% ydelse og fortsætter med at stige roligt. Dette svarer til henholdsvis 13689 trekant-trekant tests (objektstørrelser 117 trekanten hver). Ved 14×14 er ydelsen inden for 95% af max. Den dårligere ydelse af Devillers skyldes med stor sandsynlighed, at Devillers algoritmen indeholder et stort antal branches, hvorimod Segment-pierce kun har ganske få. Branches er kostbare på GPU'en og dette kan mærkes.

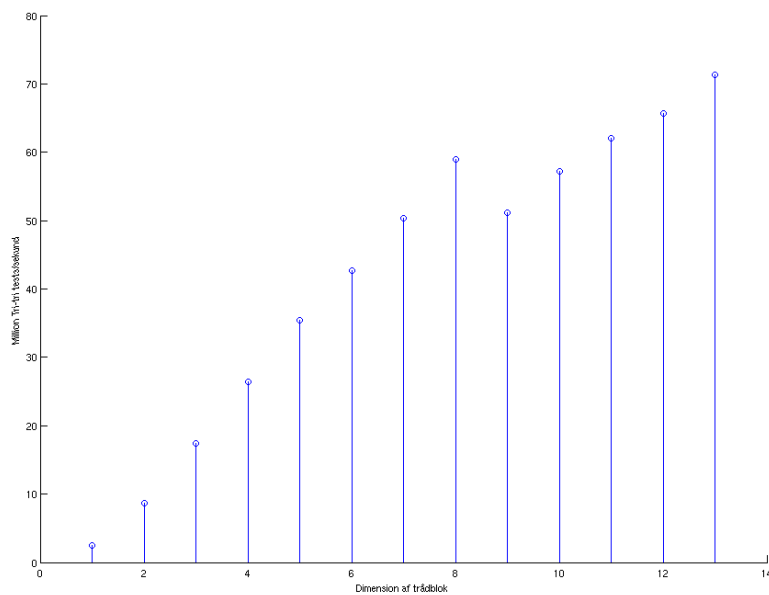
Variation af trådblok dimension

Figur 9.11 og 9.12 viser ydelsen af henholdsvis Segment-pierce og Devillers som funktion af trådblokkens størrelse.

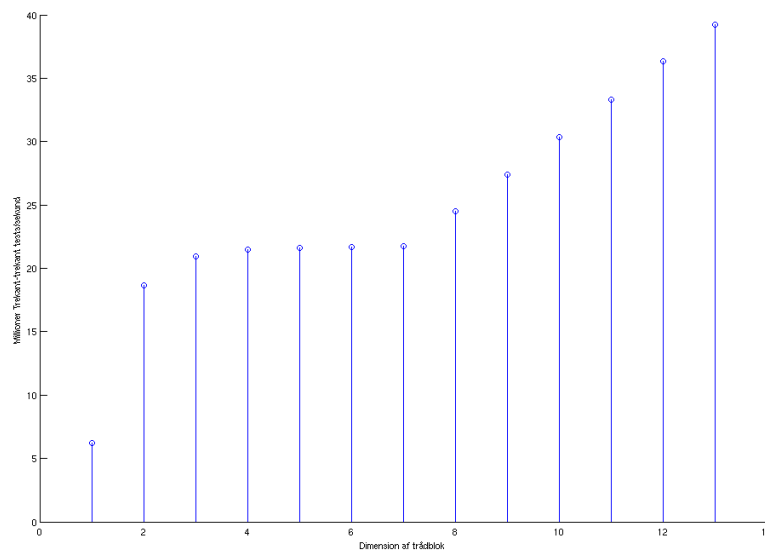
For begge algoritmer kan det ses at ydelsen stiger jævnt når trådblokkens dimension stiger og dermed forøger antallet af kørte tråde. Devillers har dog et mere ujævnt forløb.



Figur 9.10: Ydelse af GPU med Devillers algoritme som funktion af grid størrelse.



Figur 9.11: Ydelse af GPU med Segment-pierce algoritme som funktion af trådblok dimension.



Figur 9.12: Ydelse af GPU med Devillers algoritme som funktion af trådblok dimension.

9.2 OBB tests

9.2.1 Generelt

En oriented bounding box (OBB) er defineret som beskrevet i afsnit 4.7.2. Implementationen er

```

1 typedef struct obb_s
2 {
3     struct transformation_matrix t;
4     vec3d d;
5 } obb_t;

```

OBB'ens halv-dimensioner i x-, y-, og z-retningen er givet ved d som en 3D vektor af floats. Rotation og translation er indeholdt i den homogene transformationsmatrice t . Denne indeholder transformationen⁵, der skal bruges for at tage world-ramen og flytte den til at være aligned med OBB'ens frame.

De forskellige OBB-OBB test algoritmer er implementeret med samme interface, så at det er let at skifte imellem dem. Et kald til algoritmen gennem dette interface udfører en enkelt test imellem to OBB'er, A og B . Af hensyn til Gottschalk algoritmen tager interfacet ikke imod to individuelle OBB'er. Det skal derimod have transformation af boks B i forhold til A ($\mathbf{F}^A)^T \mathbf{F}^B$ som beskrevet i afsnit 4.7.2) samt deres dimensioner.

9.2.2 Exhaustive

Implementationen af exhaustive edge-face algoritmen, beskrevet i afsnit 4.7.1 tester alle edges fra en OBB A imod alle faces fra en OBB B og vice-versa.

Til at teste liniesegment dannet af en edge fra A imod en face fra B , genbruges segment-trekant testen fra segment-piercing trekant-trekant testen beskrevet i afsnit 9.1.2.

Segment-plan testen er brugt direkte, ved kun kalde testen med 3 af de 4 kanter der udgør en OBB's face. Linie-trekant testen er skrevet om til en linie-obb-face test. Her testes der for om linien er indenfor de 4 kanter i facen.

Er alle edge-face test negative, er det nødvendigt at udføre tests der afgør om A er helt indeholdt i B eller omvendt.

At teste om et punkt i B er indeholdt i A er ligetil, da centrum af B er givet direkte i A 's koordinatsystem som translationsdelen af den homogene transformation $(\mathbf{F}^A)^T \mathbf{F}^B$ der er input til algoritmen. Et simpelt range-tjek er derfor tilstrækkeligt.

For test af et punkt fra A indeholdt i B er det nødvendigt at beregne den inverse transformation. Da vi anvender centerpunktet fra A , der er rotationsinvariant kan vi nøjes med at beregne den inverse translation $\mathbf{trans}_{BA} = -\mathbf{trans}_{AB}$.

Den anvendte repræsentation af en OBB baseret på en homogen transformation og boksens dimensioner, betyder at hjørnerne ikke er eksplicit tilgængelige. For få fat i de 8 hjørner fra hver OBB kræves der en del arbejde. Hjørnerne dannes først ud fra halv-dimensionerne og transformeres så til de rigtige world-koordinater. De 12 edges og 6 faces fra hver OBB udtrykkes så som pointere til hjørnerne. Dette resulterer i et højt hukommelsesforbrug på stacken på ca. 672 bytes, for data alene. I implementationen af denne algoritme har der ikke været fokus på at optimere, da den grundlæggende er for langsom til at kunne anvendes i praksis. Ikke desto mindre har den været nyttig som reference algoritme under udviklingsprocessen og har hjulpet med at finde en fejl i Gottschalk algoritmen.

⁵Samme transformation der bruges for at tage et punkt i OBB koordinater og beregne dets placering i world koordinater

9.2.3 Gottschalk

Implementationen af Gottschalk-algoritmen som beskrevet i afsnit 4.7.2 udfører 15 separating-axis tests for at afgøre om to OBB'er er i kollision eller ej. Som beskrevet i teori-afsnittet er den optimerede separating axis test delt op i de tre tilfælde af kandidatakser. Disse er

1. Kandidataksen er en face-normal fra boks A
2. Kandidataksen er en face-normal fra boks B
3. Kandidataksen er vinkelret til en edge fra hver boks

Det er bedst at udføre testene i nævnte rækkefølge. Først og fremmest skyldes dette, at type 1 og 2 rumligt dækker store områder for hver enkelt test. Der er derfor stor chance for at de første 2×3 tests vil kunne finde en separerende akse og testen kan derfor terminere hurtigt. Derudover er det billigere at udføre tests af type 1, da disse er ganske simple.

Hvis der ikke er blevet fundet en separerende akse skal de resterende tests af type 3 udføres. En interessant variation af algoritme i forbindelse med hierarkiske metoder, er at udelade de sidste 9 tests for at opnå hurtigere køretid. Dette gør algoritmen mere konservativ, idet at den oftere vil melde kollision. Kolliderende OBB'er vil i den hierarkiske metode i sidste ende føre til en afgørende test af trekkanterne. Prisen ved at udføre flere trekant tests skal i så fald holdes op imod det der spares på at have en billigere OBB test.

Gottschalk-algoritmen er brugt i flere forskellige hierarkiske kollisionsdetektionssystemer, bl.a. *PQP* og *Opcode*. Der er derfor en flere implementationer tilgængelige, bl.a. i [6] og [9].

Sidstnævnte implementation blev valgt som udgangspunkt, da den bedre afspejler strukturen af de testede kandidatakser og er udtrykt på overskuelig vis med kun ca. 40 liniers kode.

Dette er gjort ved at, lave en `define` for hvert tilfælde af kandidatakse. Disse indsættes så med de nødvendige kombinationer i selve testen. Der er 3 tests af type 1, 3 tests af type 2 og 9 tests af type 3. [9] indeholdt imidlertid en fejl i kodeeksemplet for type 2, hvor der var byttet rundt på kolonner og rækker for udregningen af r_B . Dette blev opdaget under udviklingsforløbet, da resultatet af Gottschalk algoritmen skulle holdes op imod exhaustive edge-face algoritmen. Den rettede implementation er tilgængelig i kildekoden.

9.2.4 Massive OBB tests på CELL

Implementationen af de massive OBB tests, er baseret på ALF. Der er ikke implementeret en version med Libspe. Det ville dog være en simpel sag at udvide den eksisterende Libspe kode til trekant tests, til også at omfatte OBB tests. Som tidligere nævnt er ALF implementationen lavet så den er så fleksibel som muligt. Dette har betydet at implementationen af OBB tests har være mere eller mindre lige ud af landevejen. Den største forskel ligger faktisk i at der på SPU siden kaldes en OBB test funktion, istedet for en trekant test funktion. Ellers er det den samme kode der er benyttet til begge typer af tests.

9.2.5 Massive OBB test på CUDA

Massiv OBB test på CUDA platformen udføres på stort set samme måde som den massive trekant test. Derfor vil denne ikke blive beskrevet yderligere her.

9.3 Test

9.3.1 Generelt

Korrekthed af trekant tests

For at teste korrektheden af de udviklede trekant-trekant kollisions test algoritmer, er der blevet udviklet en hel række af forskellige tests.

I den første test loaded 100.000 trekant par fra en fil. Til hver trekant par er der tilknyttet en værdi der fortæller om trekanten er i kollision eller ej. Hver algoritmer køres igennem med disse trekanten og resultaterne sammenlignes med reference værdien. Test trekanten er genereret med Mathematica, der også er brugt til at beregne reference værdien. Alle algoritmer består denne test uden problemer.

Den anden test består i at teste 5000 trekant par der ikke er i kollision. Alle algoritmer tests med disse trekanten, og alle algoritmer består denne test uden problemer.

Den tredje test består i at alle algoritmer teste mod 5000 trekant par der alle er i kollision. Denne test består alle algoritmerne også uden problemer.

Den fjerde tests er opbygget af en række trekanten der er koplanare. Disse trekanten er håndlavede og kollisions resultatet er noteret for en række kombinationer af trekanten. Kombinationerne er lavet således de forskellige koplanare grænsetilfælde er dækket (f.eks hjørne mod hjørne, kant mod hjørne osv.). De enkelte algoritmer holdes op imod disse, for at teste om de håndtere de forskellige koplanare tilfælde korrekt. Alle algoritmerne består uden problemer.

Den femte test ligner den fjerde test. Denne retter sig blot mod trekanten der ikke er koplanare. Igen er der håndlave et række trekanten og kombinationer af disse, for at dække de forskellige grænse tilfælde i 3D. Igen er der ingen af algoritmerne der fejler.

Den sjette test består i at autogenerere 100 millioner tilfældige trekant par. Hver par gives til alle tre algoritmer, der så stemmer om resultatet. Hvis de alle tre er enige antages resultatet at være korrekt. Hvis en derimod er uenige med de andre, udskrives de enkelte algoritmers resultater sammen med trekant parret. Ved denne test er der altid omkring 10 tests algoritmerne ikke er enige om. Når algoritmerne ikke er enige er det pga. numeriske usikkerheder. Da algoritmerne ikke fungere på samme måde, vil denne usikkerhed komme til udtryk i forskellige tilfælde, og derved bliver de uenige om nogle få af de mange tests.

Grunden til de mange forskellige test er at der under testen af algoritmerne blev brugt et program kaldet Geomview til at visualisere trekanten. Problemet med dette var at dette program som standard normalisere alle koordinater til at ligge mellem nul og en. Dette resulterede i at programmet ikke altid korrekt viste om to trekanten var i kollision eller ej. Dette førte naturligvis til en vis forvirring, og dermed flere og flere tests. Da fejlen endeligt blev opdaget havde test suiten nået det nuværende omfang. Desuden blev devillers algoritmen modificeret til at omfatte tolerancer.

Korrekthed af OBB tests

Korrektheden af de implementerede OBB-OBB test algoritmer, er blevet afprøvet ved hjælp af en række konstruerede testcases og visuel inspektion af tilfældigt genererede OBB'er.

Den første test består i at generere et testsæt af OBB'er, der er centreret omkring en OBB placeret i origo. Testsættet er genereret så at det gennemløber alle rotationer af et multiplum af $\pi/4$ for alle koordinataksler i alle kvadranter af rummet.

Den anden test består i at generere et stort antal af tilfældige OBB'er og lade de to implementerede algoritmer stemme om resultatet. Ved uoverensstemmelse udskrives OBB'erne i Geomview format, der så kan visuelt inspiceres.

9.3.2 CELL

I de følgende afsnit gives et overblik over de opnåede resultater på CELL platformen. Først præsenteres resultaterne for trekant testene under Libspe. Dernæst resultaterne for de samme tests under ALF. Og endeligt præsenteres resultaterne af OBB testene under ALF.

Trekant tests - Libspe

For at give et billede af algoritmernes performance er der blevet udført to tests på hver algoritme. I den første test varierer størrelsen af de to objekter, mens workunit størrelsen holdes konstant på 8192 bytes.

I tabel 9.1 ses resultaterne for devillers algoritmen. I den første test er der kun $128 \times 128 / 8192 = 2$ SPE'er i gang, og processoren udnyttes derfor ikke optimalt. I den næste er der 8 workunits, og alle SPE'erne arbejder derfor, dette ses tydeligt på performance. I de næst følgende tests ses det at performance ikke stiger nævneværdigt, hvilket er som forventet, idet der ikke er flere ledige SPE'er at udnytte.

Objektstørrelse	Millioner tests/sekund
128×128	6.370140
256×256	10.812737
512×512	11.100271
1024×1024	11.131853

Tabel 9.1: Antal tests/sekund for devillers algoritmen ved varierende objektstørrelser.

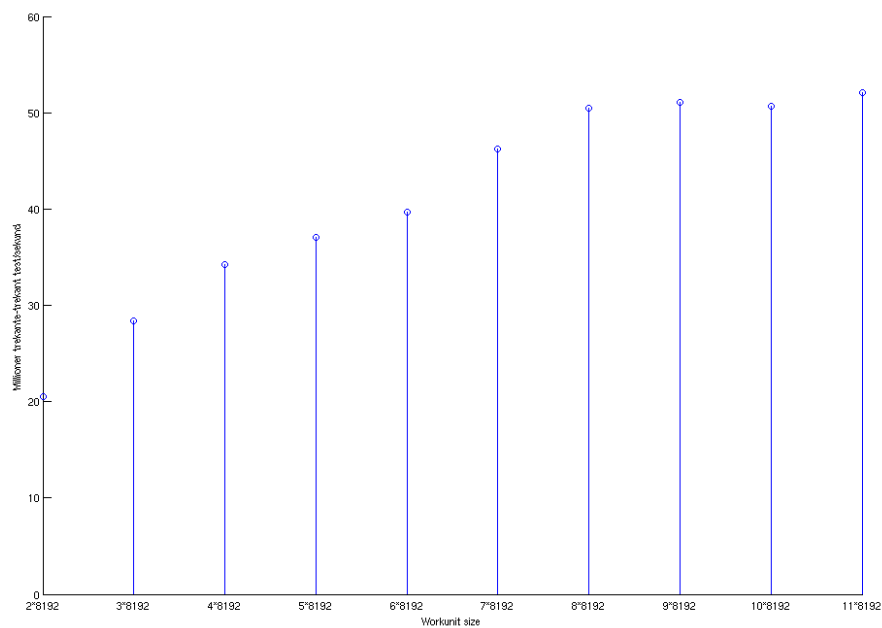
Stort set samme mønster gentager sig for segment-piercing algoritmen, som det ses i tabel 9.2

Objektstørrelse	Millioner tests/sekund
128×128	2.784500
256×256	6.486144
512×512	8.972003
1024×1024	9.298277

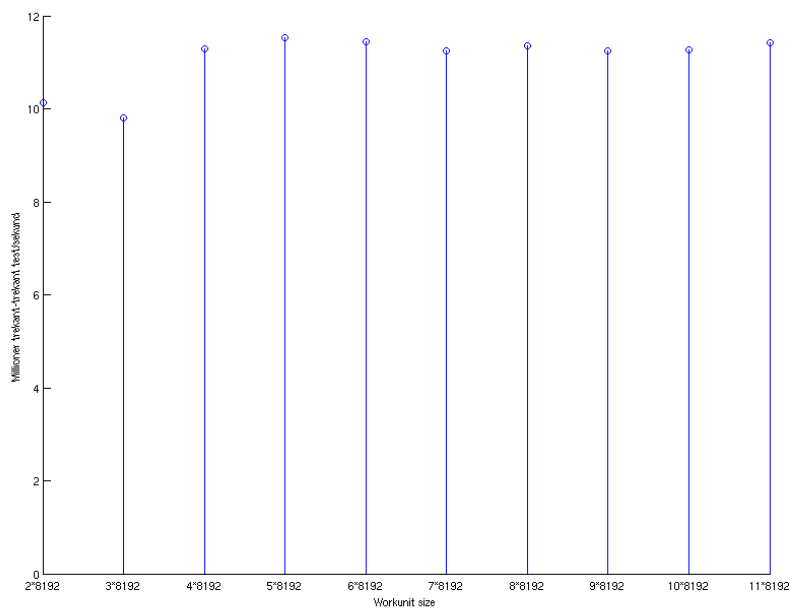
Tabel 9.2: Antal tests/sekund for segment-piercing algoritmen ved varierende objektstørrelser.

