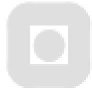

NTNU
Norges teknisk-naturvitenskapelige universitet
Institutt for marin teknikk

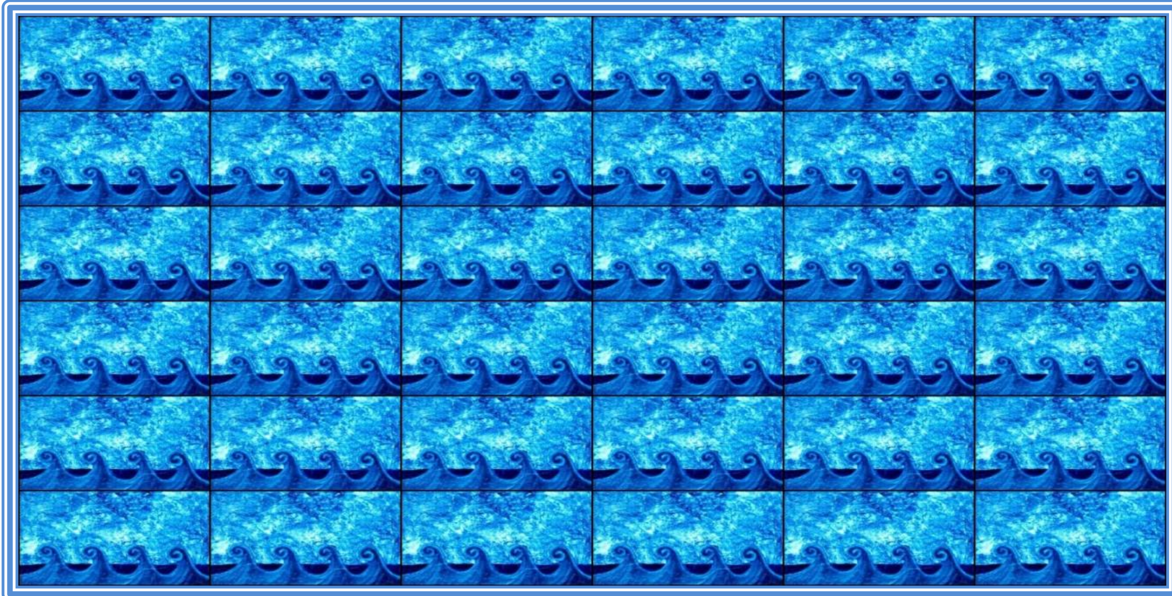
Hydrodynamiske beregninger vha GPU

MASTEROPPGAVE

Av:

Jørgen Haavind Jensen

Vår 2010



Veileder: Håvard Holm

Postadresse:
NTNU
Inst. for marin teknikk
N-7491 Trondheim

Besøksadresse:
Marinteknisk Senter
O. Nielsens vei 10

Telefon +47 73 595501
Telefaks +47 73 595697

Innhold

Abstrakt 2
Oppgavetekst 3
Anerkjennelser 4
Introduksjon til GPGPU 5
CUDA 7
  Konseptuell introduksjon til CUDA programmering 7
Hvordan få god ytelse i CUDA 9
  (1) Maksimere parallell utførelse 9
  (2) Optimalisere minnetilgang 9
  (3) Optimalisere instruksjoner og kontrollflyt 12
Lattice Boltzmann 13
Visualisering 16
  Stereoskopisk 3D 16
  Generering av isoflater 19
Lattice Boltzmann på GPU 23
  (1) Maksimere parallell utførelse 23
  (2) Optimalisere minnetilgang 24
  (3) Optimalisere instruksjoner og kontrollflyt 28
Ytelse 29
Validering av koden 33
  Reynoldstallregimer 34
  Strouhaltall 37
Konklusjon 41
Litteraturliste 42
Appendiks A - D3Q19 Lattice konstanter 43
Appendiks B - Minneoptimering 44
Appendiks C - CUDA Kildekode 48
Appendiks D - Visualisering Kildekode 59

Abstrakt

Om man skal gjøre prediksjoner av hydrodynamiske egenskaper til større marine strukturer i dag, er den vanligste måten fremdeles modellforsøk, empiri og lineærteori. Viskøse CFDsimuleringer ved hjelp av RANSmetoder har begynt å gi brukbare resultater, men en direkte Navier-Stokesløser for de Reynoldstall man typisk støter på i større marine applikasjoner, er enda langt framme i tid, både med tanke på beregningskraft og minne.

General Purpose GPU har i den siste tiden vist lovende resultater for å øke ytelsen på aritmetisk tunge beregninger. Samtidig har Lattice-Boltzmann-metode for løsing av Navier-Stokes ligninger, i kraft av sin høye parallelliserbarhet, vist seg godt egnet for implementering på GPU. Ved å se på muligheten for å gjøre Lattice-Boltzmann-beregninger over flere GPUer samtidig, håper jeg å ta oss ett steg nærmere en fullskala løser for Navier-Stokes, løse noen nyttige problemer på veien dit, og samtidig få lært meg mer om GPGPUprogrammering som jeg tror vil bli et svært nyttig verktøy for framtidens ingeniør.

Oppgavetekst

MASTEROPPGAVE I MARIN TEKNIKK

VÅR 2010

FOR

STUD. TECHN. Jørgen Haavind Jensen

Hydrodynamiske beregninger vha GPU

- Kandidaten skal sette seg inn i Lattice Boltzmann metoden, med vekt på 3D.
- Kandidaten skal implementere en tredimensjonal Lattice-Boltzmann løser til å utnytte GPUen.
- Ytelse skal dokumenteres.
- Dersom tiden tillater det, skal strømning rundt marin applikasjon anvendes til å validere koden.
- Dersom tiden tillater det, skal strømningen visualiseres.

Det forutsettes at Institutt for marin teknikk, NTNU, fritt kan benytte resultatene i sitt forskningsarbeid, da med referanse til studentens arbeid.
Besvarelsen redigeres mest mulig som en forskningsrapport med resymé, konklusjon, litteraturliste etc. Ved utarbeidelsen av teksten skal kandidaten legge vekt på å gjøre den så kort, oversiktlig og velskrevet som mulig.

Besvarelsen leveres i to eksemplar innen 14. juni 2010.

Håvard Holm

Førsteamanuensis, institutt for marin teknikk, NTNU

Anerkjennelser

Takk til Håvard Holm for godt råd om oppgavevalg, og god veiledning underveis.

Takk til Anne C. Elster og alle på HPC-laben for behjelpelighet med lån og oppsett av utstyr samt hendige tips under utviklingsprosessen.

Sist men ikke minst takk til alle som har korrekturlest for meg, og luket ut de mange små og store feil jeg selv var for blind til å se.

Introduksjon til GPGPU

Den 7 september 2004 kom versjon 2.0 av openGL (**open G**raphics **L**ibrary). Det var den første programmeringsAPIen for 3D-grafikk som hadde støtte for brukerprogrammerte piksel/vertex-shadere. Tidligere hadde 3D-grafikk vært begrenset til noen få klart definerte operasjoner som var hardkodet inn i grafikkprosessoren (GPU). Nå sto man fritt til å definere sine egne lys-, farge- og koordinatmodeller ved hjelp av en enkel C-lignende kode. Dette åpnet for noen muligheter som skaperne av openGL kanskje ikke hadde forutsett.

Ved å betrakte fargedata som vanlige tall, og bilder/teksturer/rammebuffere som datatabeller kunne man bruke openGL til å løse differensialligninger, invertere matriser++.

3D-grafiske beregninger har følgende egenskaper som har vært styrende for designet av GPUer:

1. Fargen på ett enkelt bildeelement (piksel) er uavhengig av fargen på andre piksler i bildet.
2. Antallet piksler som skal oppdateres er i størrelsesorden 1 million.
3. Det vil som regel være mange piksler som skal beregnes etter samme lys-/fargemodell, og disse pikslene vil ofte ligge inntil hverandre.

Dette betyr at fargen på hver piksel kan beregnes i parallell uten noen form for kommunikasjon. GPUer har derfor et svært høyt antall prosessorkjerner, samt en svært høy overføringshastighet mellom RAM og GPU. Det betyr videre at flere av kjernene i GPUen vil kjøre akkurat det samme programmet, på data som ligger etter hverandre i Random Access Memory (RAM). GPUer kjører derfor akkurat de samme instruksjonene på flere av kjernene samtidig og overfører lineære blokker av data til/fra RAM i grupper.