I den næste test der er blevet udført på algoritmerne, holdes objektstørrelserne konstante på 2048×2048 , mens workunit størrelsen ændres i spring på 8192 bytes.

På figur 9.13 ses resultatet for devillers algoritmen. Det ses at workunit størrelsen har afgørende betydning for performance. Performance stiger jævnt med workunit størrelsen, hvilket da også er forventeligt, idet det betyder at SPE'erne får en større andel af "regnetid", i forhold til den tid der bruges på data overførelser. Ved en workunit størrelse på 11×8192 bytes er der ikke længere plads til data i SPE'ernes lokale hukommelse. Performance ser ud til at flade ud ved omkring 52 millioner tests/sekund. Dette hænger utvivlsomt sammen med selve algoritmen og implementationen af denne. Figur 9.14 viser resultatet for den samme test med segment-piercing algoritmen. Igen ses det samme billede, dog flader performance noget hurtigere ud, end ved devillers algoritmen. Performance er endvidere noget lavere, devillers er ca. 4.5 gange hurtigere i peak performance. Det er forventeligt at devillers algoritmen har en performance fordel, og i modsætning til på GPU'en manifestere denne performance fordel sig på Cell processoren. Dette skyldes givetvis at Cell processoren er noget bedre til at håndtere branches end GPU'en.



Figur 9.13: Antal tests/sekund for devillers algoritmen ved varierende workunit størrelse.



Figur 9.14: Antal tests/sekund for segment-piercing algoritmen ved varierende workunit størrelse.

Trekant tests - ALF

Det har ikke været muligt at opnå nogle brugbare målinger af trekant testene implementeret med ALF. Dette skyldes at det har været svært at finde ud hvornår ALF begynder på beregningerne. I dokumentationen nævnes der at funktionen `alf_task_finalize` skal kaldes efter at den sidste workblock er tilføjet. Desuden nævnes det at en opgave ikke kan der ikke er „finalized“ ikke kan køres færdig. Efter dette kald skal funktionen `alf_task_wait` kaldes, denne funktion blokerer indtil tasken er færdig. Udgangspunktet for målingerne var derfor at måle hvor lang tid disse funktioner tog. Dette viste sig dog at give nogle højest overraskede resultater. Faktisk var kaldende så hurtigt at det svarede til flere milliarder trekant tests i sekundet! Flere eksperimenter bekræftede dette. Den bedste forklaring på dette må være at ALF starter processeringen tidligere end antaget. Det blev derfor målt hvor lang tid der gik fra den første workblock blev tilføjet til tasken var færdig. Men her gik resultaterne i den anden grøft. Her var der tale om resultater der svarede til nogle få millioner trekant tests i sekundet. Noget kunne altså tyde på at ALF starter med at udføre tasken på et tidspunkt midt imellem. En mulighed for at finde ud af hvad der forgår er at studere de events som ALF forårsager. Der er skrevet kode til at udskrive disse events, men der er ikke gjort noget for at sætte disse ind i et tidsmæssigt perspektiv.

OBB tests - ALF

De samme forhold der gør sig gældende for trekants testene under ALF, gør sig også gældende for OBB testene under ALF. Der forligger således ingen meningsfulde data for OBB testene under ALF.

OBB tests - CUDA

Tabel 9.3 viser ydelsen af Gottschalk-algoritmen ved kørsel på GPU'en. Generelt fås en ydelse der er ca. det dobbelte af Segment-piercing algoritmen. Dette stemmer fint overens med at antal basale operationer er det halve for denne algoritme.

Objektstørrelse	Millioner tests/sekund
1318 × 1318	160
1757 × 1757	162
2635 × 2635	165
5271 × 5271	163
10542 × 10542	164

Tabel 9.3: Antal tests/sekund for Gottschalk algoritmen på CUDA.

9.4 Delkonklusion

Under arbejdet med Cell processoren er det blevet klart at der er tale om en særdeles potent arkitektur. Dette ses også af de opnåede resultater, der på trods af mangel på optimeringer, stadig er imponerende. Der skal dog ikke herske nogen tvivl om at processoren langt fra er udnyttet optimalt, med den nuværende kode. Der er plads til mange forbedringer.

Endeligt at skal det siges at det har vist sig at være arbejdskrævende at udvikle til Cell processoren, idet den kræver meget manuel håndtering af hukommelsen. På andre områder kræves der ligeledes meget manuelt arbejde, dette gælder f.eks. styringen af de enkelte processor elementer. Alt i alt er det

en meget anderledes processor at arbejde med, og man skal forvente at det tager en del mere tid at få tingene til at virke, end man er vant til.

Kapitel 10

Implementation af hierarkisk metode

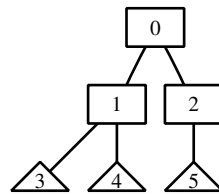
10.1 Partitionering

10.1.1 Generelt

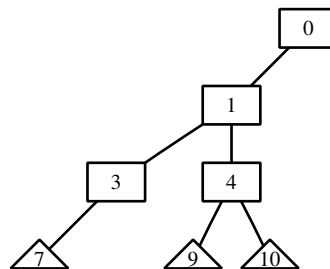
Opgaven for den hierarkiske metode er at teste to objekt træer mod hinanden. Noderne i træerne kan være bounding volumes eller trekanter, som illustreret på figur 10.1, 10.2 og 10.3.

Fastlægges traverseringen og nedstigningen (som beskrevet i afsnit 4.3.2), fås der et træ af alle kombinationer af kollisionstest imellem noderne (Collision Test Tree, CTT). For en samtidig, bredde-først nedstigning af de viste træer A og B fås der et træ som vist på figur 10.4.

En node (x,y) i test træet svarer til en kollisionstest mellem node x fra objekt træ A imod node y fra B. Nummereringen af noder i test træet på figur 10.4, er indekserne til noderne i de fulde, balancerede objekt træer, nummereret bredde-først. Hvorfor dette er nødvendigt vil senere blive forklaret.

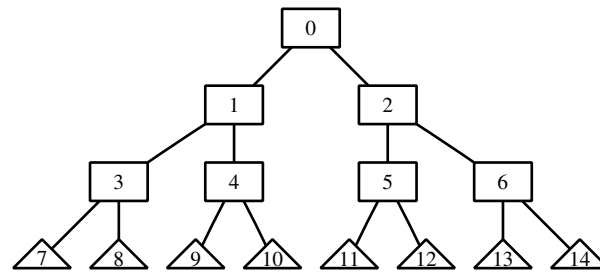


Figur 10.1: Objekt træ A.



Figur 10.2: Objekt træ B.

Kollisions test træet kan generaliseres til at omfatte hele scenen som vist på figur 10.5. Dette giver en række fordele mht. parallelisering som senere vil blive beskrevet.



Figur 10.3: Objekt træ C.

Oprindeligt var det tænkt at opbygningen af hele træet kunne udføres som en præprocessering. Træet kunne så placeres i hukommelsen og beregningsenhederne kunne efterfølgende så 'fylde træet ud' med kollisionstatus for en given node. Imidlertid vil test træet kræve enormt meget plads for scener af selv moderat kompleksitet. Som eksempel vil indbyrdes test af 11 binære objekt træer af 8000 noder hver kræve et test træ med plads til $8000^2 \cdot 11^2 = 7.74 \cdot 10^9$ noder. Antages en node til at fylde kun en byte, vil pladsforbruget langt overstige hvad der er til rådighed på platformene.

Ved at repræsentere objekt træerne så direkte indeksering til en node er mulig, kan dette dog gøres ret effektivt runtime. Direkte indeksering er mulig, da objekttræet er repræsenteret implicit. For et binært træ vil henholdsvis venstre og højre barn til en node på position i i arrayet, kunne findes på position $2i + 1$ og $2i + 2$, illustreret på figur 10.6. Ligeledes kan parent til en node findes som $\lfloor \frac{i-1}{2} \rfloor$. Dette virker også for mere generelle træer.

Den implicite repræsentation af et binært træ har et hukommelsesforbrug svarende til 2^h hvor h er højden af træet. For et perfekt balanceret, fuldt træ er dette optimalt, men det kan imidlertid blive meget dyrt for høje, ubalancerede træer. Hver eneste gang der tages et nyt lag i træet i brug, fordobles hukommelsesforbruget.

Et generelt objekt træ garanterer ikke at OBB'erne og trekantene er afgrænset så pænt i træet som vist på forrige figurer. Trekantene der udgør leaf-noder er ikke nemlig ikke begrænset til at kunne optræde på det sidste niveau i træet.

Det er derfor nødvendigt at for en given node i den implicite repræsentation, at have oplysninger om hvilken type node det er. Dette er gemt i en `it_node_t` som beskrevet herunder

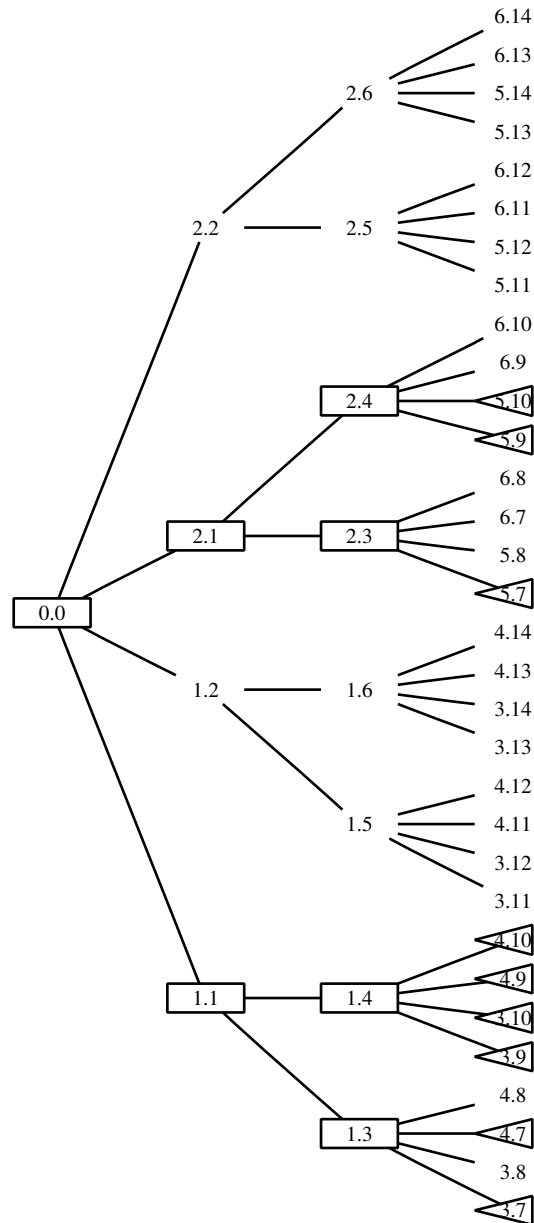
```

1 |-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----|
2 | is_leaf | type | b14 | b13 | b12 | b11 | b10 | b9 | b8 | b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |
3 |-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----|

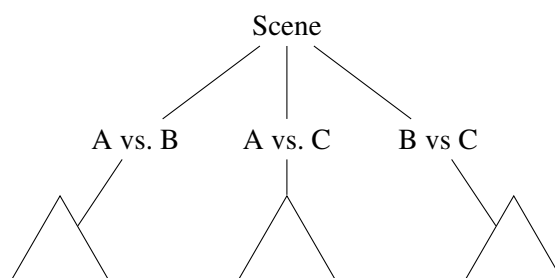
```

Den implicite repræsentation af et objekt træ er et array af 16 bit tal. De 16 bit er delt op i et et indeks $b_0 - b_{14}$ og to flag. `is_leaf` er 1 når en OBB er en leaf node (dens børn er trekanter) ellers 0. `type` er 1 hvis noden er en OBB, 0 hvis det er en trekant. Indekset er, afhængigt af nodens type et offset til en trekant eller en OBB ind i et array. Med 14 bit afsat til indekset er det maksimale antal OBB'er eller trekanter i et enkelt objekt træ dermed $2^{14} = 16384$.

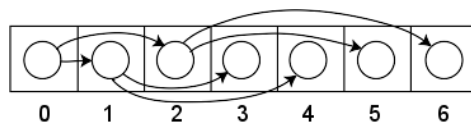
Følgende struct definerer et objekt træ. Det indeholder det implicite træ af `it_node_t`'s samt dets størrelse og selve pointere til OBB og trekant data indeholdt i træet. OBB'erne og trekantene er wrappet i de samme `obb_soup` og `triangle_soup` datastrukturer som massiv trekant-trekant/OBB-OBB tests (beskrevet i 8.3.2).



Figur 10.4: Kollisions test træ (CTT) for objekt træ A og B. Første indeks er ind i træ A, andet indeks er B.



Figur 10.5: Kollisions test træ (CTT) for en hel scene, bestående af A, B og C



Figur 10.6: Implicit representation af et binært træ i et array (Kilde: http://en.wikipedia.org/wiki/Binary_tree)

```

1 struct object_tree {
2     it_node_t *it; /**< implicit representation of the tree */
3     int it_nnodes; /**< Number of implicit tree nodes */
4     struct obb_soup *os; /**< Soup of OBB's covering the triangles in the object
5         */
6     struct triangle_soup *ts; /**< Soup of triangle that make up the object */
7 };

```

Kollisions test træet (CTT) er, på trods af at det ikke lader sig ligge fuld repræsenteret i hukommelsen, stadig en meget nyttig abstraktion. En `ctt_node` repræsenterer en enkelt test imellem to noder fra hver sit objekt træ i en scene.

```

1 struct ctt_node {
2     unsigned short objt1; /**< tree 1, index into array of object trees */
3     unsigned short objt2; /**< tree 2, index into array of object trees */
4     unsigned short n1; /**< node of tree 1, index into it_node_t array
5         */
6     unsigned short n2; /**< node of tree 2, index into it_node_t array
7         */
8     unsigned short obj1_idx; /**< the optionally resolved index of (tri/obb)
9         object 1 */
10    unsigned short obj2_idx; /**< the optionally resolved index of (tri/obb)
11        object 2 */
12 };

```

Indekserne `objt1` og `objt2` identificerer de to objekt træer. `n1` og `n2` er indekser ind i de implicitte træ repræsentationer.

Givet en CTT node, kan noderne fra henholdsvis objekt træ 1 og 2 tilgås som henholdsvis `objts[objt1].it[n1]` og `objts[objt2].it[n2]`, hvor `objts` er arrayet af alle objekt træer indeholdt i scenen. De to sidste members af `ctt_node`'en bruges til at cache indekset af OBB'en/trekanten som de implicitte træ noder peger på. CTT noder letter i høj grad paralleliseringen af arbejdet i en scene, da paralleliteten i scenen og i de enkelte træer håndteres under et.

```

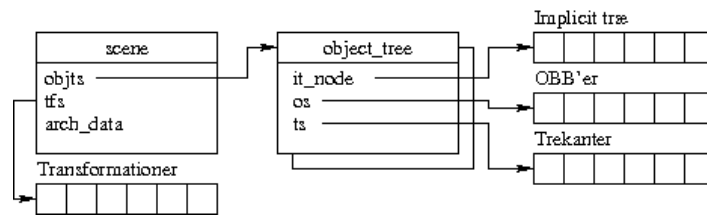
1 struct scene {
2     struct object_tree *objts; /**< array of object trees */
3     struct transformation_matrix *tfs; /**< array of transformations, one
4         transformation for each object tree */
5     int n_objts; /**< Number of objects in the scene */
6     void *p_data; /**< Pointer to private data */
7     void *arch_data; /**< Pointer to architecture specific data */
8     struct ctt_node tps[TREEPAIRS_MAX]; /**< array of tree pairs to be tested
9         for collision */
10    int n_tps; /**< number of tree pairs in the array */
11 };

```

Scenen indeholder alle `n_objts` objekt træer `objts` og deres tilhørende homogene transformationer `tfs`. For et objekt træ `objts[i]` er den tilhørende transformation `tfs[i]`. CTT noderne der definerer hvilke objekt træer der skal testes i scenen, er gemt i arrayet `tps`. Scenens relation til objekt træerne kan ses på figur 10.7.

Derudover er der to felter `p_data` og `arch_data` som benyttes til platformsspecifikke data.

Da en CTT node er en selvstændig enhed, og på grund af den implicitte repræsentation af objekt



Figur 10.7: En scene

træerne er det muligt givet kun en CTT node, at bestemme de nedarvede kollisions test og hvilke objekt træer disse børn refererer til.

For alle platformene bruges der en fælles funktion, `get_children` til at få fat i børnene til en CTT node. Afhængigt af om børnene eksisterer og deres type, sørger funktionen for at pushe til de nødvendige stacks.

Transformationer

Objekt træerne er udtrykt som homogene transformationer. For et objekt træ A er dens tilhørende transformation \mathbf{F}^{t_A} , den transformation der skal til for at flytte world-ramen til at være aligned med objekt træets frame.

Punkter i trekantssuppen hørende til et objekt træ er udtrykt i forhold til træets koordinatsystem. For at teste to trekanter fra forskellige objekt træer er det derfor nødvendigt at beregne punkternes placering i world koordinat. Dette kræver 6 koordinatstransformationer.

OBB'er i et objekt træ er beskrevet som den transformation der skal til for at flytte objekt træets frame til at være aligned med OBB'en. Da den anvendte Gottschalk algoritme kræver at B udtrykkes i forhold til A , skal vi som beskrevet anvende $(\mathbf{F}^A)^T \mathbf{F}^B$. Imidlertid skal objekt træernes transformationer også tages med, så i stedet fås $(\mathbf{F}^{t_A} \mathbf{F}^A)^T (\mathbf{F}^{t_B} \mathbf{F}^B)$, hvor t_A og t_B er træerne indeholdende henholdsvis OBB A og B .

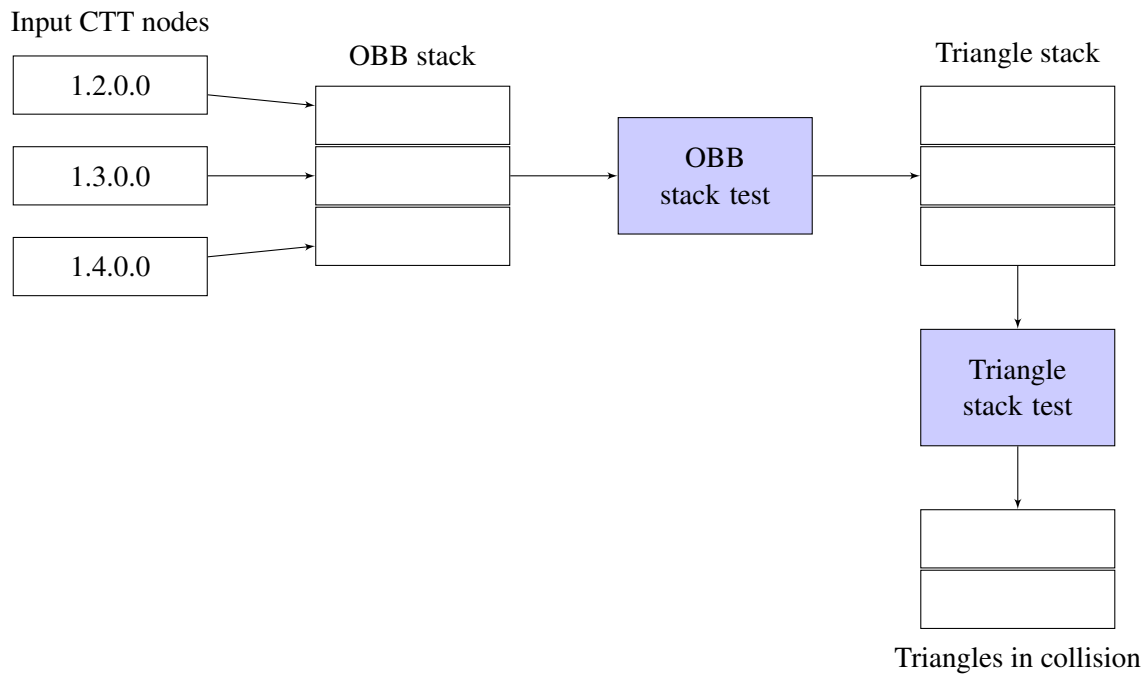
10.1.2 GPU

Overordnet følger implementationen af den hierarkiske metode på CUDA flowet som vist på figur 10.8.

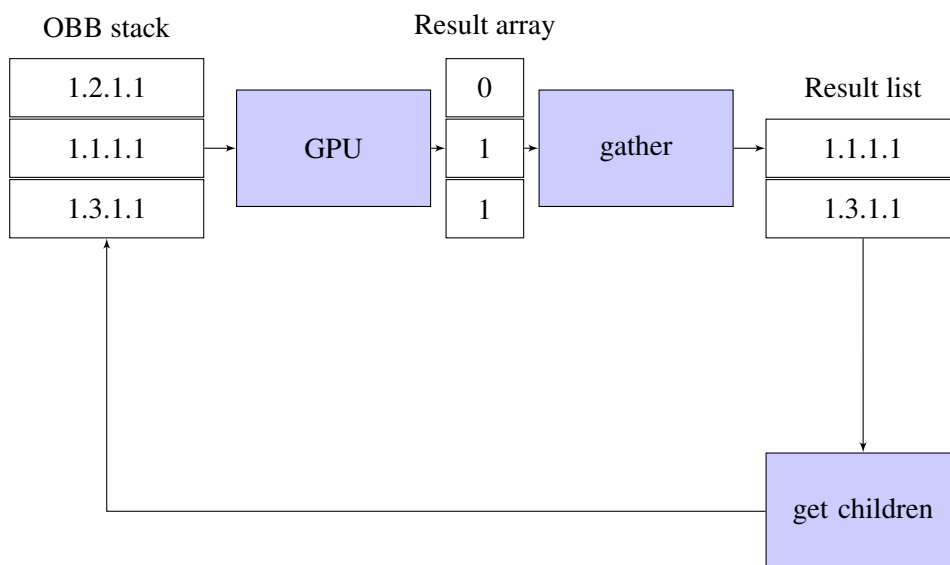
CTT noderne lægges på en stack der gives som input til en OBB stack test. Denne test udfører kollisions test på input og tilføjer ekstra CTT noder til input stacken. Når der ikke er flere noder tilbage i OBB stacken, går der videre til at teste de opakkumulerede trekanter.

Testen af OBB stacken foregår som vist på figur 10.9. En stack af OBB'er, repræsenteret som CTT noder skal testes for kollision. Disse gives til GPU'en, der udfører OBB kollisions test (mere herom senere).

Resultatet af kollisions test fra GPU'en er et array af positive og negative resultater (0 og 1). Før at GPU'en giver resultatarrayet videre til `gather`, udføres der en optimering, der gør det lettere at finde de positive resultater. Optimeringen foregår ved at en CUDA kerne, `cuda_result_list_optimize` deler resultatarrayet op i flere delområder, og erstatter alle pladser hvor der står nul. I stedet for nul skrives der en negativ værdi der fortæller hvor langt der skal springes for at komme til det næste positive resultat. For at undgå for meget serialitet i kernen, er dog en øvre begrænsning på, hvor langt der kan springes fra en enkelt position.



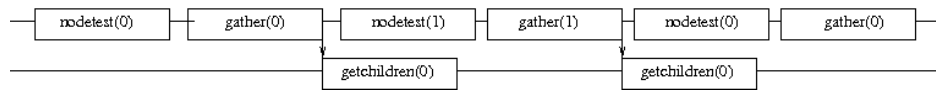
Figur 10.8: Overordnet flowdiagram for CTT noder i CUDA model testen.



Figur 10.9: Flowdiagram for test af OBB noder i CUDA model testen.

Oprindeligt var det tænkt at GPU'en selv skulle stå for at foretage `get_children` og `gather` og dermed undgå at data skulle sendes frem og tilbage imellem CPU'en. Dette har imidlertid ikke været muligt, da det anvendte grafikkort i projektet ikke har understøttet atomisk læsning/skrivning i hovedhukommelsen. Uden denne funktionalitet er implementationen af delte datastruktur udelukket. `get_children` og `gather` kører derfor på CPU'en.

Disse to operationer udføres begge serielt, men ved at indføre en opdeling af input OBB stacken er det muligt at fordele arbejde på flere tråde som vist på figur 10.10.



Figur 10.10: Fordeling af `get_children` og `gather` på to tråde

Ved at dele OBB stacken op i flere mindre dele er det muligt at overlappende testen af noder/`gather` med `get_children`.

En samling af CTT noder der skal testes for kollisioner placeres i en `test_list` struct.

```

1 /* list of ctt_nodes to be tested for collision */
2 struct test_list {
3     struct ctt_node *tests;
4     int n_tests;
5 };
  
```

Noder rapporteret til at være i kollision placeres i en `result_list`.

```

1 /* list of ctt_nodes reported to be in collision */
2 struct result_list {
3     struct ctt_node *results;
4     int n_results;
5 };
  
```

Synkroniseringen af de to tråde er implementeret standard vha. semaforer. Dette gør det lettere at styre samarbejdet, og gør det samtidigt muligt for en tråd at frigive CPU'en hvis at der ikke er input data eller output buffere til rådighed. En tråd der forespørger en ikke endnu ledig ressource bliver lagt til at sove af operativsystemet.

Når systemet initialiseres er der `N_LISTS` frie lister til rådighed i henholdsvis `free_tl_q` og `free_rl_q`. Når en node skal testes for kollision, hentes en test liste fra køen af fri test lister. Noden placeres i listen og lægges derefter i køen af parate lister. Tilsvarende, når en node er bestemt til at være i kollision skal den behandles af `get_children`. Dette gøres ved at placere den kolliderende node i køen af parate resultat lister.

De forskellige køer er samlet i en global `list_queues` struct.

```

1 /* central queue of test and result lists */
2 struct list_queues {
3     struct test_list *ready_tl_q[N_LISTS]; /* queue of ready test lists with
4     nodes to be tested for collision */
5     struct test_list *free_tl_q[N_LISTS]; /* queue of free test lists with
6     room for storing new nodes to be tested */
7     struct result_list *ready_rl_q[N_LISTS]; /* queue of ready result lists
8     with nodes in collisions */
9     struct result_list *free_rl_q[N_LISTS]; /* queue of free result lists
10    with room for storing new nodes in collision */
11    sem_t ready_tl_sem;
  
```

```

8   sem_t free_tl_sem;
9   sem_t ready_rl_sem;
10  sem_t free_rl_sem;
11  int n_ready_tl;
12  int n_free_tl;
13  int n_ready_rl;
14  int n_free_rl;
15  pthread_mutex_t lock;
16 };

```

Da ovenstående datastruktur er delt imellem trådene er det nødvendigt at låse når den tilgås. Dette håndteres af en mutex.

Til hvert enkelt objekt træ er der, som beskrevet i foregående afsnit 10.1.1 en tilhørende homogen transformation. Denne skal kunne tilgås på grafikortet af OBB-OBB og trekant-trekant test kernerne. Dette håndteres af en `cuda_context` struct, der gives til hver enkelt kerne kørsel

```

1  struct cuda_context {
2      transformation_matrix *tfs;
3      obb_t **ost;
4      triangle_t **tst;
5  };

```

Denne indeholder en pointer til et array af homogene transformationer, allokeret i grafikortets hukommelse. Transformationen for et objekt træ af indeks i kan tilgås som `tfs[i]`. Da transformationerne skal kunne ændres under kørsel, er der funktioner til at synkronisere transformationerne med dem der defineres i computerens hukommelse.

I samme struct er også to dobbelt pointerne `ost` og `tst`. Disse peger på tabeller indeholdende henholdsvis OBB- og trekantsupper. Denne opbygning skyldes at supperne kan være af variabel længde og derfor må alloc'es dynamisk. Det første element i f.eks. OBB-suppen for et objekt træ af indeks i kan tilgås som `*(ost[i])`.

Følgende kodeblok viser hvordan at OBB kernen til test af CTT noder er implementeret. Den nuværende tråd, identificeret ved indekset `gi` loader en CTT node og bestemmer ud fra dens oplysninger, hvilke OBB'er fra hvilke træer der skal testes for kollision. Resultatet af kollisionstesten skrives til arrayet `r` i grafikortets hukommelse.

```

1  __global__ void cuda_ctt_node_obbkernel(struct cuda_context cc, struct ctt_node *
2      tl, opp_result_t *r, int length) {
3      const unsigned int gi = (blockIdx.x * blockDim.x) + threadIdx.x;
4      if (gi >= length)
5          return;
6      struct ctt_node n = tl[gi];
7      obb_t *const os1 = cc.ost[n.objt1]; /* pointer to obb soup for object
8          tree 1 */
9      obb_t *const os2 = cc.ost[n.objt2]; /* pointer to obb soup for object
10         tree 2 */
11      obb_t *const obb1p = os1 + n.obj1_idx; /* could be optimized away */
12      obb_t *const obb2p = os2 + n.obj2_idx; /* could be optimized away */
13
14      /* the subsequent will hopefully start an early load of the data */
15      const struct transformation_matrix tf1 = cc.tfs[n.objt1];
16      const struct transformation_matrix tf2 = cc.tfs[n.objt2];
17
18      const obb_t obb1 = *obb1p;
19      const obb_t obb2 = *obb2p;

```

```

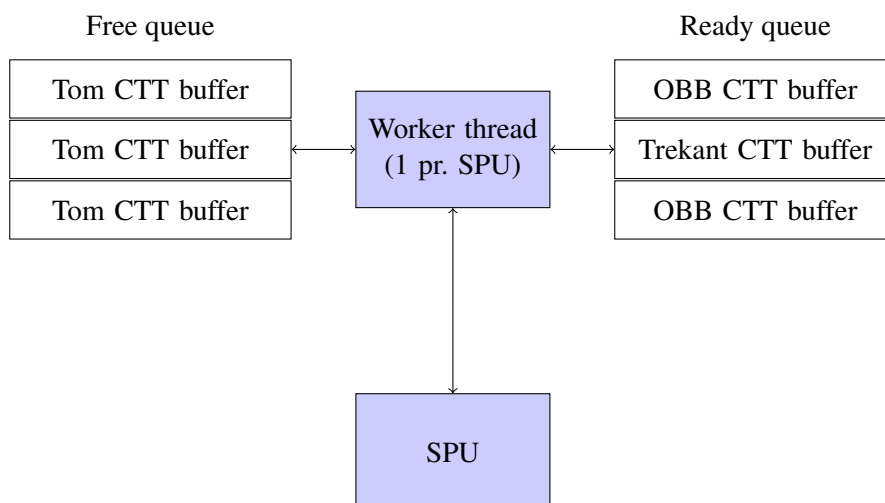
17
18  /*
19   * transformations and collision check
20   * -> write result to r[gi]
21   */
22 }

```

Trekant-kernen til test af CTT noder er meget lig ovenstående, og der henvises derfor til kildekoden.

10.1.3 CELL/BE

Implementationen af den hierarkiske metode på CELL platformen er beskrevet i det følgende afsnit. Metoden er baseret på den i de foregående afsnit beskrevne partitionering. Grundtanken i implementationen er at lade PPU'en være „orkersterleder“, og lade SPU'erne være slave processorer der styres af PPU'en. PPU'en foretager med andre ord partitioneringen af problemet og lader SPU'erne foretage de egentlige kollisions test, enten mellem OBB'er eller mellem trekanter. En kollisions test repræsenteres, som beskrevet ovenfor, af en CTT node, der beskriver hvilke primitiver der skal testes mod hinanden, samt hvilke objekter disse tilhører. Resultatet fra en mængde kollisionstests er en række CTT noder, der beskriver de primitiver der ret faktisk var i kollision af de testede noder. Figur 10.11 viser det overordnede flow i implementationen.



Figur 10.11: Overblik over CELL implementationen af den hierarkiske metode.

Når programmet startes, køres et program på hver SPU. Dette program venter på kommandoer fra PPU'en. Der er tre forskellige kommandoer: udfør kollisions test mellem trekanter, udfør kollisions test mellem OBB'er og luk SPU programmet ned. Hver SPU har sin egen PPU tråd¹ som det bruger til at kommunikere med PPU'en. På PPU'en er det et kø-system af buffere. En buffer beskriver input til en SPU, og kan befinde sig i en af tre tilstande. Hvis bufferen ikke er i brug, befinder den sig i en „free queue“, hvorfra den kan tages når der er brug for en ledig buffer. Den kan også være i en „ready queue“. Buffere i denne kø indeholder tests (CTT noder) og er klar til at blive overført til en SPU, så testene kan udføres. Den tredje og sidste tilstand er når testene i en buffer bliver udført af en SPU, i dette tilfælde er bufferen knyttet til den udførende SPU.

¹Faktisk har den to: En arbejdsstråd som beskrevet her, og en „run“ tråd der holder styr på det kørende SPU program.

Flowet i selve kollisions testen kan beskrives ved følgende pseudokode

```

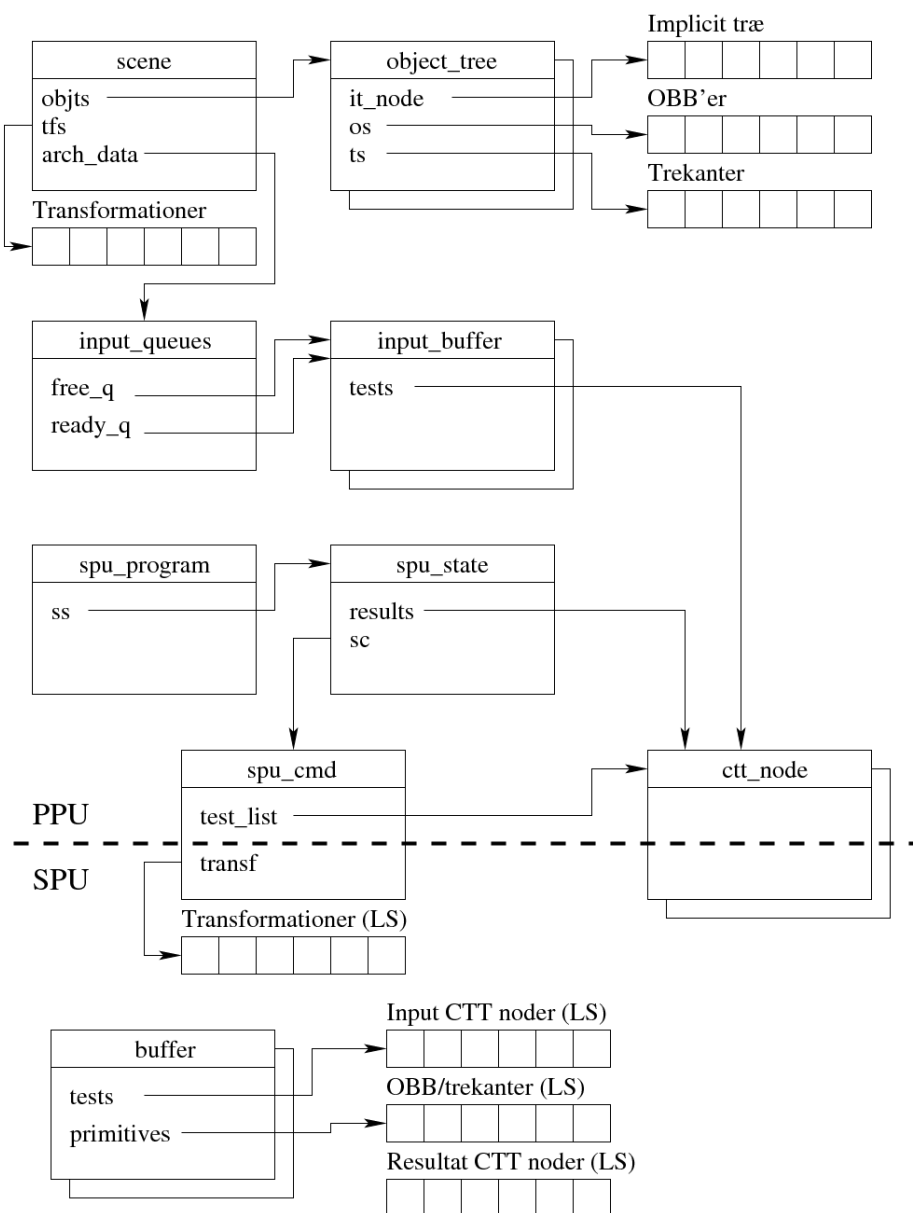
1  while (not shutdown) {
2    buf = get_ready_buffer() // Blocks until buffer is ready
3    signal_spu(buf) // Start test on SPU
4    result = wait_for_spu() // Blocks until SPU is done
5    if (SPU was running OBB tests) { // We are dealing with results of OBB tests
6      triangle_buf = get_free_buf() // New buffer for triangle tests
7      obb_buf = get_free_buf() // New buffer for OBB tests
8      cur_ctt = get_first_result(result)
9      while (more_results(result)) { // Create new tests from result
10       ctt_node_getchildren(cur_ctt)
11       cur_ctt = get_next_result(result, obb_buf, triangle_buf) // Fill in
           children of CTT's in collision
12       if (buffer_full(obb_buf)) { // We need more room for OBB's
13         put_ready_buffer(obb_buf)
14         obb_buf = get_free_buffer()
15       }
16       if (buffer_full(triangle_buf)) { // We need more room for triangles
17         put_ready_buffer(triangle_buf)
18         triangle_buf = get_free_buffer()
19       }
20     }
21     if (not_empty(obb_buf))
22       put_ready_buffer(obb_buf)
23     else
24       put_free_buffer(obb_buffer)
25     if (not_empty(triangle_buf))
26       put_ready_buffer(triangle_buf)
27     else
28       put_free_buffer(triangle_buffer)
29   }
30   else { // We are dealing with results of triangle tests, ie. real collisions
31     copy_results_to_final_result_buffer(results)
32   }
33   put_free_buffer(results)
34   put_free_buffer(buf)
35 }

```

Pseudokoden beskriver det arbejde der foregår i de enkelte arbejdsstråde. Først venter tråden på at der er arbejde i form af en buffer fra ready køen. Dernæst beder den SPU'en om at udføre arbejdet. Tråden venter til SPU'en melder tilbage at den er færdig, hvorefter tråden behandler resultaterne fra testen. Hvis der var tale om en trekant test, kopieres de CTT noder der var i kollision til en global resultats buffer. Hvis der derimod var tale om resultater fra en OBB test, findes børnene til de kolliderende noder, og disse lægges enten i en ny trekant buffer eller en ny OBB buffer fra ready køen. Herefter ventes på mere arbejde.