Siden man ikke har interkjernekommunikasjon, trenger man ikke sette av plass eller transistorer på grafikkchipen til denne oppgaven, og siden instruksjoner og minneoverføringer skjer i grupper, kan flere kjerner dele instruksjons- og minnekontrollenheter.

For å redusere antall nødvendige transistorer enda mer, er gjerne kjernene på en GPU mye enklere enn på en vanlig prosessor (CPU). CPUer har etter hvert begynt å stange mot gigahertzbarrieren for ytelse. For å møte dette problemet har man gjerne avanserte enheter for styring av hurtig cacheminne for å senke latenstid for minneaksesser, og branch-prediktorer som "gjetter" hvilken vei programflyten vil gå for å slippe å kaste bort verdifulle klokkesykluser mens man venter på resultatet av flytstyrende beregninger. GPUer har en litt annen tilnærming til dette problemet. En vanlig dataskjerm i dag har en oppløsning på f.eks. 1680x1050. Det blir over 1.7 millioner punkter som skal oppdateres, og som i prinsippet kan oppdateres i parallell av hver sin uavhengige programtråd. En GPU har langt under 1.7 millioner kjerner, så isteden har hver kjerne et stort antall tråder av gangen som står på vent. Om tråden som for øyeblikket kjøres må vente på data fra RAM, bytter kjernen kjapt og uanstrengt over til en annen tråd som er klar til å kjøre akkurat da.

Alt dette gjør at man kan få plass til langt flere kjerner på en chip uten at faktorer som overoppheting og produksjonskostnad blir uoverkommelig. Dagens GPUer har i størrelsesorden 500 kjerner, en minnebåndbredde på nesten 200 GB/s og en flyttallsytelse på over 1 Tflop. Til rundt samme pris får man en vanlig prosessor med 4 kjerner, 25 GB/s minnebåndbredde og under 100 Gflop ytelse.

Det betyr ikke at GPUen er CPUen overlegen. For problemer med avanserte datastrukturer, uforutsigbar kontrollstruktur og resultatavhengighet er CPUen fremdeles best. På problemer med relativt store regulære datastrukturer, høy grad av parallellisering og en SIMD-natur (Same Instruction Multiple Data) vil GPUen som regel vinne. Et annet aspekt som skiller CPU og GPU, er at GPUer hovedsaklig regner med enkeltpresisjons-flyttall (32 bit). Dette har vært ansett som tilstrekklig nøyaktighet for grafikk, men det er ikke nødvendigvis nok for vitenskapelige beregninger. Forrige generasjons grafikkort fra NVIDIA hadde kun støtte for dobbeltpresisjons flyttall på en av åtte kjerner. Dette er dog i ferd med å bedre seg, og på nyere kort fra NVIDIA (400 serien) er tallet steget til en av to. Min spådom er at man på sikt vil få dedikerte GPGPUkort med full dobbeltpresisjonsytelse. RAM er også et problem. De nyeste grafikkortene i dag har mellom 1 og 2 GB ram, mens man finner knapt noen PC overhodet med så lite systemminne. 1 GB RAM er nok til vanlig spillgrafikk, men det blir fort for lite når man skal løse større beregninger. Det finnes grafikkort som er beregnet på GPGPU, med opptil 6 GB ram, men disse er for øyeblikket flere ganger dyrere enn vanlige forbrukerkort.

De første til å utnytte kraften i GPUen til noe annet enn grafikkberegninger benyttet da altså openGL, en API som er designet først og fremst med tanke på 3D grafikk. Når man skal løse mere generelle problemer fører det gjerne til obfuskert kode som er vanskelig å tolke. Etter hvert som bransjen har sett verdien av GPGPUprogrammering har det kommet nye APIer som er beregnet for nettopp GPGPU.

- **openCL** (open Computing Language), er en åpen standard for generell parallellprogrammering, ikke bare GPGPU. Den kan brukes for å programmere både for ATI's og NVIDIA's grafikkort.

- **Direct Compute** er Microsofts standard for GPGPU, og bygger på directX11. Den kan programmere for alle directX10 og 11 kompatible enheter

- **CUDA** (Compute Unified Device Architechture) er NVIDIAs egen standard. Den fungerer kun med NVIDIAS egne kort.

- **Firestream** er ATI's egen standard, og kan kun brukes med ATIs egne kort.

CUDA

I denne oppgaven benyttes NVIDIAS proprietære standard for GPGPUprogrammering. CUDA har den bakdelen at den kun kan brukes med NVIDIA kort, men med det følger at den også er spesifikt designet opp mot NVIDIA hardware. Man har f.eks. direkte kontroll over en liten mengde level 1 cache som finnes på hver eneste multiprosessor.

CUDA har samme syntaks som C, men med noen begrensinger, blant annet at rekursive funksjoner og funksjonspekere ikke er tillatt.

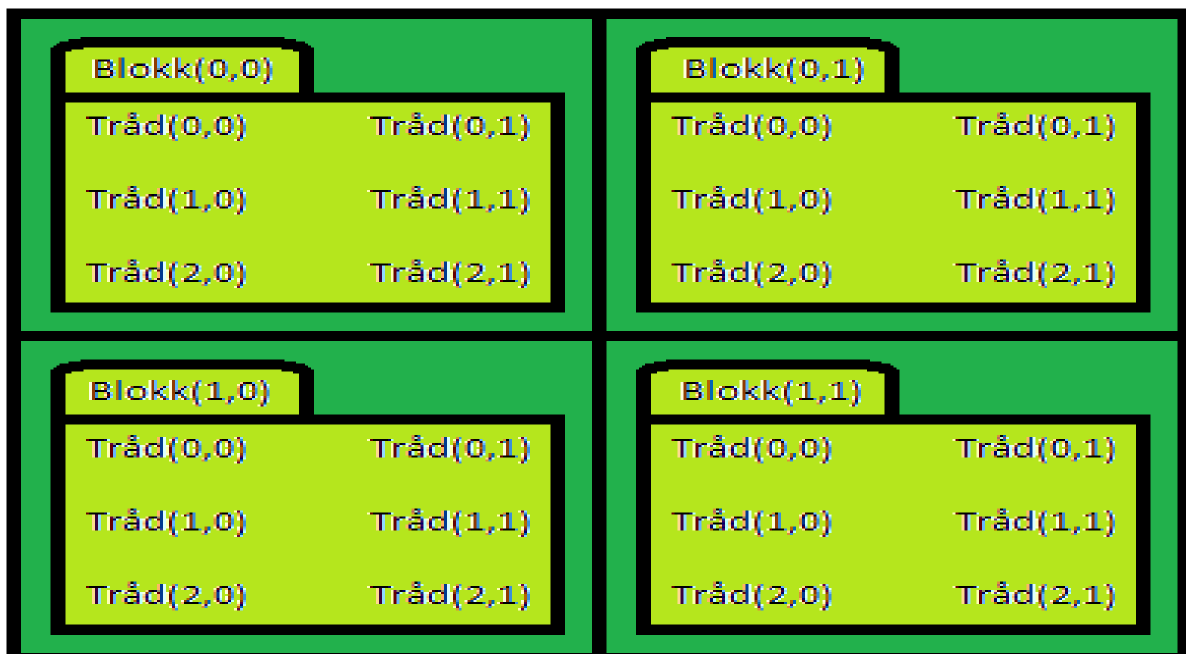
Konseptuell introduksjon til CUDAprogrammering

CUDAfunksjoner kalles *kernels* og et kall til en CUDAfunksjon genererer typisk tusenvis av tråder som alle kjører samme kernel. En kernel kalles fra et program som kjøres fra CPU, og CPUen bestemmer da hvor mange tråder som skal kjøres.

Tråder kan organiseres i to nivåer som vist i Figur 1. Blokk, og grid. Ett grid består av flere blokker. En blokk består av flere tråder.

En blokk holder på et konseptuelt en-, to- eller tredimensjonalt rutenett av tråder. De individuelle sidestørrelsene på blokken kan settes fritt mellom 512x512x64, men blokken kan ikke inneholde flere enn 512 tråder til sammen. Hver tråd har sin egen indeks som forteller dens relative posisjon innad i blokken.

Et grid er en endimensjonal eller todimensjonal samling av blokker. Alle blokkene i gridet er like store. Maksimal størrelse av gridet er 65535x65535. Hver tråd innad i en blokk har tilgang på en indeks som forteller hvilken blokk den hører til.



Figur 1: Konseptuelt diagram over grid/blokk/tråd hierarki i CUDA. 3x2 blokker i et 2x2 grid

For å minimere problemer med minnelatens har som nevnt hver kjerne mange tråder stående på vent til enhver tid. Om gjeldende tråd må vente på data fra GPURAM settes denne på vent, og en ny tråd som er klar til kjøring tar over. Dette får to viktige konsekvenser. Den ene har med ytelse å gjøre og vil diskuteres i et senere kapittel. Det andre har med konsistens å gjøre. Siden man ikke har noen kontroll over rekkefølgen på trådene må programmet skrives slik at resultatet er uavhengig av rekkefølgen.

Om man ikke tenker på ytelse, er organiseringen av blokker og grid, samt problematikken rundt trådrekkefølge alt man trenger å vite om for å begynne å programmere i CUDA. Nedenfor følger et helt enkelt eksempelprogram. Programmet allokerer først en tabell i GPURAM, og så kjøres en kernel som setter en verdi til alle elementene i tabellen.

```
__global__ void myCudaKernel(int* tabell){
      /*
      ### CUDA kernel som kjøres av flere uavhengige tråder på GPU.    ###
      ### Metodenavn med prefixet __global__ er CUDA kernels.          ###
      ### Denne metoden setter verdien i tabell[i] = i^2               ###

            threadIdx.x er nummeret på tråden
            blockIdx.x er nummeret på blokken
            blockDim.x er størrelsen på blokken
            tabell er referanse til en minneadresse i GPU RAM
      */
      int i = threadIdx.x;
      int j = blockIdx.x;
      tabell[i+j*blockDim.x] = pow(i+j*blockDim.x,2.0);
}

int main(int argc, char* argv[]){
      int gridSize = 100;
      int blockSize = 512;
      int n = gridSize*blockSize;

      int* tabell;
      cudaMalloc((void**)&tabell,sizeof(int)*n); // alloker minne på GPU
      myCudaKernel<<<gridSize,blockSize>>>(tabell);// kall GPU kernel
      cudaFree(tabell);                          // frigjør allokert GPU minne

}
```

Hvordan få god ytelse i CUDA

Som nevnt i kapitlet "Introduksjon til GPGPU" er det mange store forskjeller i designet på en CPU og en GPU. Disse forskjellene får store innvirkninger på hvordan man skriver effektiv kode, og på hvilke problemer som er egnet for løsing på GPU.

NVIDIAs egen best practices guide**[3]** for CUDA oppgir tre ting som særskilt viktige for å få god ytelse:

(1) Maksimere parallell utførelse

Å maksimere parallell utførelse betyr noe litt annet på GPU enn man kanskje er vant til. For å maksimere parallell utførelse bør man tenke på at hver GPU har hundrevis av kjerner, og at hver kjerne trenger flere tråder av gangen for å fungere optimalt. Tenk størrelsesorden titusenvis av tråder før alt går optimalt. De største problemene jeg har løst på en enkelt GPU hadde 30 millioner tråder, og det er ingenting i veien for å ha flere. Der tradisjonell parallellisering betyr for eksempel at hver tråd tar hver sin del av en for-løkke, er det i GPGPUkontekst mer naturlig å ha en tråd per iterasjon i løkka.

(2) Optimalisere minnetilgang

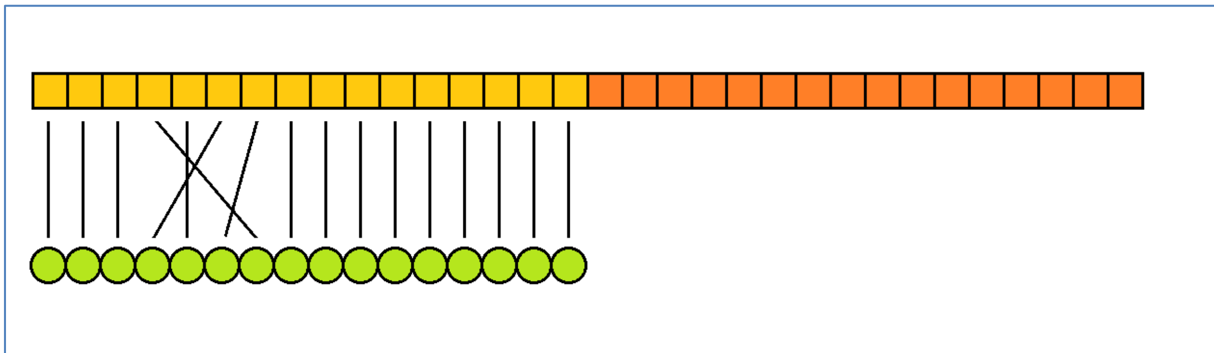
Lesing og skriving til minne er en viktig faktor for ytelse i CUDA. For det første bør slike minneaksesser gjøres minst mulig. For det andre, når det først gjøres, bør det gjøres riktig. Forskjellen mellom helt korrekt og helt feil utgjør en stor ytelsesforskjell. Så lenge man har et problem som kan tilstrekkelig parallelliseres for bruk av GPU, er minnebruk oppgitt som den typisk viktigste parameteren å optimalisere for.

Minne-coalescing

Tråder i CUDA er delt inn grupper på 32. En slik gruppe med 32 konsekutive tråder kalles en *warp*. I tillegg er hver warp delt inn *half-warps* på 16 og 16 tråder. All minneaksess til GPURAM skjer i grupper på half-warps. I beste fall kan alle trådene i en half-warp gjøre en minneaksess i en enkelt overføring, dette er definert som coalescing og er det man alltid prøver å oppnå. I verste fall trengs 16 separate overføringer. Om det blir det ene eller det andre avhenger litt av hvilket skjermkort man bruker og mye av minneaksessmønsteret. Minnemodellen som beskrives er for GPUer av compute capability 1.3, siden alle nyere kort av interesse oppfyller dette kravet. Om kortet har lavere compute capability gjelder noen andre begrensinger i tillegg.

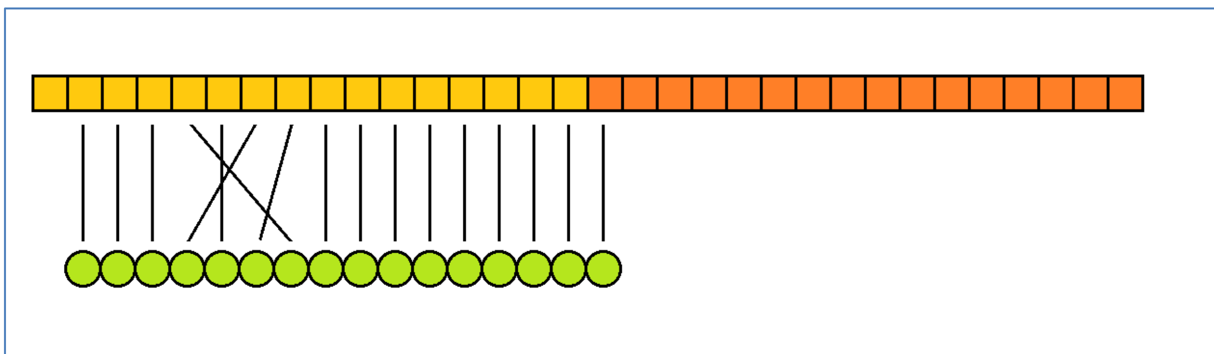
RAM i CUDA er gruppert inn i segmenter på 128 byte. Dette tilsvarer 16 dobbeltpresisjons flyttall. CUDA overfører data i slike segmenter av gangen. Segmentene kan deles videre inn i to 64-byte-segmenter, og om alle minneaksesser i en half-warp er fra enten øvre eller nedre halvdel kan CUDA overføre kun denne ene delen. Denne delen kan *igjen* deles i to 32-byte-segmenter, og om alle aksessene er fra enten øvre eller nedre halvdel, kan også den blokken deles i to. Merk at om minneaksessen krysser midtgrensen for 128-byte-segmentet kan man ikke dele opp overførselen i to 32-byte-segmenter. Om tråder fra samme half-warp leser fra flere ulike 128-byte-segment, ekspederes først trådene fra det ene segmentet etter halveringsreglene som beskrevet, og så tas trådene fra neste segment etter samme mønster. Med unntak av denne segmentlokaliteten er det ingen fordel å hente på å lese data som er nært i minnet. Om man leser fra to ulike segmenter har det ingenting å si om disse segmentene ligger nært eller langt unna hverandre. Unntaket er om man bruker teksturer til å lagre data.