For at få det hele til at hænge sammen, er der blevet udviklet et slags framework, baseret på en række datastrukturer. Figur 10.12 viser et overblik over hvordan disse datastrukturer hænger sammen.

De to structs scene og object_tree, vil ikke blive gennemgået her, da de er forklaret andetsteds. Structen input_queues holder styr på de ledige input buffere og de input buffere der er klar til at blive overført til en SPU. Definitionen af de to structs er gengivet herunder. Som det ses er der to semaforer til at beskytte de to køer. Den semafor der beskytter free køen initialiseres til det maksimale antal køer systemet bruger. Det vil med andre ord sige at så længe der er frie buffere vil de tråde der beder om en, få en med det samme, og semaforen vil blive talt ned. Hvis en tråd beder om en free buffer, og der



Figur 10.12: Overblik over datastrukturer i CELL implementationen.

ikke er flere, vil tråden blive lagt til at sove indtil der igen er ledige free buffere. Det omvendte gør sig gældende for ready bufferne. Deres semafor initialiseres til nul, og tråde der beder om en ready buffer, vil altså blive lagt til at sove, indtil der er en buffer klar til dem. Endeligt benyttes der en mutex til at beskytte adgangen til selve datastrukturen.

```

1 struct input_buffer_queues {
2     sem_t free_sem; /**< Semaphore used to keep track of the free buffers */
3     sem_t ready_sem; /**< Semaphore used to keep track of the ready buffers */
4     int free_idx; /**< Index pointing to the current free buffer */
5     int ready_idx; /**< Index pointing to the current ready buffer */
6     struct input_buffer *free_q[N_INPUT_BUFFERS]; /**< list of free (empty)
7         input buffers */
8     struct input_buffer *ready_q[N_INPUT_BUFFERS]; /**< list of buffers ready to
9         be processed by a SPU*/
10    pthread_mutex_t lock; /**< Mutex used to protect this struc */
11 };

```

En input buffer kan være en af to typer, enten er den en buffer der indeholder trekanter, eller også indeholder den OBB'er. Typen angives af type feltet. Structen indeholder en liste af CTT noder, der beskriver de enkelte tests der skal udføres. Definitionen af en CTT node kan ses i afsnit 10.1.1.

```

1 struct input_buffer {
2     enum BUF_TYPE type; /**< Type of buffer */
3     int n_test; /**< Number of tests in the buffer */
4     struct ctt_node *tests; /**< The buffer itself */
5 };

```

Til at holde styr på det program der afvikles på de enkelte SPU'er, benyttes structen `spu_program`, som ses herunder. Denne struct indeholder navnet på det spu image der afvikles på SPU'erne, samt et handle som Libspe returnere når programmet åbnes fra en fil. Endeligt indeholder structen en række `spu_state` structs.

```

1 struct spu_program {
2     char *image_name; /**< Image name of program */
3     spe_program_handle_t *pgm_handle; /**< Handle to opened program image */
4     struct spu_state ss[NUM_SPU]; /**< State of the SPU's running this
5         program */
6 };

```

Definitionen af `spu_state` er gengivet her. Formålet med denne struct er at holde styr på de detaljer der er knyttet til den enkelte SPU. Disse omfatter blandt andet SPU'ernes Libspe kontekst, deres arbejdsstråd og deres „run“ tråd. Derudover indeholder den også en liste af integers, der benyttes af SPU'en til at meddele resultaterne af en test. SPU'en skriver de indexer på de CTT noder der er fundet at være i kollision i dette integer array, og det er på denne måde muligt at finde ud af hvilke CTT noder der var i kollision, uden at overføre selve noderne. På samme måde indeholder strukturen også en pointer til den input buffer som den tilhørende SPU skal behandle. Endeligt er der en pointer til en `spu_cmd` struktur, der beskrives nedenfor.

```

1 struct spu_state {
2     int id; /**< SPU id */
3     spe_context_ptr_t ctx; /**< Context of this SPU */
4     spe_spu_control_area_t *psa; /**< Control area of the spu, used to
5         communicate directly from the PPU to the SPU */
6     struct spu_cmd *sc; /**< Command object, used to send commands to
7         this SPU */
8 };

```

```

6   pthread_t r_thrd;          /**< Thread used by this SPU to run the
    spu_run_context function */
7   pthread_t w_thrd;          /**< Thread used by this SPU to do
    postprocessing of the results */
8   unsigned short int *results; /**< Main memory result array for this SPU */
9   struct input_buffer *input; /**< If the SPU is processing this points to
    its input buffer */
10  struct scene *scene; /**< Backpointer to the opp scene */
11 };

```

Strukturen `spu_cmd` er bindeledet imellem PPU siden og SPU siden af programmet. Dens primære funktion er at videregive en kommando fra PPU'en til SPU'en. Denne kommando kan som tidligere nævnt være besked om at udføre en trekant, eller OBB test, eller en besked om at terminere SPU programmet. Ud over selve kommandoen indeholder structen, en række parametre der overføres til SPU'en fra PPU'en. Disse er adresserne på hhv. alle trekant supper og OBB supper, de homogene transformations matricer, adressen på input bufferen samt adressen på resultat listen i `spu_state` structen.

```

1  struct spu_cmd {
2      enum SPU_CMD cmd;          /**< Command to execute */
3      addr64_t obb_soup_ea[MAX_OBJS]; /**< Addresses of the model obb
    soups in main memory */
4      addr64_t tri_soup_ea[MAX_OBJS]; /**< Addresses of the model
    triangle soups in main memory */
5      struct transformation_matrix tfs[MAX_OBJS]; /**< Transformation matrix for
    each object */
6      addr64_t test_list;        /**< Address of the list of
    tests */
7      int test_list_size;        /**< Number of tests */
8      addr64_t results;          /**< Address of result array */
9  };

```

Adresserne der overføres fra PPU'en til SPU'erne overføres med en særlig union, som ses herunder. Denne gør det muligt at opdele en 64 bit adresse i to 32 bit halvdele. Dette er nødvendigt, da PPU'en er en 64 bit processor, mens mange af Libspe funktionerener på SPU'en benytter 32 bit adresser. Desuden overføres adressen til en `spu_cmd` fra PPU'en til SPU'erne via en mailbox, og denne kan kun overføre 32 bit ad gangen. Det skal i denne sammenhæng nævnes at i der pt. kompiles 32 bit kode til PPU'en, og denne funktionalitet derfor ikke er strengt nødvendig. Den er dog taget med alligevel for at sikre mod problemer hvis det ønskes at kompilere koden til 64 bit. Grunden til at koden kompiles til 32 bit, er at IBM anbefaler dette medmindre man har brug for mere end 4GiB hukommelse. Denne anbefaling skyldes bla. at koden bliver mindre, da pointer osv. fylder mindre. Derudover anføres det at DMA overførelser er en smule mere effektive i 32 bit mode end i 64 bit mode.

```

1  typedef union addr64 {
2      uint64_t addr64;          /**< Full 64 bit address */
3      struct {
4          uint32_t high;        /**< High 32 bit of the address */
5          uint32_t low;         /**< Low 32 bit of the address */
6      } addr32;
7      uint32_t addr32array[2]; /**< Same as above only as an array */
8  } addr64_t;

```

Programflowet på en SPU er illustreret ved følgende pseudo kode.

```

1
2  while (cmd != SHUTDOWN) {
3     cmd = get_command() // block until command arrives from PPU
4     if (cmd == DO_TRIANGLE_TEST)
5         do_triangle_test()
6     else if (cmd == DO_OBB_TEST)
7         do_obb_test()
8 }

```

OBB tests og trekant test er stort set identiske, så derfor er kun trekant testen gengivet i pseudokode her.

```

1
2  test_list = get_test_list_slice() // Fetch first slice of CTT node list from
   PPU
3  fetch_buf = get_primitive_slice(test_list) // Start transfer of first slice of
   primitives
4
5  while (more_tests) {
6     swap_buffers(fetch_buf, calc_buf);
7     test_list = get_test_list_slice() // Fetch next CTT node slice
8     fetch_buf = get_primitive_slice(test_list) // Start next transfer of
   primitives
9     wait_for_buffer(calc_buf) // Wait till transfer is done
10    check_collision(calc_buf) // Do the actual collision checks
11 }
12 swap_buffers(fetch_buf, calc_buf);
13 wait_for_buffer(calc_buf) // Wait for final transfer
14 check_collision(calc_buf) // Do the actual collision checks

```

Som det ses på pseudokoden benyttes der double buffering til at hente primitiverne fra hovedhukommelsen. Dette gøres idet denne operation er en „gather“ operation, og derfor kan forventes at tage et stykke tid. Denne tid udnyttes til at udføre kollisions tests på de primitiver der allerede er hentet fra hovedhukommelsen. For at holde styr på de buffere der benyttes til disse overførelser, benyttes følgende struct.

```

1  struct buffer {
2     int n_test; /**< Number of tests */
3     int tag; /**< DMA tag */
4     struct dma_list_elem tests_dma_list[INPUT_SLICE_SIZE] __attribute__((
   __aligned__(ALIGNMENT))); /**< DMA list used to fetch CTT nodes */
5     struct dma_list_elem primitives_dma_list[2 * INPUT_SLICE_SIZE] __attribute__((
   __aligned__(ALIGNMENT))); /**< DMA list used to fetch primitives */
6     struct ctt_node tests[INPUT_SLICE_SIZE] __attribute__((__aligned__(ALIGNMENT
   )))); /**< The CTT nodes themselves */
7     char primitives[2 * INPUT_SLICE_SIZE * LARGEST_PRIMITIVE] __attribute__((
   __aligned__(ALIGNMENT))); /**< The primitives themselves */
8 };

```

Denne struct indeholder en liste af primitiver som skal testes. Denne liste er defineret så den både kan indeholde trekanter og OBB'er. Den indeholder også en liste af CTT noder, der beskriver hvilke tests der skal udføres. Derudover er der to lister der benyttes af DMA'erne til at hente de to førnævnte lister. Den første af disse benyttes til at dele CTT node listen op i mindre bidder, hvis denne bliver større end 16 KiB. Den anden DMA liste benyttes til at lave en „gather“ operation i hovedhukommelse,

når de enkelte primitiver skal hentes. Hvert element i denne liste svarer altså til er primitiv der skal hentes.

Resultaterne af kollisions testene gemmes i en global buffer på SPU'en. Denne buffer overføres vha. en DMA overførelse når alle tests er udført. Igen benyttes en DMA liste til at dele resultaterne op i passende bidder. Derefter signalernes PPU'en via en mailbox.

Brug af SPU SIMD funktionalitet

Som beskrevet i afsnit 9.1.4, benytter trekant testene og OBB testene SIMD funktionalitet på SPU'erne til bla. at beregne krydsproduktet og prikproduktet. Ud over dette benytter der også SIMD funktionalitet til alle beregninger af homogene transformationer og lignende. På alle platformene er denne funktionalitet pakket ind i en række funktioner der omfatter bla. invertering af en homogen transformation, beregning af en komposit transformation, og transformation af punkter vha. en homogen transformation. På CUDA og i386 platformene er denne funktionalitet implementeret vha. almindelige skalar beregninger. På CELL platformen er disse funktioner implementeret med SIMD instruktioner. Herunder er vist hvordan en funktion der ganger to 3×3 matricer sammen er lavet.

```

1 /* (a11, a12, a13) (b11, b12, b13) (r11, r12, r13) */
2 /* (a21, a22, a23) * (b21, b22, b23) = (r21, r22, r23) */
3 /* (a31, a32, a33) (b31, b32, b33) (r31, r32, r33) */
4
5     vector float v1, v2, v3; // Temp vectors
6     // Patter to splat first element in a vector
7     const vector unsigned char splat0 = (vector unsigned char) {
8         0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3};
9     // Patter to splat second element in a vector
10    const vector unsigned char splat1 = (vector unsigned char) {
11        4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6, 7};
12    // Patter to splat third element in a vector
13    const vector unsigned char splat2 = (vector unsigned char) {
14        8, 9, 10, 11, 8, 9, 10, 11, 8, 9, 10, 11, 8, 9, 10, 11};
15
16    v1 = spu_shuffle(a[0], a[0], splat0); /* v1 = (a11, a11, a11, a11) */
17    v2 = spu_shuffle(a[0], a[0], splat1); /* v2 = (a12, a12, a12, a12) */
18    v3 = spu_shuffle(a[0], a[0], splat2); /* v3 = (a13, a13, a13, a13) */
19
20    /* r11 = a11*b11 + a12*b21 + a13*b31 */
21    /* r12 = a11*b12 + a12*b22 + a13*b32 */
22    /* r13 = a11*b13 + a12*b23 + a13*b33 */
23    (*r)[0] = spu_mul(v1, b[0]);
24    (*r)[0] = spu_madd(v2, b[1], (*r)[0]);
25    (*r)[0] = spu_madd(v3, b[2], (*r)[0]);
26
27    v1 = spu_shuffle(a[1], a[1], splat0); /* v1 = (a21, a21, a21, a21) */
28    v2 = spu_shuffle(a[1], a[1], splat1); /* v2 = (a22, a22, a22, a22) */
29    v3 = spu_shuffle(a[1], a[1], splat2); /* v3 = (a23, a23, a23, a23) */
30
31    /* r21 = a21*b11 + a22*b21 + a23*b31 */
32    /* r22 = a21*b12 + a22*b22 + a23*b32 */
33    /* r23 = a21*b13 + a22*b23 + a23*b33 */
34    (*r)[1] = spu_mul(v1, b[0]);
35    (*r)[1] = spu_madd(v2, b[1], (*r)[1]);
36    (*r)[1] = spu_madd(v3, b[2], (*r)[1]);
37

```

```
38 v1 = spu_shuffle(a[2], a[2], splat0); /* v1 = (a31, a31, a31, a31) */
39 v2 = spu_shuffle(a[2], a[2], splat1); /* v2 = (a32, a32, a32, a32) */
40 v3 = spu_shuffle(a[2], a[2], splat2); /* v3 = (a33, a33, a33, a33) */
41
42 /* r31 = a31*b11 + a32*b21 + a33*b31 */
43 /* r32 = a31*b12 + a32*b22 + a33*b32 */
44 /* r33 = a31*b13 + a32*b23 + a33*b33 */
45 (*r)[2] = spu_mul(v1, b[0]);
46 (*r)[2] = spu_madd(v2, b[1], (*r)[2]);
47 (*r)[2] = spu_madd(v3, b[2], (*r)[2]);
```

Koden ovenfor består af tre trin. I hvert trin beregnes en række i resultat matrixen. Det observeres at for at beregne en sådan række skal der bruges de tre rækker fra B matrixen samt tre vektorer dannet af en kolonne fra A matrixen (se kommentarerne i koden). For at danne de tre vektorer fra A benyttes funktionen `spu_shuffle`. Denne funktion tager to vektorer og en vektor med et mønster og returnerer en tredje vektor. Mønstret beskriver hvorledes den tredje vektor ser ud. Mønstret er bygget op således at hver byte kan opfattes som en adresse. Adresser fra 0-15 adresserer den tilsvarende byte i den første vektor, mens adresser fra 16-31 adresserer den tilsvarende byte i den anden vektor. Den adresserede bytes placeres i den resulterende vektor på samme plads som i mønstret. Når de tre vektorer er opbygget udføres beregningen af resultat rækken med tre instruktioner. Den første `spu_mul` tager to vektorer, multiplicerer disse og returnerer resultatet i en tredje vektor. Denne vektor bruges herefter af instruktionen `spu_madd`. Denne instruktion multiplicerer to vektorer og lægger resultatet til en tredje vektor. Resultatet returneres i en fjerde vektor. Denne bruges igen i en `spu_madd` instruktion, der beregner det endelige resultat.

De andre funktioner til matrix beregninger og homogene transformationer bygger på samme principper og teknikker. Den interesserede læser henvises til kildekoden for mere information.

10.2 Test

10.2.1 Korrekthed og antal udførte tests

LibOPP og det velkendte kollisions test system PQP anvender grundlæggende de samme typer bounding volumes, træopbygning og algoritmer til tests.

Koden til at generere OBB hierarkierne for LibOPP er en første test-udgave lavet af JAJ, på basis af [8]. Dette blev gjort, da det i forbindelse med kørsel af hierarkiske metoder på parallelle arkitekturer er interessant at have et bredt, men ikke ret dybt træ. PQP's træopbygning understøtter ikke n-ary træer, så derfor var det nødvendigt at lave en ny implementation. Implementationen lider desværre under at have en ret ineffektiv convex hull algoritme der kører $O(n^2)$ og derved ikke kan håndtere ret større objekter end ca. 4000 trekanter. Dette antal er derfor valgt som det generelle loft for antallet af trekanter i objekter i scenerne.