Den tradisjonelle oppgaven til en GPU er tross alt grafikk, og det er også muligheter for å bruke teksturenhetene til å oppnå høyere minnebåndbredde. Moderne GPUer har støtte for både 2D- og 3D-teksturer. Disse har en struktur som gjør dem optimalisert for lesing av data i romlig nærhet i 2D- eller 3D-rom. Om man lagrer dataene i teksturer, kan man oppnå båndbredder tett opp mot maksimum, om dataene man leser ligger nær hverandre i rommet. Bakdelen er at man ikke kan bruke dobbeltpresisjonsflyttall med teksturer. Teksturer er i utgangspunktet ment for å holde fargedata, og for fargedata er enkelpresisjon regnet som tilstrekkelig.



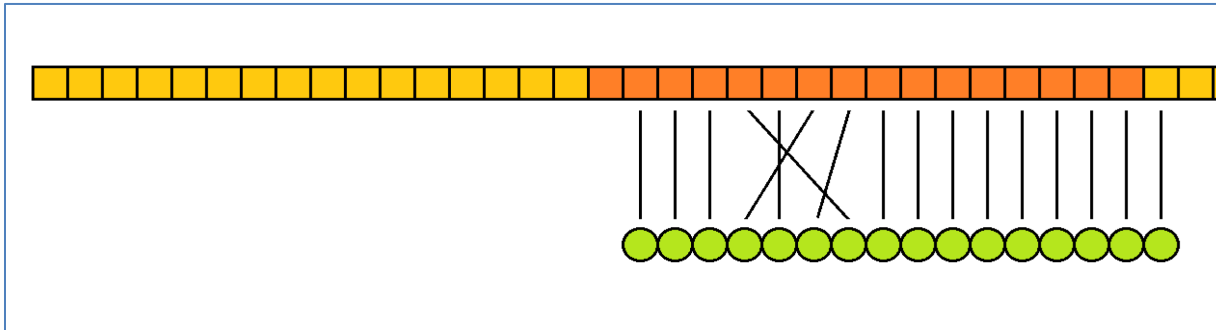
Figur 2: Perfekt 64-byte-minneaksess

Figur 2 viser en perfekt minneaksess. De grønne sirklene representerer en tråd hver, til sammen utgjør de en half-warp, og de gule/oransje firkantene representerer 4 byte med data hver. De gule firkantene er de første 64 bytes i et 128-byte-segment, mens de oransje er de siste 64 bytes. I Figur 2 er alle minneaksessene i nedre halvdel av blokken, og 64 byte overføres på en gang. Merk at tråder kan lese på kryss og tvers av hverandre internt i minnesegmentet uten ytelsesreduksjon.



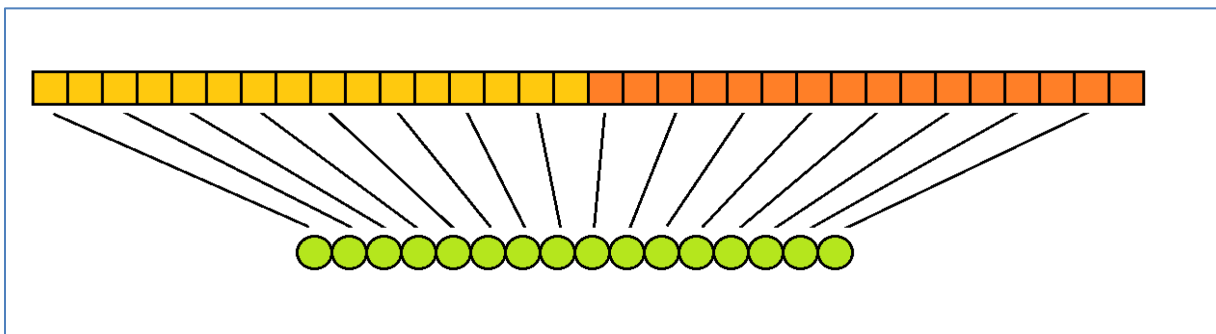
Figur 3: 64-byte-minneaksess med 1 offset

Figur 3 viser en minneaksess der leseadressen er forskjøvet med én. Alle trådene unntatt en, leser fra nedre halvdel, mens en leser fra øvre halvdel. I dette scenarioet må alle 128 byte overføres, og halvparten av minnebåndbredden vil være bortkastet.



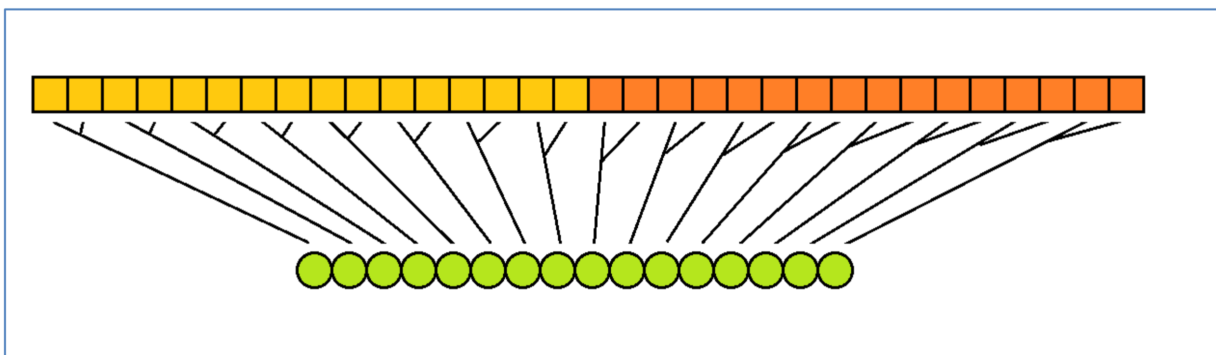
Figur 4: 64-byte-minneaksess med 17 offset

Figur 4 viser en minneaksess der leseadressen er forskjøvet med 17. Aksessen leser data fra øvre halvdel av et 128-byte-segment, samt en aksess i neste segment. Her overføres de siste 64 byte fra første segment, og de første 32 byte fra neste segment. 1/3 av minnebåndbredden vil da være bortkastet. Merk at Figur 4 viser den naturlige fortsettelsen av Figur 3 i en situasjon der flere half-warps leser konsekutive adresser med en offset på en. Den gjennomsnittlige båndbredden i en slik situasjon vil da være (1/2 + 2/3)/2 = 7/12 = 58% av maksimal båndbredde. I tilfellet der offseten er 8, vil half-warper med partallsnummer fremdeles trenge et helt 128-byte-segment, men oddetalls-warper kan deles i to 32-byte-segmenter og gjennomsnittlig båndbredde blir (1/2+1)/2 = 3/4. For alle andre offsetverdier mellom 1 og 15 blir verdien 58%.



Figur 5: 64-byte-minneaksess med stride

Figur 5 viser en 64-byte-minneaksess med stride på 2, altså at hver tråd hopper over en adresse etter forrige. Med en stride på 2, overføres 128 byte, og med 64 byte nyttedata gir dette en båndbredde på 1/2. Om striden er 3, overføres en blokk på 128 byte, og en på 64 og man oppnår 1/3 båndbredde. Stride 4 gir 1/4, stride 5 gir 1/5 osv.



Figur 6: Perfekt 128-byte-minneaksess

Figur 6 viser en perfekt 128-byte-minneaksess. Merk at figuren er nesten den samme som Figur 5, men at hver tråd leser 8 byte(dobbeltpresisjon). Alle 128 byte overføres i en operasjon og man oppnår full båndbredde med tanke på datamengde, men bare halvparten av ytelsen til "Figur 2: Perfekt 64-byte-minneaksess", med tanke på antall overførte tall per tråd per tidsenhet.

Det er få problemtyper der hver tråd alltid leser fra samme sted i samme segment, så for å hjelpe til å oppnå coalescing, har hver kernelblokk tilgang på et lokalt hurtig cacheminne. Dette cacheminnet er tilgjengelig for alle trådene i en blokk, og kan brukes til å bytte data mellom tråder. I enkelte tilfeller der minneaksessen ikke er optimal, kan man la hver tråd lese fra sin naturlige adresse, og så bytte data gjennom cache etterpå.

GPU-CPU overføring

Databussen mellom CPU og GPU er typisk en størrelsesorden tregere enn minnebussen internt i GPU. Latenstiden for overføringer er også høy. Overføringer mellom GPU og CPU burde derfor minimeres.

Om man sparer seg en overføring fra GPU til CPU og tilbake, er det ofte raskest å gjøre beregningen på GPUen direkte, selv om beregningen ikke er optimal for GPU.

Om mengden beregninger som skal gjøres per data er liten, er det sjelden noe ytelsesforbedring å hente på å overføre data fra CPU til GPU for beregning og så tilbake igjen.

Occupancy

Et begrep som lanseres i best practices guiden **[3]** er occupancy, som er definert som forholdet mellom effektivt antall tråder og maksimalt antall mulige tråder per multiprosessor. En GPU bruker som nevnt hurtig bytting mellom tråder for å skjule minnelatens. For at denne byttingen skal gå fort nok, ligger alle dataene til hver eneste tråd i registeret og cacheminnet hele tiden fra tråden skapes til den endelig termineres. Både register og cache er begrenset på en GPU og det setter en øvre grense på hvor mange tråder man kan kjøre samtidig.

GPUer med CUDA Compute Capability 1.3 kan maksimalt ha 1024 aktive tråder per multiprosessor, og disse trådene skal dele til sammen 16 KB med cache og 16384 32-bits register. Om man vil kjøre 1024 tråder samtidig, kan hver tråd bruke maksimalt 16 registerplasser, og 16 byte med cache. Det er ikke veldig mye, og heldigvis er man ikke nødt til å ha 100% occupancy for å få god ytelse. Om kernelen gjør lite arbeid per minneaksess kan lav occupancy være nok til å mette båndbredden, og en tommefingerregel er at 50% occupancy er nok til å dekke over minnelatens.

(3) Optimalisere instruksjoner og kontrollflyt

Enkelte instruksjoner koster mer enn andre. Dette er imidlertid ikke særlig annerledes på GPU enn det er i vanlig programmering så det vil ikke bli brukt så mye tid på det. Det holder å nevne at divisjon er dyrere enn multiplikasjon og addisjon, og heltallsmodulus er nevnt som særskilt kostbar på GPU, så bruken av denne instruksjonen er minimert. Det som derimot ér annerledes på GPU er kontrollflyt. I CUDA gjøres minneaksesser i half-warps på 16 tråder, og kontrollflyt gjøres i warps på 32 tråder. Om koden har en divergerende kontrollflyt (if/else/while) der betingelsen er avhengig av tråd-idnummer, slik at tråder i samme warp kan gå ulike veier, vil alle divergensene kjøres sekvensielt. Det vil si at de trådene som *ikke* gikk en bestemt vei, settes på vent helt til trådene som gikk den veien er ferdige. Igjen ser man tegn på at GPUens primære oppgave er grafikk. Nærliggende piksler beregnes ofte av

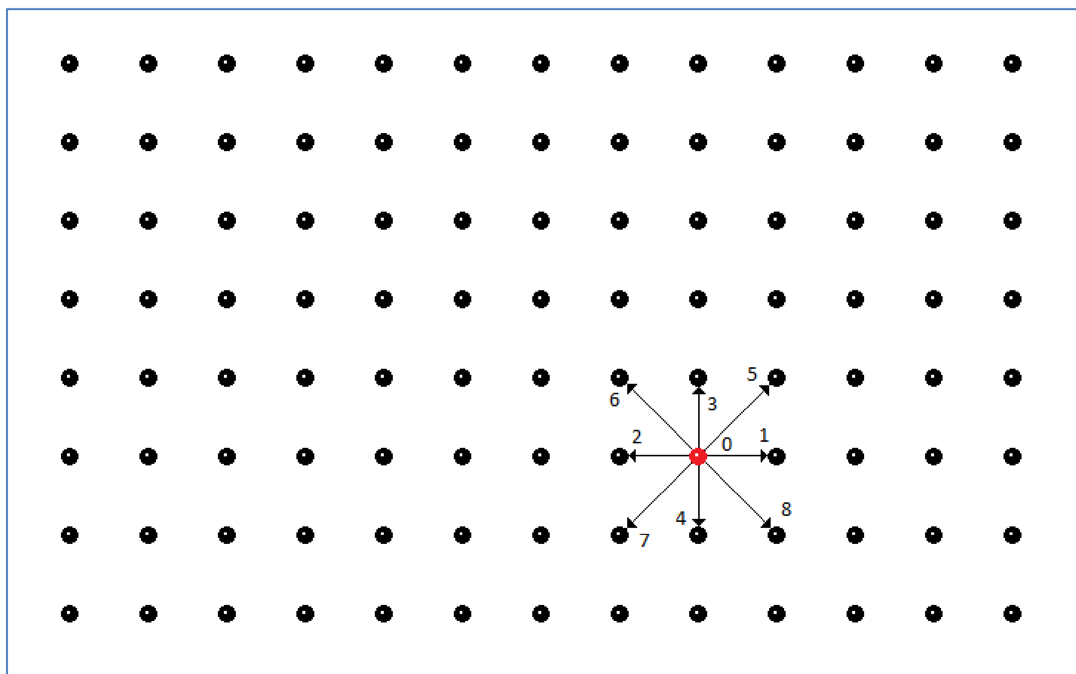
den samme lys/fargemodellen og divergerende beregningsflyt er uvanlig. Man sparer derfor transistorer på chipen ved å la flere kjerner dele instruksjonskontrollmekanismer.

Lattice-Boltzmann

Lattice-Boltzmann er en metode for å simulere viskøs fluidmekanikk. Den er en diskretisert avledning av Boltzmann-ligningen, (av Ludwig Boltzmann) som beskriver partikkeldynamikk. Abstraksjonen i Boltzmann-ligningen er at partikler i rommet kan beskrives av følgende funksjon:

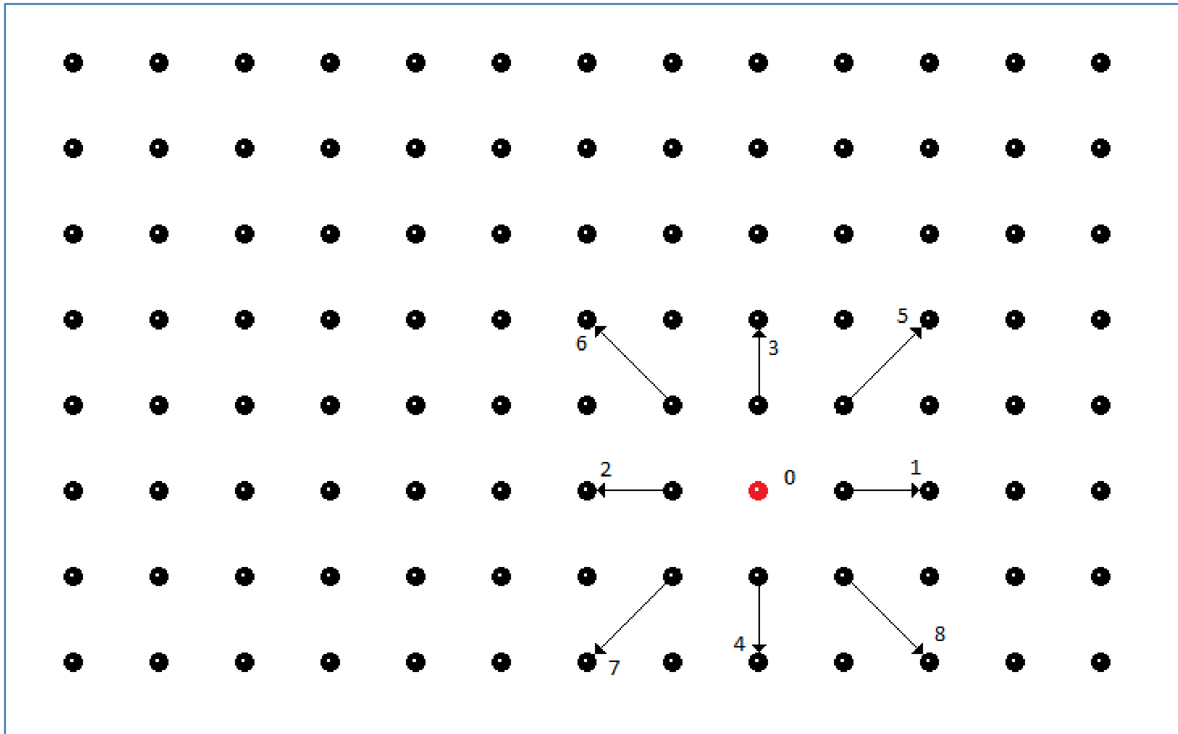
$$f(\boldsymbol{x},\boldsymbol{p},t)d\boldsymbol{x}\,d\boldsymbol{p}$$