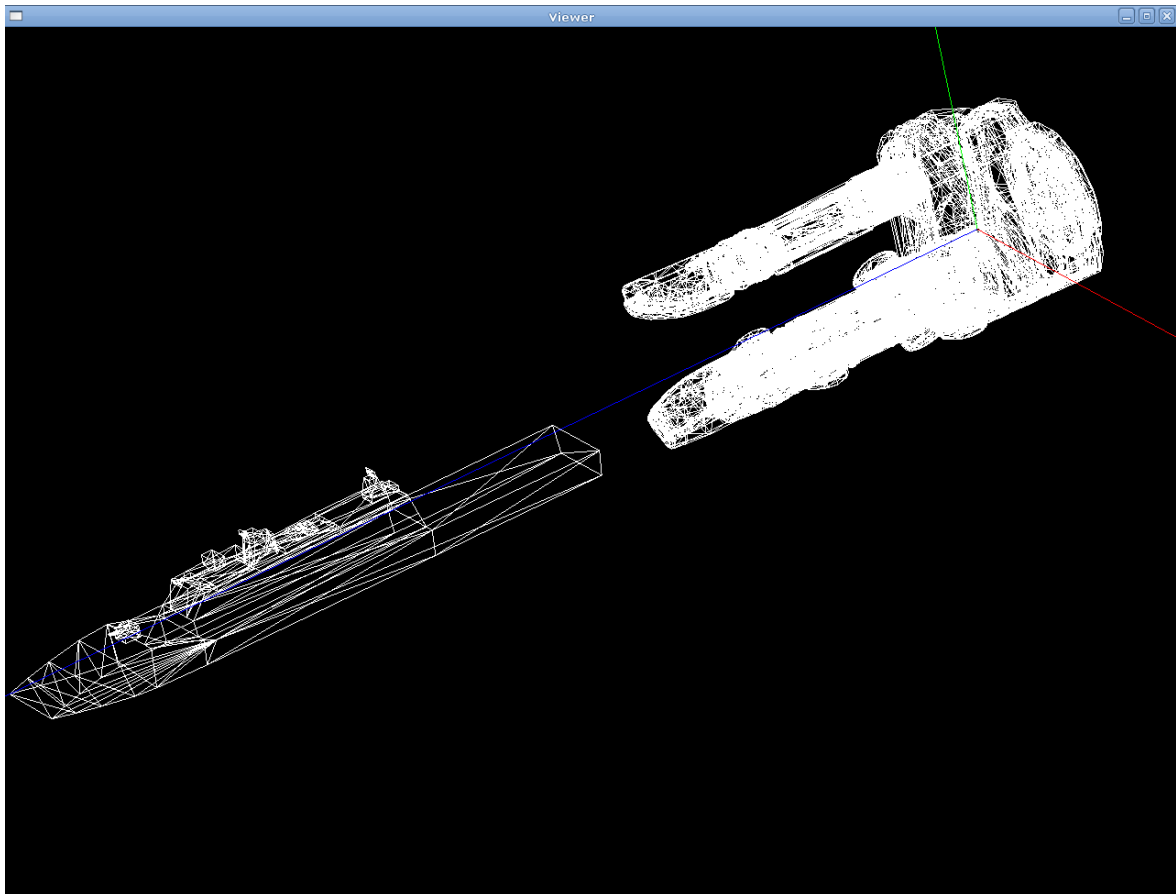
Til at sammenligne de to systemer anvendes en scene som vist på figur 10.13. Modellen af Schunk hånden indeholder ca. 4000 trekanter for hver bevægelig del (hånden består af 3 led per finger, samt en base) og modellen af skibet består af 307 trekanter. Dette giver en scene på ca. 40000 trekanter ialt. Skibets objekt træ testes imod alle led og basen, og bevæges i Z-retningen ind imod griberen. De opstillede z-værdier svarer til at skibet i $z = 0.4$ er tæt på griberen men uden at røre og for efterfølgende værdier bevæger sig igennem griberen. Endelig for $z = -0.3$ er skibet forbi bagsiden af griberen og er fri for kollision igen.

Som det fremgår af tabel 10.1 og 10.2 er de to systemer grundlæggende enige omkring antallet af trekanter i kollision. LibOPP udfører imidlertid langt flere tests. Dette skyldes at OBB'erne ikke fitter så godt til den underliggende geometri som PQP systemet. Hvad der gør, at LibOPP's bounding volumes ikke er så godt tilpasset geometrien, er uvist.

Set i lyset af at den nuværende implementation af LibOPP kun bruger binære objekt træer, vil der være en klar fordel i at anvende PQP's træopbygning.

Z-koordinat	OBBs tested	Tris tested	Collision pairs
0.4	13	0	0
0.3	1,161	117	42
0.2	8,149	1,089	383
0.1	18,481	1,965	777
0.0	28,967	2,700	1,209
-0.1	30,409	2,992	1,451
-0.2	11,905	814	347
-0.3	13	0	0

Tabel 10.1: Tabel over testede OBBer, tested trekanter og kollisionspar for PQP.



Figur 10.13: Den anvendte scene - et skib bevæger sig af Z-aksen imod en model af en Schunk griber.

Z-koordinat	OBBs tested	Tris tested	Collision pairs
0.4	13	0	0
0.3	4,913	1,521	42
0.2	27,467	16,412	383
0.1	77,425	41,065	778
0.0	114,539	53,073	1,217
-0.1	108,733	46,916	1,451
-0.2	27,289	10,313	347
-0.3	41	0	0

Tabel 10.2: Tabel over testede OBBer, testede trekanter og kollisionspar for LibOPP.

10.2.2 Performance tests

Følgende timing tests anvender i stedet for skibet i ovenstående scene en kopi af Schunk-hånden. Dette blev fundet at være nødvendigt for at opnå en tilstrækkelig kompleksitet i scenen.

Der er blevet udført 5 forskellige test. De forskellige tests varierer antallet af objekt træer der testes for kollision, for på denne måde at simulere scener med forskellig kompleksitet. Testene fremgår af tabel 10.3. I denne tabel ses hvor mange OBB og trekanten der er blevet testet i den pågældende scene, samt hvor mange kollisioner der er fundet i scenen.

Test	OBBs tested	Tris tested	Collision Pairs
Test 1	9,898,093	3,491,108	110,431
Test 2	6,950,973	2,472,383	80,371
Test 3	4,013,031	1,454,119	50,299
Test 4	1,069,561	436,291	13,191
Test 5	3	0	0

Tabel 10.3: Test setup.

For test 1 testes der nær ved 10 millioner OBB'er og 3.5 millioner trekanten. Antallet reduceres derefter gradvist i test 2-5. For hver af disse scener er performance blevet målt for alle tre platforme. Resultatet af disse målinger kan ses i tabel 10.4. Som det ses skal der en vis kompleksitet til i scenen før de parallelle algoritmer får overtaget.

Test	i386	CUDA	Cell
Test 1	22.48	16.22	13.65
Test 2	15.81	11.44	9.60
Test 3	9.16	6.70	7.07
Test 4	2.51	1.91	2.01
Test 5	0	0	0

Tabel 10.4: Sammenligninger af platforme. Tiden for 10 kørsler i sekunder.

Men selv ved en rimeligt kompliceret scene er der med de nuværende parallelle implementationer ikke meget performance at hente. Et speedup på omkring 1.6 er hvad der kan blive til i bedste fald. Det kunne dog være interessant at se om mere komplekse scener giver en endnu større forskel. Dette vil dog kræve lidt mere arbejde med memory håndtering, idet de nuværende implementationer ikke er så intelligente på dette område.

Generelt er der en række punkter der kan optimeres for at forbedre ydelsen. Der er ingen tvivl om at et af de svage punkter i den nuværende implementation er den måde som hovedprocessoren benyttes til at finde børn til de noder der er i kollision (`get_children`). Hvis denne del kunne paralleliseres ville der kunne vindes en del. Derudover skal der også kigges på den måde som de homogene transformationer bruges på. Da disse er ret dyre vil der kunne vindes ved at tænke mere over deres udregning og evt. bruge en cache. Den implementation der benyttes lige nu er temmelig unintelligent på det område. På CUDA platformen er der en række åbenlyse ting der skal rettes for at forbedre performance. Generelt er den begrænsende faktor hukommelsestilgangen. Anvendelse af konstant-hukommelsen til at gemme objekt træernes transformationer er en oplagt forbedring. I den nuværende implementation har dette dog ikke været brugt, da det giver nogle restriktioner mht. pointere og structs. Ligeledes vil lokal caching af OBB'er/trekanten i shared memory være med til forbedre tilgangen til hovedhukom-

melsen.

CELL platformen har også en række åbenlyse problemer. Et af disse er at der ikke er plads til særligt mange test af gangen på de enkelte SPU'er. Dette skyldes den forholdsvis lille lokal hukommelse. Den vil utvivlsomt kunne udnyttes bedre end det er tilfældet nu. Derudover vil en omskrivning af algoritmerne specielt til CELL platformen også give større mulighed for performance optimering.

Del III

Resultater og konklusion

Kapitel 11

Erfaringer og problemer vedr. udviklingsprocessen

11.1 Generelt

Erfaring er påkrævet for effektiv udnyttelse af enhver softwarepakke eller hardwarearkitektur. Dette gælder ikke mindst ved programmering på parallelle arkitekturer. Det er generelt svært at vurdere effekten af en ændring eller en bestemt måde at lave tingene på. Derfor er det vigtigt at eksperimentere og gennem den vej skaffe sig erfaring med den aktuelle platforms styrker og svagheder.

11.2 GPU

Udviklingen på GPU'en vha. CUDA har givet en række udfordringer og problemer der kort skal nævnes i følgende afsnit.

Den fælles kodebase med de andre platforme har været en udfordring for CUDA. Alle funktioner der kaldes fra en CUDA kerne skal kompileres separat med brug af `nvcc`. For at en funktion skal kunne anvendes, er det nødvendigt at have en speciel modifier, `__device__` i deklARATIONEN. Dette kræver lidt omtanke, da trekant-trekant test algoritmerne er afhængig af en del funktioner der så alle skal have tilføjet den ekstra modifier. Kodebasen for algoritmerne deles med de to andre platform, i386 og CELL der ikke godtager `__device__` modifieren. Derfor har det været nødvendigt at definere en makro `_CUDA_FUNC_MODIFIER` der er sat til modifieren når der kompileres med `nvcc`, og tom når andre compilere anvendes. Denne makro er så tilføjet til alle funktioner der bliver inkluderet.

Et andet problem har været at `nvcc` forventer alle funktioner der inkluderes i en kerne er til stede i filer med suffikset `.cu`. Da det ikke har været hensigtsmæssigt at skulle ændre filnavnene på de oprindelige filer har løsningen været at lave symlinks til filerne, med suffiks `.cu`.

For devillers trekant-trekant test algoritmen, har der været problemer med korrektheden af koden ved kompilering med `nvcc` til CUDA. Det viste sig at optimeringen i den underliggende `nvopencc` var for aggressiv og fjernede hele blokke af `if-else` fra koden. Dette problem viste sig ved brug af CUDA Toolkit version 1.1 der blev anvendt i starten af projektperioden og senere også med version 2.0 beta. Efter omfattende fejlsøgning, der var besværliggjort af at der ikke kunne udskrives tekst fra CUDA kernen, viste det sig at tilføjelsen af en `volatile` til visse variable gav et korrekt resultat. I fejlsøgningen var det en stor hjælp at få `nvcc` til at gemme PTX-koden. Dette kan gøres med `-ptxas-options=-v -opencc-options -LIST:source=on`, der samtidig rapporterer registerfor-

brug af CUDA kernerne.

Afslutningsvis skal det nævnes at CUDA er generelt nemt komme i gang med, men er desværre stærkt præget af at man ikke kan få nærmere detaljer omkring den underliggende hardwarearkitektur. Dette betyder at for mere avancerede algoritmer og metoder, må man selv eksperimentere eller søge sig frem på NVIDIA's forum. Desuden gør ovenstående begrænsninger med filnavne og scopes det besværligt at integrere i større projekter med en fælles kodebase.

11.3 CELL/BE

Under udviklingen på CELL platformen har der været en række udfordringer og problemer. De vigtigste af disse vil blive opremset i det følgende.

Ved projektets start blev SDK version 3.0 benyttet. Undervejs i udviklings processen blev det bemærket at compileren (gcc) fejlede med en segmentations fejl under visse omstændigheder. Desværre var disse omstændigheder almindelige nok til at være en alvorlig hemske. Det blev derfor besluttet at undersøge fejlen nærmere. Et test program blev lavet og det blev kortlagt under hvilke omstændigheder fejlen opstod. Den information blev videregivet på IBM CELL forum. Her var der en venlig IBM administrator der anerkendte at der var tale om en fejl i compileren, og han åbnede en bug report. Senere i forløbet udkom SDK 3.03 og i denne version af SDK'et er fejlen tilsyneladende rettet.

Et andet problem der er blevet konstateret under udviklingsprocessen, er at det medfølgende Eclipse IDE tilsyneladende ikke fungerer helt korrekt, når det afvikles under Gentoo Linux. Det er ikke lykkedes at få IDE'et til at fungere sammen med system sim simulatoren eller PlayStation 3 konsollen. Der har dog ikke været brugt ret meget tid på at løse problemerne.

Der har ligeledes været problemer med at få Visual Performance Analyzer softwaren til at fungere efter hensigten. Dette skyldes at det kernemodul der indgår i denne software retter sig mod en bestemt kerne version, der tilsyneladende ikke er beregnet på PlayStation 3, men derimod IBM CELL Blade servere. Dette betyder at de data der opsamles i VPA ikke vises korrekt, og værdien af dem i praksis er derfor begrænset.

Endeligt skal det nævnes at der har været problemer med at få Valgrind til at virke med CELL platformen. Hvad der er skyld i dette er uklart. I første omgang blev der benyttet en version af Valgrind der indeholdt en fejl. Efter en ny version var blevet downloadet og patchet blev det igen forsøgt at anvende Valgrind, men uden held.

Afslutningsvis skal det nævnes at miljøet omkring CELL platformen er meget komplet. Dokumentationen er fyldestgørende og de medfølgende værktøjer er generelt af høj kvalitet.

Kapitel 12

Konklusion

I dette speciale, er det blevet undersøgt om to forskellige typer af parallelle arkitekturer egner sig til acceleration af kollisionstest i forbindelse med gribesimulering. Fokus har været på at lære de to arkitekturer at kende, samt udforske deres performance i et realistisk scenarie.

Den første platform der er blevet undersøgt er en familie af grafikprocessorer (GPU'er) fra firmaet NVIDIA. Denne arkitektur er kendetegnet ved en stor grad af parallelitet, og en meget høj teoretisk ydelse.

Den anden platform der er blevet undersøgt er en PlayStation 3, baseret på en Cell BE processor. Denne platform er ligeledes karakteriseret ved en meget høj teoretisk ydelse, men den har ikke så mange parallelle elementer som GPU'en.

De to platforme er begge blevet testet med to forskellige trekant-trekant kollisionsdetektions algoritmer. Den første af disse er den såkaldte segment-piercing algoritme, der er en let forståelig algoritme baseret på simple geometriske betragtninger. Denne algoritme er ikke den hurtigste der findes, men den har visse fordele på en parallel arkitektur idet den ikke indeholder så mange branches. Den anden algoritme der er blevet implementeret og testet på de to platforme, er den såkaldte devillers algoritme. Denne algoritme bygger på nogle mere avancerede geometriske betragtninger. Algoritmen har teoretisk en performance fordel frem for segment-piercing algoritmen, men den indeholder flere branches, og har derfor i nogle tilfælde et handikap på parallelle arkitekturer.

Der er blevet udført performance tests af algoritmerne på de to platforme. Som forventet er segment-piercing hurtigere på GPU platformen end devillers. Dette skyldes at GPU'en tilsyneladende ikke er så stærk når det kommer til branches. Forskellen på de to algoritmer er ca. en faktor 2. Segment-piercing algoritmen opnår en performance på ca. 71 millioner trekant-trekant tests/sekund. Mens devillers opnår ca. 39 millioner tests/sekund. Til sammenligning opnår en 2.66 GHz Intel core 2 duo PC processor ca. 18 millioner tests/sekund med devillers, og ca. 9 millioner tests/sekund med segment-piercing. Igen et tegn på at devillers fortrækker en processor med god branchlogik. På Cell platformen er devillers hurtigere end segment-piercing. Her vinder devillers med ca. en faktor 4.5. Devillers opnår ca. 52 millioner tests/sekund, på denne platform, mens segment-piercing må nøjes med ca. 11 millioner tests/sekund. Her kommer devillers altså til sin ret.

Ud over trekant testene er der også blevet implementeret to forskellige OBB kollisions algoritmer. Exhaustive algoritmen er i lighed med segment-piercing algoritmen en „lige ud af landevejen“ algoritme til OBB tests. Denne algoritme er letforståelig, men meget dyr i antal operationer. Anderledes forholdet sig med Gottschalk algoritmen. Denne algoritme benytter mere sofistikerede geometriske betragtninger og optimeringer for at opnå en høj ydelse. Performancemæssigt skinner det klart igennem at Gottschalk er exhaustive overlegen. Med 160 millioner tests per sekund på GPU'en, ses det ty-

deligvis at algoritmen er effektiv.

Alle disse algoritmer er blevet samlet i en implementation af en komplet hierarkisk kollisions tester der anvender OBB'er som bounding volumes. Denne kollisions tester er forsøgt opbygget på en måde der egner sig til implementation på en parallel arkitektur. Netop dette har vist sig at være en stor udfordring. En hierarkisk metode har i sin natur en mængde serielle afhængigheder. Dette gør det svært at partitionere og ikke mindst load balance problemet på en fornuftig måde.

Dette ses også tydeligt i de opnåede resultater med denne metode. Metoden er blevet implementeret på både CELL og CUDA platformen, og der er ligeledes lavet en reference implementation på i386. Resultaterne viser tydeligt at den nuværende parallelle implementation ikke vinder stort over i386 implementationen. Dog har de parallelle platforme en fordel når problemet bliver tilstrækkeligt stort. Men selv der er forskellen ikke så stor som man kunne have håbet på. Dette skyldes mange forhold. En ting er at algoritmerne stadig har mange steder hvor de kan optimeres yderligere. Eksempelvis vil der ganske give kunne hentes en del performance, hvis de enkelte algoritmer blev tilpasset til de enkelte platforme, i stedet for at være generelle som de er nu. En anden ting der skal overvejes er om den nuværende arbejdsdeling er hensigtsmæssig. Eksempelvis kunne man forestille sig at mere af arbejdet end blot trekant og OBB test blev lagt ud på de parallelle enheder.