f(**x**,**p**,t)d**x** d**p** er antallet partikler innenfor en kube dx^3 om punktet **x**, som har et momentum innenfor en kube dp^3 om momentumet **p**, ved tiden t. Ved å gjøre antagelser om krefter som virker på partikkelkontinuumet og interne kollisjoner mellom partikler kan man si noe om hvordan funksjonen vil utvikle seg med tiden. Ligningen er i utgangspunktet kontinuerlig definert for alle **x**, og alle **p**. I Lattice-Boltzmann er rommet diskretisert til et endelig antall punkter, og momentrommet er diskretisert til de verdiene av **p** som gjør at partiklene akkurat kan bevege seg fra ett punkt til ett nabopunkt på ett tidssteg. Figur 7 viser en todimensjonal uniform diskretisering av rommet, og det røde punktet viser diskretiseringen av momentrommet slik at partiklene når akkurat fram til nabopunktene på et tidssteg. Eksempelet i Figur 7 er en D2Q9 diskretisering. D2, for 2 dimensjoner i rommet, Q9 for 9 hastighetsvektorer per punkt, en til hver av de 8 naboer pluss en null-hastighet.



Figur 7: Diskretisering av Boltzmannligningen

En tidsiterasjon av Lattice-Boltzmann består av en "propagate" operasjon og en "relax" operasjon. Propagate består i at alle momentverdiene flytter til sine respektive nabonoder. Figur 8 viser hvordan Figur 7 ser ut etter at propagate operasjonen er ferdig. Den samme forflyttningen skjer for alle nodene, så når operasjonen er over har alle nodene fremdeles 9 vektorverdier, en fra hver nabo.



Figur 8: Utvikling etter "propagate"

I relax-operasjonen brukes hastighetene lokalt i en node etter propagate-steget til å beregne nye hastigheter basert på antagelser om interne partikkelkollisjoner. Dette er steget som gjør at fluidet oppfører seg viskøst. Det finnes flere ulike kollisjonsmodeller, men den enkleste og mest brukte kalles BGK etter Bhatnagar, Gross og Krook som utviklet metoden i 1954. Metoden er også kjent som "single-relaxation-time approximation". Man lar hver momentvektor bevege seg mot en kjent ekvilibriumshastighet med en konstant faktor. Denne faktoren bestemmer da i praksis den kinematiske viskositeten til fluidet. BKG modellen for kollisjon ser slik ut:

$$f_a^{n+1} = f_a^n + \frac{1}{\tau}(g_a^n - f_a^n)$$

f_a^n : Verdien tilhørende vektor nr a, ved tidssteg n

g_a^n : Verdien av ekvilibriumsfunksjonen til vektor a ved tidssteg n

τ : Avslapningsfaktoren, "dimensionless relaxation time". Bestemmer kinematisk viskositet ved relasjonen:

$$\nu = \frac{(2\tau - 1)}{6} \frac{\delta_x^2}{\delta_t}$$

ν : Kinematisk viskositet

δ_x : Avstand mellom lattice-noder

δ_t : Tidssteg

For tilfellet D2Q9 med uniformt grid er ekvilibriumsfunksjonen g_a^n bestemt av ligningene:

$$g_a^n = w_a \rho \left(1 + \frac{(\boldsymbol{u} \cdot \boldsymbol{e_a})}{c_s^2 c^2} + \frac{(\boldsymbol{u} \cdot \boldsymbol{e_a})^2}{2c_s^4 c^4} - \frac{\boldsymbol{u}^2}{2c_s^2 c^2}\right)$$

$$c = \frac{\delta_x}{\delta_t}, \qquad c_s = \frac{1}{\sqrt{3}}$$

$$\boldsymbol{e_0} = (0,0), \quad \boldsymbol{e_1} = (c,0), \quad \boldsymbol{e_2} = (-c,0), \quad \boldsymbol{e_3} = (0,c), \quad \boldsymbol{e_4} = (0,-c)$$

$$\boldsymbol{e_5} = (c,c), \quad \boldsymbol{e_6} = (-c,c), \quad \boldsymbol{e_7} = (-c,-c), \quad \boldsymbol{e_8} = (c,-c)$$

$$\rho = \sum_{i=0}^{8} f_i^n, \qquad \boldsymbol{u} = \frac{1}{\rho} \sum_{i=0}^{8} f_i^n \, \boldsymbol{e_i}$$

$$w_a = \begin{Bmatrix} 4/9 & a = 0 \\ 1/9 & a = 1,2,3,4 \\ 1/18 & a = 5,6,7,8 \end{Bmatrix}$$

ρ : Tettheten til fluidet i punktet

$\boldsymbol{u}$: Makroskopisk fluidhastighet

c_s : Dimensjonsløs lydhastighet i fluidet.

Vektfaktorene w_a, vektorene $\boldsymbol{e_a}$, samt lydhastigheten c_s er avhengig av diskretiseringen. I denne oppgaven er det valgt en D3Q19 lattice, og konstantene kan finnes i Appendiks A. En detaljert utledning fra Boltzmann ligningen til diskretisering for noen ulike latticer kan man finne i et paper fra 1997 av Xiauyi He og Li-Shi Luo **[4]**.

Grensebetingelser

Grensebetingelser er implementert etter mønster fra en enkel fortan-implementasjon **[5]** av en D2Q9 Lattice-Boltzmann-løser av Joerg Bernsdorf.

Noslip-grensebetingelse er implementert som et enkelt bounce-back skjema. Det vil si at i noder som er markert som veggnoder snus alle hastigheter 180 grader istedenfor vanlig relax-rutine.

Drivende trykkbetingelse er implementert ved at i noder ved innkommende kant vil litt masse redistribueres for hvert tidssteg. Masse redistribueres fra vektorer som peker i negativ retning, til vektorer som peker i positiv retning med en konstant faktor. Størrelsen på denne faktoren bestemmer trykket.

Visualisering

Jeg har valgt å skrive mitt eget visualiseringsprogram til denne oppgaven. Det finnes selvsagt mange gode visualiseringsprogrammer man kan bruke, men visualisering er et felt som interesserer meg, så jeg hadde lyst til å gjøre det selv. Jeg hadde også lyst til å implementere en 3D-stereoskopisk visualisering siden 3D er sånn i vinden for tiden.

Visualisering er et av de nyttigste verktøyene man har for å undersøke at koden fungerer som den skal. Selv om mere kvantitative metoder også er nødvendig for en komplett verifisering, er det uvurderlig å kunne se at beregningene oppfører seg kvalitativt slik de skal.

Visualisering av tredimensjonal informasjon er ofte vanskelig å gjøre på en god måte. Særlig 3D vektorfelt er problematiske. Man kan tegne det diskré vektorfeltet direkte, men det gir ingen god følelse på hastighetsforskjeller og blir fort rotete, særlig for høyoppløste vektorfelt.

Andre muligheter er å redusere vektorfeltet til en eller annen skalarverdi (trykk, skalar curl, hastighet-absoluttverdi, osv.) og så plotte enten tredimensjonale isoflater, eller tegne isokurver på et snittplan gjennom volumet. Isoflater lar deg kun studere én verdi av gangen, mens snittplan kun lar deg se en del av volumet av gangen.

I konteksten fluidmekanikk gir det mening å snakke om strømlinjer og partikkelbaner. Om man ikke tegner for mange eller for få linjer kan de også gi en god oversikt over strømningsbildet. Det er ofte den visualiseringsteknikken som er intuitivt enklest å forstå.

Det finnes flere ulike teknikker for volumetrisk visualisering som kan gi oversikt over hele volumet av gangen, men de er som regel enten veldig beregningskrevende, vanskelige å implementere, eller begge deler.

Alle disse metodene har ulike styrker og svakheter, men til sammen kan de gi en god oversikt over strømningsbildet.

Denne oppgaven vil ta for seg stereoskopisk visning av isoflater. Snittflater har jeg gjort i tidligere prosjekter, og volumetrisk visualisering har jeg ikke nok erfaring og kunnskap om til at jeg vil prøve det ennå.