Endeligt skal det nævnes at der måske skal kikkes mere på alternative algoritmer til kollisionstest på parallelle platforme. Måske skal der endda laves helt nye algoritmer til formålet.

Når det kommer til at arbejde med de to platforme, kan det konkluderes at de begge indeholder en række gode værktøjer, dog bærer begge platforme lidt præg af at de stadig er unge. Der er stadig et par skarpe hjørner hist og her der kræver lidt afrunding og tilpasning. Men alt i alt er det ikke forbundet med nævneværdige problemer at udvikle programmer til nogen af platformene. Hvis platformene sammenlignes indbyrdes, er det dog klart at den letteste tilgængelige platform er GPU'en. Dette skyldes ikke GPU platformen i sig selv, men snarere Cell platformen, idet denne er præget af at mange ting skal styres manuelt. Dette gælder i særdeleshed hukommelsen og de enkelte processor elementer. GPU'en er derimod mere tolerant, over for f.eks. skæv hukommelsestilgang.

12.1 Perspektivering

Målet med specialet var at udvikle en færdig kollisionstester baseret på parallel hardware. Dette mål er sådan set også nået. Men løsningen lader dog en del tilbage at ønske rent performance-mæssigt. Det står derfor klart at der skal mere arbejde til før en endelig version er klar til brug. Der er dog blevet indhøstet værdifuld erfaring under arbejdet med specialet, som forhåbentligt kan benyttes i det videre arbejde. Umiddelbart kan det anbefales at basere det videre arbejde på CUDA platformen. Dette har to grunde, dels er GPU lettere at integrere i en færdig løsning, idet den allerede sidder i en PC. I modsætning hertil er PlayStationen en selvstændig enhed, og der skal derfor en del arbejde for at integrere denne i en PC løsning. Derudover er der siden specialets start kommet nye og mere kraftfulde GPU'er på markedet. Disse indeholder desuden features som har være savnet under dette speciale, bla. mulighed for atomiske operationer.

Der ligger forsat et stort potentiale i fremtiden at udvikle fremtidige algoritmer og kontrolsoftware til parallelle beregningsenheder. Hvor rent serielle processorer ikke længere skalerer til fremtidens problemstørrelser kan parallelle enheder forsat opretholde Moore's lov og dermed hjælpe til at robotter kan hjælpe os i større udstrækning i hverdagen og på arbejdspladsen.

Del IV

Appendix

Ordliste

CUDA Compute Unified Device Architecture - Teknologi fra NVIDIA til general-purpose beregninger på GPU'er. 51

DMA Direct Memory Access. 62

DOP Discretely Oriented Polytope. 5

FPGA Field Programmable Gate Array. 5

Fused-multiply-add En instruktion der udregner produktet af to tal og lægger resultatet til en akkumulator: $a \leftarrow a + b * c$, eventuelt kan der indgå afrundinger i beregningen. 57, 65

GPU Graphics Processing Unit. 4

GUI Graphical User Interface. 48

MP Multi Processor. 54

NUMA Non-Uniform Memory Access. 62

Parallelepiped I 3D en prisme hvis sider alle er parallelogrammer. 27

Pixel Shader Et program eller en funktion på en GPU der opererer på en individuel pixel i billedet. Er også kendt som en Fragment Shader. 49

RISC Reduced instruction set computer - CPU design filosofi, hvis mål er at simplificere implementationen af CPU'en, ved at nedbringe kompleksiteten af bla. instruktionssættet. Herved opnås en række performance fordele, tilgængæld bliver compileren typisk mere avanceret. 60

Sidedness Test En test der afgør et primitivs placering i forhold til et andet orienteret primitiv, f.eks. en linie imod en trekantside og en trekant imod et plan i rummet. 27

SIMD Single Instruction Multiple Data. 60

SP Stream Processor. 54

Superskalar CPU arkitektur, det gør det muligt for CPU'en at udføre flere instruktioner i parallel. 60

T&L Transform and Lighting - Opgaven med at transformere objekter imellem rum og udregning af lysforhold. 49

Vertex Shader Et program eller en funktion på en GPU der opererer på et hjørne i grafikprimitiver ad gangen. Kan ikke tilføje eller fjerne nye hjørner, kun modificere eksisterende, f.eks. translation, rotation, skalering mm. 49

Tabeller

4.1	Basale operationer i vektoroperationer	19
4.2	Basale operationer i segment-plan og linie-trekant tests	30
4.3	Basale operationer i en segment-piercing trekant-trekant test	30
4.4	Basale operationer i Devillers trekant-trekant testen	36
4.5	Basale operationer i Segment-OBB face test	40
4.6	Basale operationer i Exhaustive OBB-OBB test	40
4.7	Basale operationer i Gottschalk OBB-OBB test	46
5.1	Approximativ tilgangstid for hukommelsestyper i CUDA	57
7.1	PPU til SPU overførelse ved sekventiel brug af DMA'erne	79
7.2	SPU til SPU overførelse ved sekventiel brug af DMA'erne	80
7.3	PPU til SPU overførelse ved brug af DMA lister (128 byte element størrelse)	80
7.4	SPU til SPU overførelse ved brug af DMA lister (128 byte element størrelse)	81
7.5	PPU til SPU overførelse ved brug af DMA lister (varierende element størrelse)	82
7.6	SPU til SPU overførelse ved brug af DMA lister (varierende element størrelse)	83
9.1	Antal tests/sekund for devillers algoritmen ved varierende objektstørrelser.	129
9.2	Antal tests/sekund for segment-piercing algoritmen ved varierende objektstørrelser.	129
9.3	Antal tests/sekund for Gottschalk algoritmen på CUDA.	131
10.1	Tabel over testede OBBER, testede trekanter og kollisionspar for PQP.	150
10.2	Tabel over testede OBBER, testede trekanter og kollisionspar for LibOPP.	151
10.3	Test setup.	152
10.4	Sammenligninger af platforme. Tiden for 10 kørsler i sekunder.	152

Figurer

1.1	En industrirobot påmonteret en parallel griber.	1
1.2	Industrirobotter i bilproduktion	2
1.3	Mangeakset hånd	3
1.4	Tre-fingret griber	4
1.5	Model for projektførløb.	6
2.1	3D model repræsentation.	9
2.2	Eksempel på opbygning af kompleks objekt med CSG.	11
2.3	Eksempel på model af robot med svejsepistol.	12
2.4	Eksempler på kollision i 2D.	13
2.5	Eksempler på kollision i 3D.	13
3.1	Problemområdet	15
3.2	Vertikal dekomposition af problemområdet i fire dele.	15
3.3	Horisontal dekomposition af problemområdet.	16
3.4	Illustration af Amdahl's lov.	16
4.1	Bounding volume hierarki	20
4.2	Bounding volume hierarki	20
4.3	Niveauer i BVH	21
4.4	BVH træer	21
4.5	Top niveauer	22
4.6	Objekttræer og deres tilhørende bounding volumes.	23
4.7	Samtidig traversering: I a) er BV'erne A_0 og B_0 er i kollision så i b) bliver A_1, A_2 testet imod B_1, B_2	24
4.8	Skiftende traversering: BV'erne A_0 og B_0 er i kollision i a). A_0 testes først imod B_1 og B_2 (b). Da disse også er i kollision skiftes der træ og A_1 og A_2 testes imod B_0 i c).	24
4.9	Typer af bounding volumes	25
4.10	BV fit	25
4.11	Trekant-trekant skæringer.	27
4.12	Linie-trekant skæring.	28
4.13	Egenskaber ved 2D krydsprodukt.	30
4.14	Devillers algoritmen, trekanternes intervaller.	31
4.15	Devillers algoritmen, geometrisk prædikat.	31
4.16	Placering af intervaller på linien L	33
4.17	Overblik over interval test.	34

4.18	Klassificering af p_1 i forhold til T_2 . Gengivet fra [5]	35
4.19	Opdeling i regioner hvis p_1 ligger i regionen R_1 . Gengivet fra [5]	36
4.20	Beslutningstræ, der viser de tests der skal udføres hvis p_1 ligger i region R_1 . Gengivet fra [5]	37
4.21	Opdeling i regioner hvis p_1 ligger i regionen R_2 . Gengivet fra [5]	38
4.22	Beslutningstræ, der viser de tests der skal udføres hvis p_1 ligger i region R_2 . Gengivet fra [5]	39
4.23	Separerende plan og akse	41
4.24	Orthogonal projektion af OBB'er på kandidatakse. (Gengivet fra [9])	42
4.25	Halvbredde af intervaller. (Gengivet fra [9])	43
4.26		43
4.27	Vertex-vertex specialtilfældet. En udadrettet normalvektor til en face fra B (markeret med rødt) har størst komponent af vektoren imellem de nærmest punkter, og udgør i dette tilfælde en separerende akse.	45
4.28	Edge-vertex specialtilfældet. Krydsproduktet af edges fra hver af boksene, vist med rødt udgør separerende akse.	45
5.1	Klassisk 3D GPU pipeline	48
5.2	Evolution af 3D grafikpipeline	49
5.3	GPU/CPU ydeevne	50
5.4	Udnyttelse af CPU/GPU chipareal	50
5.5	Repræsentation af en båndmatrice i en GPU	51
5.6	CUDA software lag	52
5.7	CUDA 2-dimensionel trådningsmodel	53
5.8	CUDA hukommelsesmodel	53
5.9	GeForce 8800 GTX blokdiagram	54
5.10	GeForce 8 multiprocessor hardware model	55
5.11	GeForce 8800 GTX grafikkort	56
6.1	Overblik over Cell BE arkitekturen.	60
6.2	Overblik over PowerPC Processor Element.	60
6.3	Blokdiagram over en PPE.	61
6.4	Blokdiagram over en SPE.	62
6.5	Overblik over PowerPC Processor Element.	63
6.6	Sony Playstation 3.	64
6.7	Cell performance sammenligning.	66
6.8	Cell performance sammenligning.	67
6.9	Data Transfer Lists	68
6.10	Typisk ALF applikation.	69
7.1	Typisk flow for kørsel på GPU	71
7.2	Memory timing test, fra 32 til 2048 bytes i 32 bytes increments.	73
7.3	Memory timing test, fra 512 til 51200 bytes i 512 bytes increments.	74
7.4	Memory timing test, fra 102400 til 10240000 bytes i 102400 bytes increments.	75
7.5	Tid per kerne kørsel ifht. antal tråd blokke. Der køres fra et 1x1 til et 16x16 grid.	77
9.1	$M \times N$ resultat matrix	110

9.2	Illustration af AOS.	111
9.3	Illustration af SOA.	112
9.4	Illustration af den første workunit.	113
9.5	Illustration af den anden workunit.	114
9.6	Flowchart over trekant-trekant test på CELL med Libspe.	116
9.7	CUDA 2D grid med trådblokke	120
9.8	Flowchart over trekant-trekant test på CUDA.	122
9.9	Ydelse af GPU med Segment-pierce algoritme som funktion af grid størrelse ved en trådblokstørrelse på 13×13	123
9.10	Ydelse af GPU med Devillers algoritme som funktion af grid størrelse.	124
9.11	Ydelse af GPU med Segment-pierce algoritme som funktion af trådblok dimension.	124
9.12	Ydelse af GPU med Devillers algoritme som funktion af trådblok dimension.	125
9.13	Antal tests/sekund for devillers algoritmen ved varierende workunit størrelse.	130
9.14	Antal tests/sekund for segment-piercing algoritmen ved varierende workunit størrelse.	130
10.1	Objekt træ A.	133
10.2	Objekt træ B.	133
10.3	Objekt træ C.	134
10.4	Kollisions test træ (CTT) for objekt træ A og B.	135
10.5	Kollisions test træ (CTT) for en hel scene, bestående af A, B og C	136
10.6	Implicit representation af et binært træ i et array.	136
10.7	En scene	138
10.8	Overordnet flowdiagram for CTT noder i CUDA model testen.	139
10.9	Flowdiagram for test af OBB noder i CUDA model testen.	139
10.10	Fordeling af get_children og gather på to tråde	140
10.11	Overblik over CELL implementationen af den hierarkiske metode.	142
10.12	Overblik over datastrukturer i CELL implementationen.	144
10.13	Den anvendte scene - et skib bevæger sig af Z-aksen imod en model af en Schunk gribler.	151
A.1	Geometrisk fortolkning af 3×3 determinant.	170
A.2	Geometrisk betydning af trippel skalar produkt.	171

Litteratur

- [1] Nuzhet Atay, John W. Lockwood og Burchan Bayazit. A collision detection chip on reconfigurable hardware, 2005.
- [2] NVIDIA Corporation. Transform and lighting. http://developer.nvidia.com/object/Technical_Brief_TandL.html, Oktober 1999.
- [3] NVIDIA Corporation. Pixel shaders. http://www.nvidia.com/object/feature_pixelshader.html, Marts 2001.
- [4] NVIDIA Corporation. Vertex shaders. http://www.nvidia.com/object/feature_vertexshader.html, Marts 2001.
- [5] Olivier Devillers og Philippe Guigue. Faster triangle-triangle intersection tests. Research Report 4488, INRIA, 2002.
- [6] Christer Ericson. *Real-Time Collision Detection*. Morgan Kaufmann Publishers, 2005.
- [7] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon og D. Walker. *Solving Problems on Concurrent Processors*. Prentice-Hall International Editions, 1988.
- [8] S. Gottschalk, M. C. Lin og D. Manocha. Obbtree: a hierarchical structure for rapid interference detection. *International Conference on Computer Graphics and Interactive Techniques, Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, side 171-180, 1996.
- [9] Stefan Gottschalk. *Collision Queries Using Oriented Bounding Boxes*. Ph.d.-afhandling, The University of North Carolina at Chapel Hill, Department of Computer Science, 2000.
- [10] Mark Harris. Mapping computational concepts to gpus. I *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, side 50, New York, NY, USA, 2005. ACM.
- [11] IBM. Powerpc operating environment architecture, book iii - version 2.02, 2005.
- [12] IBM. Powerpc user instruction set architecture, book i - version 2.02, 2005.
- [13] IBM. Powerpc virtual environment architecture, book ii - version 2.02, 2005.
- [14] IBM. Software development kit for multicore acceleration version 3.0, programming tutorial, 2005, 2007.
- [15] IBM. Software development kit for multicore acceleration version 3.0, accelerated library framework for cell broadband engine, programmer's guide and api reference, 2006, 2007.

-
- [16] IBM. Software development kit for multicore acceleration version 3.0, programmer's guide, 2006, 2007.
- [17] IBM. Synergistic processor unit instruction set architecture - version 1.2, 2007.
- [18] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. august 1985.
- [19] Jyllands-Posten. Industriens nye handyman. <http://jp.dk/arkiv/?id=957901>, December 2007.
- [20] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer og D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589-604, 2005.
- [21] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.
- [22] M. C. Lin og S. Gottschalk. Collision detection between geometric models: a survey. side 37-56.
- [23] T. Möller. A fast triangle-triangle intersection test. *Graphics Tools*, side 25-30, 1997.
- [24] NVIDIA. Nvidia geforce 8800 gpu architecture overview. http://www.nvidia.com/page/8800_tech_briefs.html, November 2006.
- [25] NVIDIA. Cuda programming guide 1.1. <http://developer.nvidia.com/object/cuda.html>, 2007.
- [26] NVIDIA. Geforce 8800. http://www.nvidia.com/page/geforce_8800.html, December 2007.
- [27] NVIDIA. Ptx: Parallel thread execution isa version 1.1. http://developer.nvidia.com/object/cuda_develop.html, 2007.
- [28] Andreas Raabe, Stefan Hochgürt, Joachim K. Anlauf og Gabriel Zachmann. Hardware-accelerated collision detection using bounded-error fixed-point arithmetic. *WSCG*, 14, 2006.
- [29] RoboCluster. Handyman. <http://www.robocluster.dk/projekter/handyman>, December 2007.
- [30] RoboCluster. Om robocluster. <http://www.robocluster.dk/profile>, December 2007.
- [31] Wikipedia. Altivec — wikipedia, the free encyclopedia. <http://en.wikipedia.org/w/index.php?title=Altivec&oldid=178162609>, December 2007.
- [32] Wikipedia. Blitter — wikipedia, the free encyclopedia. <http://en.wikipedia.org/w/index.php?title=Blitter&oldid=163085651>, December 2007.
- [33] Wikipedia. Cell (microprocessor) — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Cell_%28microprocessor%29&oldid=179844350, December 2007.
- [34] Wikipedia. Geforce 3 series — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=GeForce_3_Series&oldid=173851650, December 2007.

- [35] Wikipedia. Geforce 8 series — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=GeForce_8_Series&oldid=181208338, December 2007.
- [36] Wikipedia. Graphics processing unit — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Graphics_processing_unit&oldid=179747777, December 2007.
- [37] Wikipedia. Hand — wikipedia, the free encyclopedia. <http://en.wikipedia.org/w/index.php?title=Hand&oldid=177303613>, December 2007.
- [38] Wikipedia. Human evolution — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Human_evolution&oldid=176861826, December 2007.
- [39] Wikipedia. Powerpc — wikipedia, the free encyclopedia. <http://en.wikipedia.org/w/index.php?title=PowerPC&oldid=180424870>, December 2007.
- [40] Wikipedia. Parallelepiped — wikipedia, the free encyclopedia, 2008. [Online; accessed 20-September-2008].
- [41] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands og Katherine Yelick. The potential of the cell processor for scientific computing. I *CF '06: Proceedings of the 3rd conference on Computing frontiers*, side 9-20, New York, NY, USA, 2006. ACM.

Bilag A

Determinanten og trippel skalar produktet

I artiklen *Faster Triangle-Triangle Intersection Tests* [5], benyttes et geometrisk prædikat baseret på en determinant, for at fastslå om et punkt d ligger over, under eller i, det plan hvori trekant abc ligger. Hvilken side af planet der udgør det positive halvplan hhv. det negative halvplan, afgøres ved hjælp af højrehåndsreglen, når det vedtages at trekant abc hjørner enumereres i rækkefølge mod uret. Prædikatet er positivt når d befinder sig i det positive halvplan, nul når d befinder sig i selve planet, og negativt når d befinder sig i det negative halvplan. I artiklen defineres det geometriske prædikat som det ses i ligning A.1

$$[a, b, c, d] = \begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{vmatrix} \quad (\text{A.1})$$

I ligning A.1, bemærkes det at alle punkter er blevet udvidet til at indeholde 4 koordinater, og at dette fjerde koordinat er blevet sat til 1. Dette er nødvendigt idet det, som bekendt, kun er muligt at bestemme determinanten af en kvadratisk matrice. Dette lille trick flytter punkterne op på en manifold i 4D rummet, og har ingen betydning for metoden.

Ved at udføre to på hinanden følgende række addition operationer ¹ på matricen i ligning A.1 fåes

$$[a, b, c, d] = \begin{vmatrix} a_x - d_x & a_y - d_y & a_z - d_z & 0 \\ b_x - d_x & b_y - d_y & b_z - d_z & 0 \\ c_x - d_x & c_y - d_y & c_z - d_z & 0 \\ d_x & d_y & d_z & 1 \end{vmatrix} \quad (\text{A.2})$$

Ligning A.2 kan reduceres yderligere ved at benytte ko-faktor ekspansion, idet determinanten af en matrice kan skrives som

$$|\mathbf{A}| = a_{1j}\mathbf{A}_{1j} + a_{2j}\mathbf{A}_{2j} + \dots + a_{nj}\mathbf{A}_{nj} \quad (\text{A.3})$$

¹Erstat en række med summen af rækken og et multiplum af en anden række.

Hvor \mathbf{A}_{ij} er givet ved

$$\mathbf{A}_{ij} = (-1)^{i+j} |\mathbf{M}_{ij}| \quad (\text{A.4})$$

Termen \mathbf{M}_{ij} er den matrice der fremkommer ved at slette række i og kolonne j fra den oprindelige matrice \mathbf{A} .

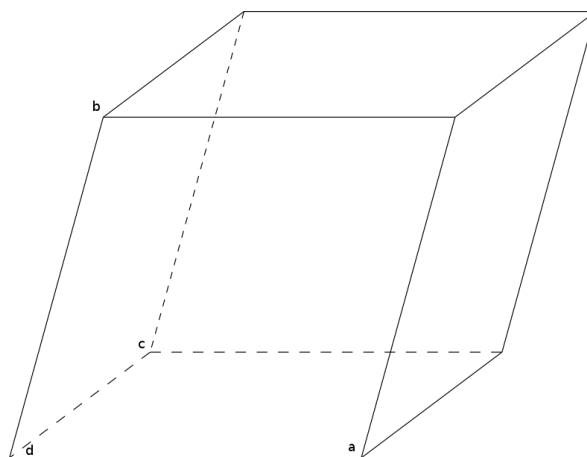
Herved kan ligning A.2 skrives som

$$[a, b, c, d] = 0 \cdot \mathbf{A}_{14} - 0 \cdot \mathbf{A}_{24} + 0 \cdot \mathbf{A}_{34} - 1 \cdot \mathbf{A}_{44} \quad (\text{A.5})$$

Hvilket kan skrives som

$$[a, b, c, d] = \begin{vmatrix} a_x - d_x & a_y - d_y & a_z - d_z \\ b_x - d_x & b_y - d_y & b_z - d_z \\ c_x - d_x & c_y - d_y & c_z - d_z \end{vmatrix} \quad (\text{A.6})$$

Determinanten i ligning A.6 kan geometrisk set opfattes som volumen (med fortegn) af parallelepipedet (se figur A.1) der udspringer af vektorene \vec{da} , \vec{db} , og \vec{dc} . Når d ligger som vist på figuren er volumen positiv, kommer d derimod over på den anden side af det plan som a , b og c ligger i, bliver volumen negativ [40]. Dermed kan positionen relativt til trekant abc bestemmes.



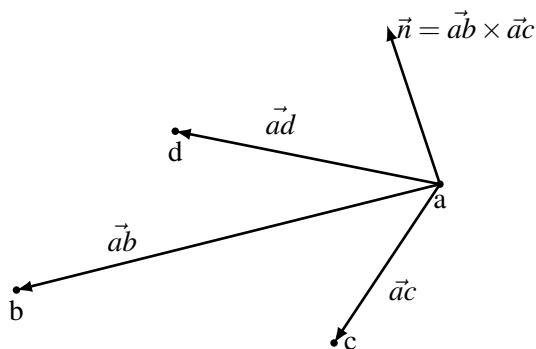
Figur A.1: Geometrisk fortolkning af 3x3 determinant.

Beregnes determinanten fra ligning A.6 fåes

$$\begin{aligned} [a, b, c, d] = & b_x c_y d_z - a_x c_y d_z - b_y c_x d_z + a_y c_x d_z + a_x b_y d_z - a_y b_x d_z - b_x c_z d_y + a_x c_z d_y \quad (\text{A.7}) \\ & + b_z c_x d_y - a_z c_x d_y - a_x b_z d_y + a_z b_x d_y + b_y c_z d_x - a_y c_z d_x - b_z c_y d_x + a_z c_y d_x \\ & + a_y b_z d_x - a_z b_y d_x - a_x b_y c_z + a_y b_x c_z + a_x b_z c_y - a_z b_x c_y - a_y b_z c_x + a_z b_y c_x \end{aligned}$$

Som beskrevet ovenfor baserer metoden i artiklen sig på determinanter. Som alternativ kan det såkaldte trippel skalar produkt anvendes. I det følgende vises det at de to metoder er ens.

Figur A.2 viser den geometriske betydning af et trippel skalar produkt. Som det ses på figuren positionen af d bestemmes relativt til trekant abc ved at bestemme vinklen mellem vektor \vec{ad} og normal vektoren \vec{n} . Da et prikprodukt afhænger af cosinus til vinklen mellem to vektorer, kan dette benyttes til at finde ud af om vinklen mellem \vec{ad} og \vec{n} er over under eller over 90° , svarende til om d ligger på den ene eller anden side af planet som trekant abc ligger i. Yderligere er krydsproduktet bestemmende for hvilken siden af planet der opfattes som det positive halvplan.



Figur A.2: Geometrisk betydning af trippel skalar produkt.

Ligning A.8 viser trippel skalar produktet der knytter sig til figur A.2.

$$[a, b, c, d]' = \vec{ad} \cdot (\vec{ab} \times \vec{ac}) \quad (\text{A.8})$$

Ligning A.8 kan omskrives til

$$[a, b, c, d]' = \begin{pmatrix} d_x - a_x \\ d_y - a_y \\ d_z - a_z \end{pmatrix} \cdot \left(\begin{pmatrix} b_x - a_x \\ b_y - a_y \\ b_z - a_z \end{pmatrix} \times \begin{pmatrix} c_x - a_x \\ c_y - a_y \\ c_z - a_z \end{pmatrix} \right) \quad (\text{A.9})$$

Udregning af krydsproduktet giver

$$[a, b, c, d]' = \begin{pmatrix} d_x - a_x \\ d_y - a_y \\ d_z - a_z \end{pmatrix} \cdot \begin{pmatrix} (b_y - a_y)(c_z - a_z) - (b_z - a_z)(c_y - a_y) \\ (b_z - a_z)(c_x - a_x) - (b_x - a_x)(c_z - a_z) \\ (b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x) \end{pmatrix} \quad (\text{A.10})$$

Endeligt giver udregningen af prikproduktet

$$\begin{aligned} [a, b, c, d]' &= b_x c_y d_z - a_x c_y d_z - b_y c_x d_z + a_y c_x d_z + a_x b_y d_z - a_y b_x d_z - b_x c_z d_y + a_x c_z d_y \\ &+ b_z c_x d_y - a_z c_x d_y - a_x b_z d_y + a_z b_x d_y + b_y c_z d_x - a_y c_z d_x - b_z c_y d_x + a_z c_y d_x \\ &+ a_y b_z d_x - a_z b_y d_x - a_x b_y c_z + a_y b_x c_z + a_x b_z c_y - a_z b_x c_y - a_y b_z c_x + a_z b_y c_x \end{aligned} \quad (\text{A.11})$$

Betragtes udtrykkene A.7 og A.11 ses det at

$$[a, b, c, d]' = [a, b, c, d] \quad (\text{A.12})$$

Hermed kan det konkluderes at de to metoder er ækvivalente.

Bilag B

Installation af Fedora release 7 på Playstation 3

B.1 Introduktion

En online udgave af denne guide kan findes på:

http://idun.homelinux.net/speciale/doku.php?id=fedora_7_pa_ps3

Denne guide er bla. baseret på:

http://www.ibm.com/developerworks/blogs/page/powerarchitecture?entry=ibomb_ps3_sdk30_fc7

<http://www.gnuradio.org/trac/wiki/PS3FC7Install>

B.2 Nødvendig software

Download og brænd en Fedora release 7 DVD. ISO filen kan findes her:

<http://mirrors.fedoraproject.org/publiclist/Fedora/7/ppc/>

Ud over Fedora, skal der også bruges en speciel Playstation 3 Addon CD. Denne indeholder de filer der skal til at starte selve installationen af Fedora. Udover disse indeholder den også en række pakker der skal benyttes efter installationen, herunder en ny kerne til Fedora. Den seneste udgave af Addon CD'en kan findes her:

<http://www.kernel.org/pub/linux/kernel/people/geoff/>

B.3 Forbered PS3'eren på installationen

For at installere linux på PS3'eren er der et par ting der skal gøres. Først og fremmest er det en god ide at opdatere dens firmware til den nyeste version. Dette gøres ved at vælge menu punktet Settings ⇒ System Update i sony's GameOS. Pt. er den nyeste version 2.10.

Herefter skal der gøres plads til Linux. Dette gøres ved at formatere harddisken. Vælg Settings ⇒ System Settings ⇒ Format Utility ⇒ Format Hard Disk. **Vælg herefter en Custom Format og vælg Allot 10GB to the PS3 system. Udfør dernæst en Quick Format. Bemærk at vælges der ikke Quick Format tager formateringen LANG tid!**

Nu er det tid til at installere bootladeren, der skal load linux. Indsæt Addon CD'en, og vælg menuen Settings ⇒ System Settings ⇒ Install Other OS. Følg herefter instruktionernern. For at få PS3'erne til at starte den nye bootlader istedet for GameOS, vælges menuen Settings ⇒ System

Settings ⇒ Default System, herefter vælges Other OS. Bemærk at det er muligt at få PS3'ernen til at startes GameOS igen ved at holde powerknappen inde under opstart, indtil der høres et bip.

B.4 Installation af Fedora release 7

Nu er det tid til at installere Linux! Tænd for PS3'en og indsæt Fedora DVD'en. Efter et stykke tid dukker bootloaderens (kboot) prompt op. Indtast følgende kommandoer:

```
cd //mnt/root/ppc/ppc64
kexec -f --initrd=ramdisk.image.gz --command-line="video=576i" vmlinuz
```

Bemærk at den valgte video mode forudsætter at PS3'en er tilsluttet et almindeligt fjernsyn. Hvis dette ikke er tilfældet skal der vælges en passende video mode (576p, 576i, 720p, 720i, 1076p eller 1076i). Efter et stykke tid starter X op med anaconda installeren. Selve installationen af Fedora er relativ standard. Der er dog et par ting man skal være opmærksom på:

- Når anaconda siger den ikke kan finde installations DVD'en, vælges Select driver. Herefter vælges Playstation 3 (ps3_storage) på listen.
- Når det kommer til partitionering, så vælg en standard partitionering med LVM.
- Når anaconda spørger hvilke pakker der skal installeres, så vælg kun Software Development og Customize Later.

Anaconda begynder at installere pakker, dette tager LANG tid... Når alle pakker er installeret, skubbes DVD ud, og systemet kan rebootes. Desværre indeholder kernen ikke PS3 support og kan derfor ikke reboot systemet. Hold derfor powerknappen inde indtil systemet slukker, og tænd derefter igen. Herefter bootes linux, og first boot programmet startes. Svar på programmets spørgsmål (root password, opret bruger osv.).

B.5 Byg en kerne med PS3 support

Som allerede nævnt har standard kernen ikke support for PS3, og det er derfor nødvendigt at kompilere en ny kerne.

Skift til en terminal og log ind som root. Indsæt Addon CD'en og mount den:

```
mount -t auto /dev/scd0 /media/
```

Installer herefter de nødvendige RPM pakker:

```
cd /media/target
rpm -ivh --force *.rpm
```

Da vi skal compilere en ny kerne er det nødvendigt at installere kerne kildekoden:

```
cd /usr/src
tar -jxvf /media/src/linux-2.6.xxxx.tar.bz2
ln -s linux-2.6.xxxx linux
umount /media
eject
```

Bemærk at kerne versionen (2.6.xxxx) afhænger af versionen af Addon CD'en. I skrivende stund er versionen 2.6.24.

For at kunne kompilere den nye kerne skal vi installere et par værktøjer:

```
yum install git
git clone git://www.jdl.com/software/dtc.git
cd dtc
make
make install
```

Da systemet benytter LVM skal der installeres support for dette:

```
rpm -Uvh http://david.woodhou.se/lvm2-2.02.24-1.fc7.ps3.ppc.rpm
```

Hvis du benytter et almindeligt fjernsyn skal de næste par trin udføres via en SSH forbindelse, idet fjernsynet ikke har en opløsning der er høj nok.. For at konfigurere kernen til at understøtte LVM gøres følgende:

```
cd /usr/src/linux
make mrproper
cp arch/powerpc/configs/ps3_defconfig .config
make menuconfig
```

I kerne konfigurations programmet gøres følgende valg:

- Device Drivers -->
- [*] Multiple devices driver support (RAID and LVM) -->
- < M> Device mapper support

Herefter lukkes programmet, og vi er klar til at kompilere den nye kerne:

```
make \&\& make modules_install
```

Når kernen er kompileret og modulerne installeret, skal vi have installeret den nye kerne:

```
cd /usr/src/linux
cp vmlinux /boot/vmlinux-2.6.xxxx
cp .config /boot/config-2.6.xxxx
cp System.map /boot/System.map-2.6.xxxx
mkinitrd /boot/initrd-2.6.xxxx.img 2.6.xxxx
```

Igen udskiftes xxxx med det rigtige versions nummer.

For at kunne boote den nye kerne skal yaboot konfigureres. Åben filen yaboot.conf:

```
nano -w /etc/yaboot.conf>
```

og tilret den så den ligner følgende:

```
\# yaboot.conf generated by anaconda

boot=/dev/sda
init-message=Welcome to Fedora! Hit <TAB> for boot options

partition=1
timeout=80
install=/usr/lib/yaboot/yaboot
```

```
delay=5
enablecdboot
enableofboot
enablenetboot
nonvram
mntpoint=/boot/yaboot
usemount

image=/vmlinuz-2.6.xxxx
  label=latest
  read-only
  initrd=/initrd-2.6.xxxx.img
  root=/dev/VolGroup00/LogVol100
  append="video=576i rhgb quiet"

image=/vmlinuz-2.6.21-1.3194.fc7
  label=linux
  read-only
  initrd=/initrd-2.6.21-1.3194.fc7.img
  root=/dev/VolGroup00/LogVol100
  append="video=576i rhgb quiet"
```

Husk igen at skifte xxxx med det rigtige versions nummer. I dette eksempel er den gamle kerne version 2.6.21-1.3194, og dens konfiguration er beholdt i yaboot.conf. På denne måde er det muligt at boote den gamle kerne ved at skrive linux i bootloader prompten, hvis den nye kerne mod forventning ikke virker.

Så er der kun tilbage at reboote systemet:

```
reboot
```

Forhåbenligt skulle den nye kerne gerne boote, og vi er klar til at opdatere systemet med de seneste RPM pakker.

B.6 Opdater pakker

For at sikre at kernen ikke bliver opdateret til en inkompatibel version, ekskluderes denne fra automatisk opdatering via yum. Dette gøres ved at redigere yum konfigurationen:

```
nano -w /etc/yum.conf
```

Linien:

```
exclude=kernel*
```

indsættes så filen kommer at indeholde noget ala dette:

```
[main]
cachedir=/var/cache/yum
keepcache=0
debuglevel=2
logfile=/var/log/yum.log
exactarch=1
obsoletes=1
gggcheck=1
plugins=1
metadata_expire=1800
```

```
exclude=kernel*

\# PUT YOUR REPOS HERE OR IN separate files named file.repo
\# in /etc/yum.repos.d
```

Herefter kan systemet opdateres med kommandoen (Sæt noget kaffe over... Det tager LANG tid):

```
yum update
```

B.7 Spar på hukommelsen

Da PS3 kun har 256 MiB hukommelse, er det en god ide at begrænse hukommelsesforbruget. Følgende kan gøres for at opnå dette.

B.7.1 Undgå X

Skift til runlevel 3, for at disable grafisk login og X:

```
nano -w /etc/inittab
```

Ret linien:

```
id:5:initdefault
```

til:

```
id:3:initdefault
```

B.7.2 Slå unødvendige services fra

Som standard har systemet en masse unødvendige services kørende. For at se hvilke udføres:

```
ntsysv 35
```

Dette program kan også benyttes til at disable uønskede services. Følgende foreslås disabled:

```
anacron
atd
auditd
autofs
avahi-daemon
bluethooth
cpuspeed
cups
dhcdbd
firstboot
gpm
haldaemon
hidd
hplip
ip6tables
iprdump
iprinit
```

```
iprupdate
iptables
isdn
kudzu
mcstrans
mdmonitor
messagebus
netfs
nfslock
pcscd
readahead_early
readahead_later
restorecond
rpcbind
rpcgssd
rpcidmapd
sendmail
setroubleshoot
smartd
xfs
yum-updatesd
```

Følgende services er dem jeg har kørende på min PS3:

```
ConsoleKit
crond
irqbalance
network
sshd
syslog
udev-post
```

Bilag C

Software Development Kit for Multicore Acceleration version 3 på Playstation 3

En online version af denne guide kan findes på:

http://idun.homelinux.net/speciale/doku.php?id=ibmsdk3_ps3

Denne guide beskriver hvorledes IBM Software Development Kit for Multicore Acceleration version 3 (IBMSDK) installeres på Playstation 3. Guiden går ud fra at Fedora release 7 er installeret på playstationen. For en guide til at installere Fedora release 7 på PS3 se appendix B

IBM egen installations dokumentation kan findes her:

<http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/3D3C5B2932876AEB00257353006C179E>

Yderligere info kan findes her:

<http://www-128.ibm.com/developerworks/power/cell/>

C.1 Installation af IBMSDK

I dokumentationen er det nævnt at pakkerne `rsync`, `sed`, `tcl` og `wget` skal installeres før SDK'et installeres. Det ser ud til at disse pakker allerede er installeret med en standard installation af Fedora release 7, men for en sikkerheds skyld kan nedenstående kommando udføres:

```
yum install rsync sed tcl wget
```

Før download af de nødvendige filer oprettes et bibliotek til filerne:

```
mkdir -p /tmp/cellsdkiso  
cd /mnt/cellsdkiso
```

For at downloade de nødvendige filer kræver IBM at man lader sig registrere, dette er dog gratis. Åben siden: <http://www-128.ibm.com/developerworks/power/cell/pkgdownloads.html> og klik på Fedora 7 download page linket. Efter registrerings processen, åbnes download siden. Download filen `cell-install-3.0.0-1.0.noarch.rpm` og placer den i mappen der blev oprette før. (Benyt evt. `wget` til dette: Kopier linket til dowload af pakken og udfør `wget link`, hvor link er det omtalte link). Download ligeledes ISO filerne

`CellSDK-Devel-Fedora_3.0.0.1.0.iso` og `CellSDK-Extras-Fedora_3.0.0.1.0.iso` og placer dem i samme mappe. (Igen kan `wget` benyttes med fordel).