Stereoskopisk 3D

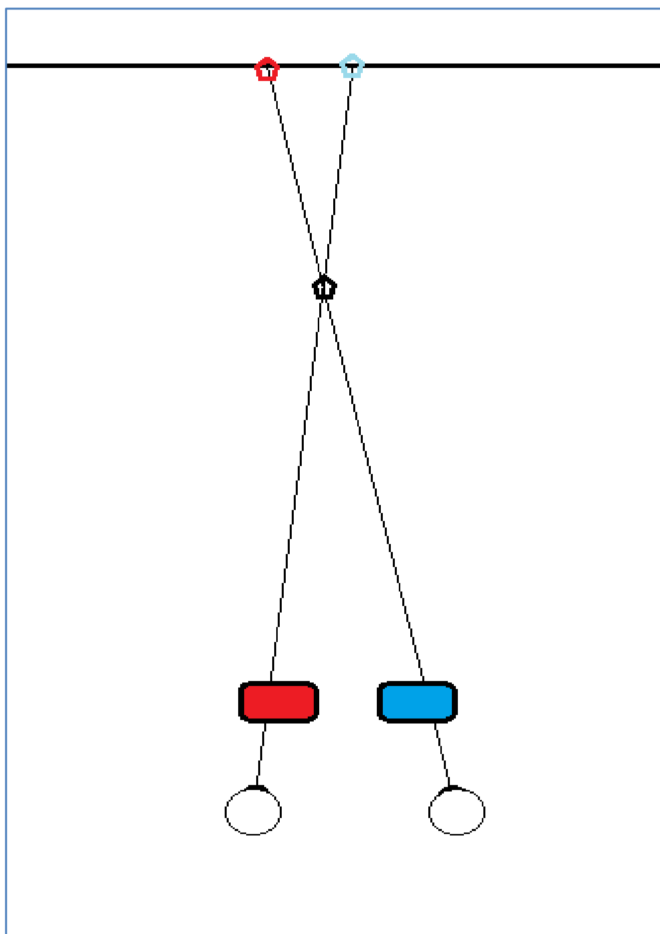
En mulighet for å gjøre 3D-strukturer på en dataskjerm enklere å forstå, er å visualisere bildet stereoskopisk. Siden vi har to øyne som sitter ved siden av hverandre, mottar de litt ulike bilder av verden foran seg. Hjernen kan så bruke disse små ulikhetene til å rekonstruere en 3 dimensjonal forståelse av rommet.

Om man finner en måte å vise litt ulike bilder til hvert øye, kan man skape en illusjon av tredimensjonalitet. Det finnes ulike teknikker for å sende ulikt bilde til ulike øyne. Man kan projisere hvert bilde med ulik polarisering av lyset, enten lineært eller angulært, og så benytte polariserte brilleglass til å filtrere lyset. Løsningen med angulær polarisering er i bruk i moderne 3D kinoanlegg i dag. Andre løsninger som er i bruk er spektralfilter-brilleglass. De baserer seg på at det menneskelige

øyet kun ser i 3 ulike farger, rød, grønn og blå. Det interne styrkeforholdet mellom de tre fargene brukes til å rekonstruere hvilken farge man "egentlig" ser på. Om man har to brilleglass som hver kun slipper gjennom et smalt spekter av fargen rød, men de to "rødfargene" har litt ulik frekvens, og man gjør det samme for grønn og blå, kan man projisere et komplett fargebilde til hvert øye. Det finnes løsninger med aktive brilleglass, der hvert glass er en gjennomsiktig LCD skjerm som alternerer på å sperre/slippe gjennom lyset. Så kan man synkronisere brillene med en skjerm som alternerer mellom å sende bildet til høyre øye og venstre øye.

Løsningen som er valgt er den gamle og enkle løsningen med fargefilter. Ett rødt brilleglass og ett blått(cyan). Ved å tegne ett bilde i rødtoner og ett i blåtoner og så legge bildene oppå hverandre, kan man sende et bilde til hvert øye. Bakdelen med denne løsningen er at det er vanskelig å gjengi fargeinformasjon utover gråtoner. Det er mulig å skape en viss grad av farge ved å variere blågrønnbalansen i bildet som skal til cyanfilteret, men illusjonen er ikke perfekt. En annen bakdel er at det blir en ubehagelig rød/blå flimring når man har på brillene. Det er dog en vanesak, og man legger mindre merke til det etter hvert. Fordelen er at det er den billigste, enklest tilgjengelige løsningen. Det krever ingen spesialskjerm eller prosjektør med polariserte filter, kun briller med fargeglass og en vanlig dataskjerm. Om bedre utstyr skulle bli tilgjengelig er det en smal sak å omskrive koden til å dra nytte av det.

Prinsippet bak stereografisk visualisering



Figur 9: Stereoskopisk prinsippdiagram

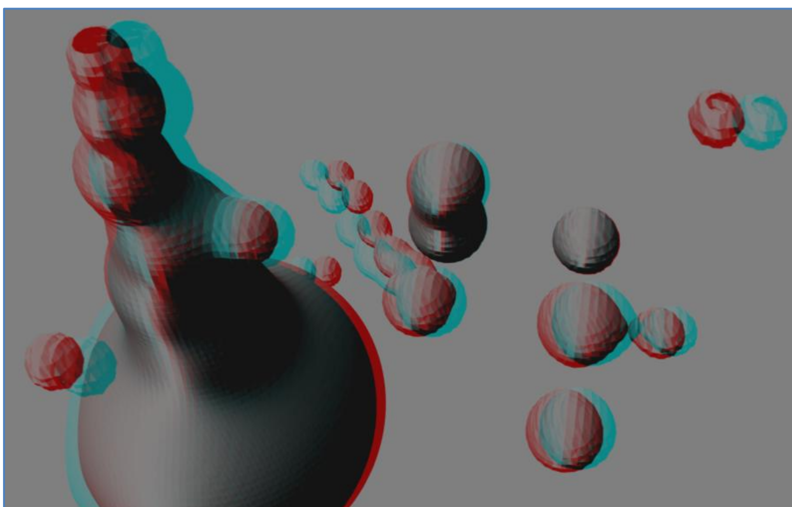
Figur 9 viser hvordan stereoskopisk visualisering fungerer. Hele scenen er sett fra fugleperspektiv. De to kulene nederst i bildet er to øyne, som ser gjennom hvert sitt brilleglass med hver sin farge. Streken øverst på bildet er en skjerm. Den røde og blåe prikken er en henholdsvis rød og blå prikk på skjermen. Bakgrunnen ellers er helt hvit. Siden fargen hvit og fargen rød inneholder like mye rødfarge, ser ikke det venstre øyet den røde prikken. Det samme gjelder for det høyre øyet og den blå prikken. Derimot inneholder fargen svart og fargen rød like lite blåfarge, så det høyre øyet ser den røde prikken som en svart prikk. Høyre og venstre øyet ser nå hver sin svarte prikk på litt ulikt sted og konkluderer med at den svarte prikken befinner seg et sted i rommet foran skjermen. Man kan styre hvor langt foran skjermen, ved å justere avstanden mellom prikkene. Det romlige punktet vil befinne seg i skjæringspunktet mellom synslinjene.

Problemer med stereografisk visualisering

For å tegne stereoskopisk trenger man da i teorien bare å vite størrelsen på skjermen, avstanden mellom øynene og hvor langt unna skjermen man sitter. Så tegner man bildet to ganger fra litt ulike vinkler. I praksis kan det lønne seg å underdrive den stereoskopiske forvrengningen litt ved å sette en mindre øyeavstand enn den reelle. Grunnen til det ser ut til å være todelt.