Hvis ikke automatisk opdatering af Fedora via YUM allerede er slået fra, skal denne service stoppes før installationen fortsættes:

```
/etc/init.d/yum-updatesd stop
```

Installer SDK install programmet:

```
rpm -ivh cell-install-3.0.0-1.0.noarch.rpm
```

Installer selve SDK'et:

```
cd /opt/cell
./cellsdk --iso /tmp/cellsdkiso install
```

Husk at acceptere diverse licenser undervejs...

Herefter skulle SDK'et være installeret! For at undgå at YUM overskriver pakker installeret af SDK'et, tilføjes linien:

```
exclude=blas blas-devel
```

til filen etc/yum.conf.

Hvis YUM opdaterings servicen blev stoppet før installationen, startes denne igen:

```
/etc/init.d/yum-updatesd start
```

C.2 Installation af ekstra pakker

SDK installations programmet installere ikke alle pakker som standard. For at se hvilke pakker der er installeret udføres kommandoen:

```
./cellsdk verify
```

På undertegnede system giver dette følge output:

```
repository=CellSDK-Devel-Fedora-cbea
default CellDevelopmentLibraries alf-devel-3.0.0-9.ppc
default CellDevelopmentLibraries alf-devel-3.0.0-9.ppc64
default CellDevelopmentLibraries alfman-3.0-10.noarch
default CellDevelopmentLibraries blas-devel-3.0-6.ppc
default CellDevelopmentLibraries blas-devel-3.0-6.ppc64
default CellDevelopmentLibraries dacs-devel-3.0.0-19.ppc64
default CellDevelopmentLibraries dacsman-3.0-6.noarch
default CellDevelopmentLibraries libspe2man-2.2.0-5.noarch
default CellDevelopmentLibraries ppu-mass-devel-4.5.0-10.ppc
default CellDevelopmentLibraries ppu-mass-devel-4.5.0-10.ppc64
default CellDevelopmentLibraries ppu-simdmath-devel-3.0-5.ppc
default CellDevelopmentLibraries ppu-simdmath-devel-3.0-5.ppc64
default CellDevelopmentLibraries simdman-3.0-6.noarch
default CellDevelopmentLibraries spu-mass-devel-4.5.0-10.ppc
default CellDevelopmentLibraries spu-simdmath-devel-3.0-5.ppc
default CellProgrammingExamples alf-examples-source-3.0.0-7.noarch
default CellProgrammingExamples cell-buildutils-3.0-11.noarch
default CellProgrammingExamples cell-demos-3.0-10.ppc64
default CellProgrammingExamples cell-demos-source-3.0-10.noarch
default CellProgrammingExamples cell-examples-3.0-11.ppc64
default CellProgrammingExamples cell-examples-source-3.0-11.noarch
default CellProgrammingExamples cell-libs-3.0-16.ppc
default CellProgrammingExamples cell-libs-3.0-16.ppc64
```



```

default CellProgrammingExamples cell-libs-devel-3.0-16.ppc
default CellProgrammingExamples cell-libs-devel-3.0-16.ppc64
default CellProgrammingExamples cell-libs-source-3.0-16.noarch
default CellProgrammingExamples cell-tutorial-3.0-6.ppc
default CellProgrammingExamples cell-tutorial-source-3.0-6.noarch
default CellRuntimeEnvironment alf-3.0.0-9.ppc
default CellRuntimeEnvironment alf-3.0.0-9.ppc64
default CellRuntimeEnvironment blas-3.0-6.ppc
default CellRuntimeEnvironment blas-3.0-6.ppc64
default CellRuntimeEnvironment dacs-3.0.0-19.ppc64
default CellRuntimeEnvironment ppu-simdmath-3.0-5.ppc
default CellRuntimeEnvironment ppu-simdmath-3.0-5.ppc64
mandatory CellDevelopmentLibraries cell-documentation-3.0-5.noarch
mandatory CellRuntimeEnvironment cell-devel-license-3.0.0-1.0.noarch
optional CellDevelopmentLibraries alf-trace-devel not installed
optional CellDevelopmentLibraries dacs-trace-devel not installed
optional CellDevelopmentLibraries pdt-devel not installed
optional CellDevelopmentLibraries trace-devel not installed
optional CellDevelopmentTools alf-ide-template not installed
optional CellDevelopmentTools cellide not installed
optional CellPerformanceTools alf-trace not installed
optional CellPerformanceTools dacs-trace not installed
optional CellPerformanceTools fdprpro not installed
optional CellPerformanceTools pdt-module not installed
optional CellPerformanceTools pdt not installed
optional CellPerformanceTools pdtr not installed
optional CellPerformanceTools trace not installed
optional CellProgrammingExamples cell-compliance-tests not installed
optional CellProgrammingExamples cell-compliance-tests-source not
    installed
optional CellRuntimeEnvironment alf-debuginfo not installed
optional CellRuntimeEnvironment dacs-debuginfo not installed
optional CellRuntimeEnvironment simdmath-debuginfo not installed

repository=CellSDK-Extras-Fedora-cbea
mandatory CellDevelopmentLibraries cell-extras-documentation-3.0-5.noarch
mandatory CellRuntimeEnvironment cell-extras-Fedora-license-3.0.0-1.0.
    noarch
optional CellDevelopmentLibraries alf-hybrid-devel not installed
optional CellDevelopmentLibraries alf-hybrid-trace-devel not installed
optional CellDevelopmentLibraries cell-spu-isolation-devel not installed
optional CellDevelopmentLibraries dacs-hybrid-devel not installed
optional CellDevelopmentLibraries dacs-hybrid-trace-devel not installed
optional CellDevelopmentLibraries libfft-devel not installed
optional CellDevelopmentLibraries libmc-rand-devel not installed
optional CellDevelopmentLibraries spu-timer-devel not installed
optional CellDevelopmentTools cell-spu-isolation-tool not installed
optional CellDevelopmentTools cell-spu-isolation-tool-source not
    installed
optional CellDevelopmentTools cell-xlc-ssc-cmp not installed
optional CellDevelopmentTools cell-xlc-ssc-help not installed
optional CellDevelopmentTools cell-xlc-ssc-lib not installed
optional CellDevelopmentTools cell-xlc-ssc-man not installed
optional CellDevelopmentTools cell-xlc-ssc-omp not installed
optional CellDevelopmentTools cell-xlc-ssc-rte-lnk not installed
optional CellDevelopmentTools cell-xlc-ssc-rte not installed
optional CellPerformanceTools alf-hybrid-trace not installed

```

```

optional CellPerformanceTools cellperfctr-tools not installed
optional CellPerformanceTools cell-perf-hybrid-tools not installed
optional CellPerformanceTools cell-spu-timing not installed
optional CellPerformanceTools dacs-hybrid-trace not installed
optional CellProgrammingExamples alf-hybrid-examples-source not installed
optional CellProgrammingExamples cell-spu-isolation-emulated-samples not
    installed
optional CellProgrammingExamples libfft-examples-source not installed
optional CellRuntimeEnvironment alf-hybrid not installed
optional CellRuntimeEnvironment cell-spu-isolation-loader not installed
optional CellRuntimeEnvironment dacs-hybrid not installed
optional CellRuntimeEnvironment libfft not installed
optional CellSimulator systemsim-cell not installed

repository=CellSDK-Open-Fedora-cbea
default CellDevelopmentLibraries libspe2-devel-2.2.0-91.ppc
default CellDevelopmentLibraries libspe2-devel-2.2.0-91.ppc64
default CellDevelopmentLibraries numactl-devel-0.9.10-1.ppc
default CellDevelopmentLibraries numactl-devel-0.9.10-1.ppc64
default CellDevelopmentTools ppu-binutils-2.17.50-32.ppc
default CellDevelopmentTools ppu-gcc-4.1.1-57.ppc
default CellDevelopmentTools ppu-gcc-c++-4.1.1-57.ppc
default CellDevelopmentTools ppu-gdb-6.6.50-28.ppc
default CellDevelopmentTools spu-binutils-2.17.50-33.ppc
default CellDevelopmentTools spu-gcc-4.1.1-107.ppc
default CellDevelopmentTools spu-gcc-c++-4.1.1-107.ppc
default CellDevelopmentTools spu-gdb-6.6.50-12.ppc
default CellDevelopmentTools spu-newlib-1.15.0-82.ppc
default CellRuntimeEnvironment numactl-0.9.10-1.ppc
default CellRuntimeEnvironment numactl-0.9.10-1.ppc64
mandatory CellRuntimeEnvironment elfspe2-2.2.0-91.ppc
mandatory CellRuntimeEnvironment libspe-1.2.2-2.ppc
mandatory CellRuntimeEnvironment libspe-1.2.2-2.ppc64
mandatory CellRuntimeEnvironment libspe2-2.2.0-91.ppc
mandatory CellRuntimeEnvironment libspe2-2.2.0-91.ppc64
optional CellDevelopmentLibraries libspe2-adabinding-devel not installed
optional CellDevelopmentTools crash-spu-commands-debuginfo not
    installed
optional CellDevelopmentTools crash-spu-commands not installed
optional CellDevelopmentTools ppu-binutils-debuginfo not installed
optional CellDevelopmentTools ppu-gcc-debuginfo not installed
optional CellDevelopmentTools ppu-gcc-fortran not installed
optional CellDevelopmentTools ppu-gcc-gnat not installed
optional CellDevelopmentTools ppu-gdb-debuginfo not installed
optional CellDevelopmentTools spu-binutils-debuginfo not installed
optional CellDevelopmentTools spu-gcc-debuginfo not installed
optional CellDevelopmentTools spu-gcc-fortran not installed
optional CellDevelopmentTools spu-gdb-debuginfo not installed
optional CellDevelopmentTools spu-newlib-debuginfo not installed
optional CellDevelopmentTools spu-tools-debuginfo not installed
optional CellDevelopmentTools spu-tools not installed
optional CellPerformanceTools oprofile-0.9.3-4bsc.ppc
optional CellPerformanceTools oprofile-debuginfo-0.9.3-4bsc.ppc
optional CellRuntimeEnvironment libspe2-debuginfo not installed
optional CellSimulator sysroot_image not installed

```

C.2 Installation af ekstra pakker

For at installere enkelte pakker kan YUM benyttes. For at det kan lade sig gøre skal de downloadede ISO filer først mountes, så YUM kan finde RPM pakkerne. Dette kan gøres vha. SDK installations programmet:

```
cd /opt/cell
./cellsdk --iso /tmp/cellsdkiso mount
```

Herefter kan enkeltpakker installeres med kommandoen:

```
yum install pakkenavn
```

Som et eksempel ses her hvordan forskellige performance tools kan installeres:

```
yum install alf-trace-devel pdt pdt-devel alf-trace fdprpro pdtr trace alf-
debuginfo spu-timer-devel cell-spu-timing oprofile
```

Når installationen er færdig kan ISO filerne unmountes

```
./cellsdk --iso /tmp/cellsdkiso unmount
```

Bilag D

GPU timing testdata

Dette bilag indeholder scripts og testdata til brug i timings analysen af GPU'en.

D.1 Test scripts

D.1.1 Memory tests

```
1 #!/bin/awk -f
2 # Convert bandwidth in MB/s to elapsed time from the bandwidthTest program from
   CUDA 1.1SDK
3 # TODO: Determine accuracy, the program only reports one decimal for MB/s
4 /^ +[0-9]+\t+[0-9]+\.[0-9]/ \
5     { \
6         bytes=$1;\
7         bandwidth=$2;\
8         MEGA=1048576;\
9         FACTOR=1e3; \
10        print bytes, (1/bandwidth)*(bytes/(MEGA)) * FACTOR, "" \
11    }
```

```
1 #!/bin/awk -f
2 BEGIN {avgsum=0.0} \
3 {\
4     #for (i = 1; i<=NF; i++) \
5         avgsum+=$2;\
6         data[NR]=$2;\
7 } \
8     END { \
9         average=avgsum/NR; \
10
11        max=-999999999; \
12        min=999999999; \
13        stdsum = 0.0; \
14        for (x in data) { \
15            stdsum += ( (data[x] - average) * (data[x] - average) ); \
16            if (data[x] > max) \
17                max = data[x];\
18            if (data[x] < min)\
19                min = data[x];\
20        } \
```

```

21
22     variance = stdsum / NR; \
23     std = sqrt( variance ); \
24     \
25     print "average: " average ; \
26     print "min: " min; \
27     print "max: " max; \
28
29     print "variance: " variance; \
30     print "std: " std; \
31 }

```

```

1 #!/bin/awk -f
2 /^[0-9]+\ [0-9]+\.[0-9]+/ \
3     { \
4         print $1,$2 * 2.0;
5     }

```

```

1 set title 'Memory transfer timing test'
2 set xlabel 'Size (bytes)'
3 set ylabel 'Time (msec)'
4 #set xtics 2048000
5
6 plot '<./bandwidth2time.awk dtoh.bandwidth' ti "Device to Host" with linespoints,
7     \
8     '<./bandwidth2time.awk dtod.bandwidth | ./times2.awk' ti "Device to Device" with
9     linespoints, \
10    '<./bandwidth2time.awk htod.bandwidth' ti "Host to Device" with linespoints
11
12 #set term postscript enhanced color
13 #set output "memorytimeplot.ps"
14 #replot

```

D.2 Memory

Resultaterne af timings testene er blevet analyseret og fittet til et 1. gradspolynomie med Gnuplot. De relevante parametre er her a og b , der indgår i et polynomiet som

$$T(x) = ax + b \quad (\text{D.1})$$

De efterfølgende sektioner er resultater af analysen af timingsdataene.

D.2.1 Low-range transfer size

Device-host

```

1 average: 0.0137938
2 min: 0.0131
3 max: 0.0143
4 variance: 7.80859e-08
5 std: 0.000279439
6

```

```

7 After 7 iterations the fit converged.
8 final sum of squares of residuals : 8.10423e-07
9 rel. change during last iteration : -1.04256e-13
10
11 Final set of parameters           Asymptotic Standard Error
12 =====
13 a             = 4.32692e-07      +/- 2.418e-08      (5.587%)
14 b             = 0.0133438       +/- 2.892e-05     (0.2167%)

```

Host-device

```

1 average: 0.0145219
2 min: 0.0137
3 max: 0.0152
4 variance: 7.6084e-08
5 std: 0.000275833
6
7 After 6 iterations the fit converged.
8 final sum of squares of residuals : 3.3023e-06
9 rel. change during last iteration : -1.03847e-09
10
11 Final set of parameters           Asymptotic Standard Error
12 =====
13 a             = 2.64709e-07      +/- 4.88e-08      (18.44%)
14 b             = 0.0142466       +/- 5.838e-05     (0.4098%)

```

Device-device

```

1 average: 0.00853125
2 min: 0.0052
3 max: 0.0124
4 variance: 1.12812e-05
5 std: 0.00335875

```

D.2.2 Mid-range transfer size

Device-host

```

1 average: 0.024757
2 min: 0.0136
3 max: 0.036
4 variance: 4.25285e-05
5 std: 0.00652138
6
7 After 8 iterations the fit converged.
8 final sum of squares of residuals : 6.42993e-07
9 rel. change during last iteration : -8.15597e-10
10
11 Final set of parameters           Asymptotic Standard Error
12 =====
13 a             = 4.41214e-07      +/- 5.481e-10     (0.1242%)
14 b             = 0.013349        +/- 1.632e-05     (0.1223%)

```

Host-device

```

1 average: 0.023515
2 min: 0.0145

```

```

3 max: 0.0329
4 variance: 2.95017e-05
5 std: 0.00543154
6
7 After 7 iterations the fit converged.
8 final sum of squares of residuals : 1.05218e-05
9 rel. change during last iteration : -7.31673e-08
10
11 Final set of parameters          Asymptotic Standard Error
12 =====
13 a              = 3.66851e-07      +/- 2.217e-09      (0.6043%)
14 b              = 0.0140297       +/- 6.603e-05     (0.4706%)

```

Device-device

```

1 average: 0.006203
2 min: 0.0052
3 max: 0.012
4 variance: 6.23091e-07
5 std: 0.000789361
6
7 After 7 iterations the fit converged.
8 final sum of squares of residuals : 4.52356e-05
9 rel. change during last iteration : -2.68455e-06
10
11 Final set of parameters          Asymptotic Standard Error
12 =====
13 a              = 2.79579e-08      +/- 4.597e-09     (16.44%)
14 b              = 0.00548012      +/- 0.0001369    (2.498%)

```

D.2.3 High-range transfer size

Device-host

```

1 average: 2.30288
2 min: 0.0586
3 max: 4.5423
4 variance: 1.70568
5 std: 1.30602
6
7 After 3 iterations the fit converged.
8 final sum of squares of residuals : 0.00397658
9 rel. change during last iteration : -2.75004e-08
10
11 Final set of parameters          Asymptotic Standard Error
12 =====
13 a              = 4.4367e-07       +/- 2.155e-10     (0.04857%)
14 b              = 0.00548012      +/- 0.001284     (23.42%)

```

Host-device

```

1 average: 1.93436
2 min: 0.053
3 max: 3.8115
4 variance: 1.19893
5 std: 1.09495
6

```

```
7 After 3 iterations the fit converged.
8 final sum of squares of residuals : 0.0044296
9 rel. change during last iteration : -3.31778e-08
10
11 Final set of parameters           Asymptotic Standard Error
12 =====
13 a             = 3.7237e-07        +/- 2.274e-10    (0.06108%)
14 b             = 0.00548012       +/- 0.001355    (24.72%)
```

Device-device

```
1 average: 0.144643
2 min: 0.0093
3 max: 0.3078
4 variance: 0.00636726
5 std: 0.0797951
6
7 After 3 iterations the fit converged.
8 final sum of squares of residuals : 0.00416064
9 rel. change during last iteration : -9.16399e-09
10
11 Final set of parameters           Asymptotic Standard Error
12 =====
13 a             = 2.69101e-08       +/- 2.204e-10    (0.8192%)
14 b             = 0.00548012       +/- 0.001313    (23.96%)
```