Den første grunnen er at bildeseparasjonen sjelden er helt perfekt. Litt av lyset fra det røde bildet vil lekke gjennom til det blåe øyet og lage et vagt spøkelsesbilde som ligger over. Denne krysskommunikasjonen forstyrrer 3D-opplevelsen. Dette vil gjøre at hjernen har vanskelig for å bestemme seg for om den ser to 2D-bilder eller ett 3D-bilde, og illusjonen bryter sammen. Dette gjelder spesielt om man har store ensfargede flater eller linjer med skarpe kontraster. Om bildet har variert tekstur og krokete linjer vil denne krysskommunikasjonen erfaringsmessig ha mindre betydning. Dette problemet kan løses med bedre teknologi, men det hjelper også å underdrive 3D-effekten litt. Det skal mindre til for at illusjonen bryter sammen om høyre-/venstrebildene er langt fra hverandre

Den andre grunnen har å gjøre med linsefokus. Øyet har en viss blenderåpning for å slippe mere lys inn på retina. Det gjør at vi kan se tydeligere i svakere lys, men det gjør også at vi må endre fokuspunkt etter hvilken avstand vi ser. Jo nærmere det vi ser på er, jo mer må øyets linse krumme for å se klart. Denne refleksen er kodet inn i nervesystemet og vi trenger ikke tenke over det. Når man ser stereoskopisk på en skjerm derimot, er det korrekte fokuspunktet alltid like langt unna. Når øynene skal se på noe de tror er nærmere i det virtuelle rommet, vil linsen automatisk krumme, fokusere på et punkt foran skjermen, og bildet vil bli uklart. Hjernen vil også forvente at gjenstander som befinner seg i perifersynet, og som er nærmere/fjernere enn fokuspunktet skal være ute av fokus. På en skjerm vil ikke dette være tilfellet. Dette problemet vil være vanskelig å løse med bedre teknologi. Det krever i så fall briller med automatisk hurtig justerbare linser, sanntidsinformasjon om øyefokus, pupillsiktepunkt og hodeposisjon, samt visualisering med dybdeskarphet. En annen mulig teknisk løsning kan være kontaktlinser med smal blenderåpning, samt veldig lyssterke skjermer. Det kan derimot løses delvis med enkel tilvenning. Om man bruker litt tid med 3D-briller vil hjernen etter hvert lære at den ikke trenger å justere fokus likevel.



Figur 10: 3D stereo "metaball" bilde

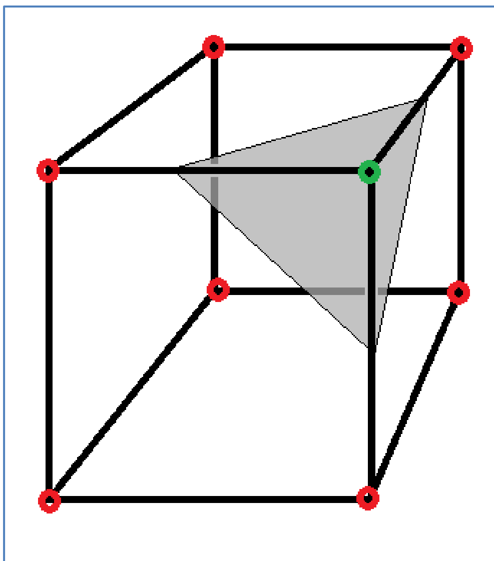
Figur 10 viser et stereoskopisk bilde fra en utviklingsversjon av visualiseringsprogrammet basert på marching tetrahedron. Med et par rød-/cyanbriller kan man kanskje observere noen av de effektene som er beskrevet. Ghost-image effekten vil typisk variere avhengig av fargekalibrering av skjerm eller printer.

Det som er gjort i praksis er å definere 3D-rommet i forhold til skjermplanet slik at ting stikker ca. like mye inn i, som ut av skjermen, og da ikke så veldig langt. Slik vil det virtuelle fokuspunktet og det reelle fokuspunktet være mest mulig likt, og avstand mellom rød/blå bildet vil minimeres. Grå bakgrunn istedenfor helt hvit eller svart viste seg også å hjelpe litt på ghost-imagingen, ved å senke kontrasten.

Generering av isoflater

Det finnes en kjent metode for å generere en polygonbasert isoflate av et tredimensjonalt diskret skalarfelt. Metoden kalles for "marching cubes algorithm" og baserer seg på å iterere over hele volumet, kube for kube og finne ut om kuben har hjørner som ligger både utenfor og innenfor isoflaten.

Marching cubes algorithm



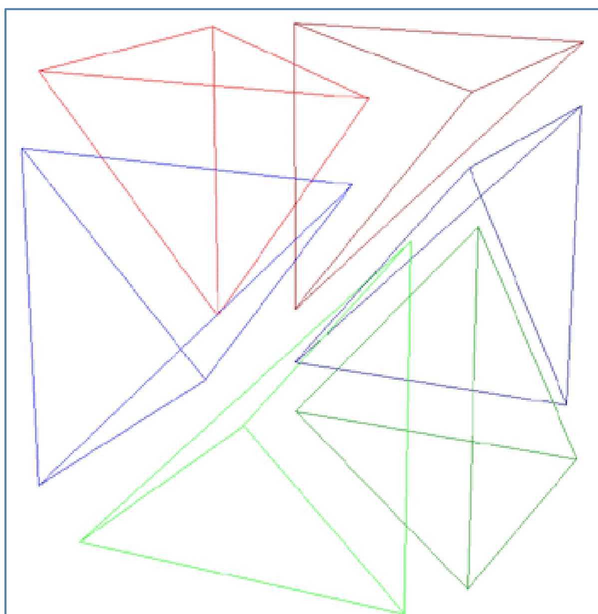
Figur 11: Marching cubes algorithm

Figur 11 viser et eksempel på en situasjon marching cubes algoritmen kan havne i. Hvert hjørne i kuben er et diskret datapunkt som har en funksjonsverdi. Det grønne hjørnet har en funksjonsverdi som er større enn isoverdien, og ligger da utenfor isoflaten. De røde hjørnene ligger innenfor isoflaten. Man antar så at funksjonverdien endrer seg lineært langs kubekantene, og finner punktene som ligger akkurat på isoflaten. Disse punktene brukes som hjørnene i et polygon.

Det finnes 256 ulike kombinasjoner av ute/inne hjørner som gir ulike regler for hvilke polygoner som skal tegnes. Noen av disse har derimot flere ulike lovlige konstellasjoner av polygoner som gir kvalitativt annerledes resultat. Det viser seg at på grunn av disse ubestemte tilfellene er man ikke garantert en "vanntett" overflate om man blindt kjører algoritmen kube for kube **[10]**. Av og til er man avhengig av å vite hvordan situasjonen er i nabokubene for å vite hvilken case man skal velge. For å få en enklest mulig implementasjon som garanterer vanntett overflate, er det valgt en variant av marching cubes som heter marching tetrahedron.

Marching tetrahedron algorithm

I marching tetrahedron blir hver kube oppdelt i 6 tetrahedroner, som vist i Figur 12. Hvert tetrahedron behandles så på samme måte som kuben i marching cubes. Hjørner utenfor og innenfor isoflaten identifiseres, og man interpolerer funksjonsverdien mellom hjørnene for å finne isoflaten.



Figur 12: Marching tetrahedron algorithm

For et tetrahedron finnes det bare tre ulike tilfeller om man tar hensyn til symmetrier og speilinger. Alle hjørnene er på samme side av isoflaten, én er på ulik side av resten, og to på hver side. Alle disse har kun en lovlig polygonkonfigurasjon hver, og siden alle tetrahedronene deler side/hjørne med en annen, og interpolasjonen er lineær mellom hjørnene, er man garantert en vanntett overflate.

Ytelse på marching tetrahedron algorithm

Det viste seg at for de latticestørrelsene av typisk interesse i denne oppgaven, var ytelsen alt for dårlig. Selv på under middels store latticer kunne det ta over ett sekund å lage en ny isoflate. Ideelt sett burde man kunne endre isoverdien og få realtime feedback på isoflaten. Min opprinnelige plan var å bruke GPUen til å beregne isoflaten og tegne direkte, uten noen kommunikasjon med CPU. På grunn av problemer med kompilering og linking i Visual Studio 2008 gikk det ikke å kjøre openGL og CUDA i samme program, og når det først fungerte var det sent å starte fra grunnen av med å lære CUDA-openGL interface.

Ulike optimaliseringsmetoder ble forsøkt. Parallellisering av koden fungerte ikke siden openGL ikke støtter kall fra flere ulike tråder uten videre. En annen teknikk som ble forsøkt, var å bruke GPU til å gjøre en hurtig prekalkulering av kuber som lå enten helt utenfor eller helt innenfor isoflaten, slik at CPUen kunne hoppe hurtig over dem. Antall kuber som inneholder en del av isoflaten vil typisk være en størrelsesorden mindre enn antall kuber totalt, så dette burde kunne gi en brukbar ytelsesforbedring. Det viste seg derimot at den dominerende tidsbruken var i selve kallene til openGL funksjoner, og den størrelsen er ureduserbar.

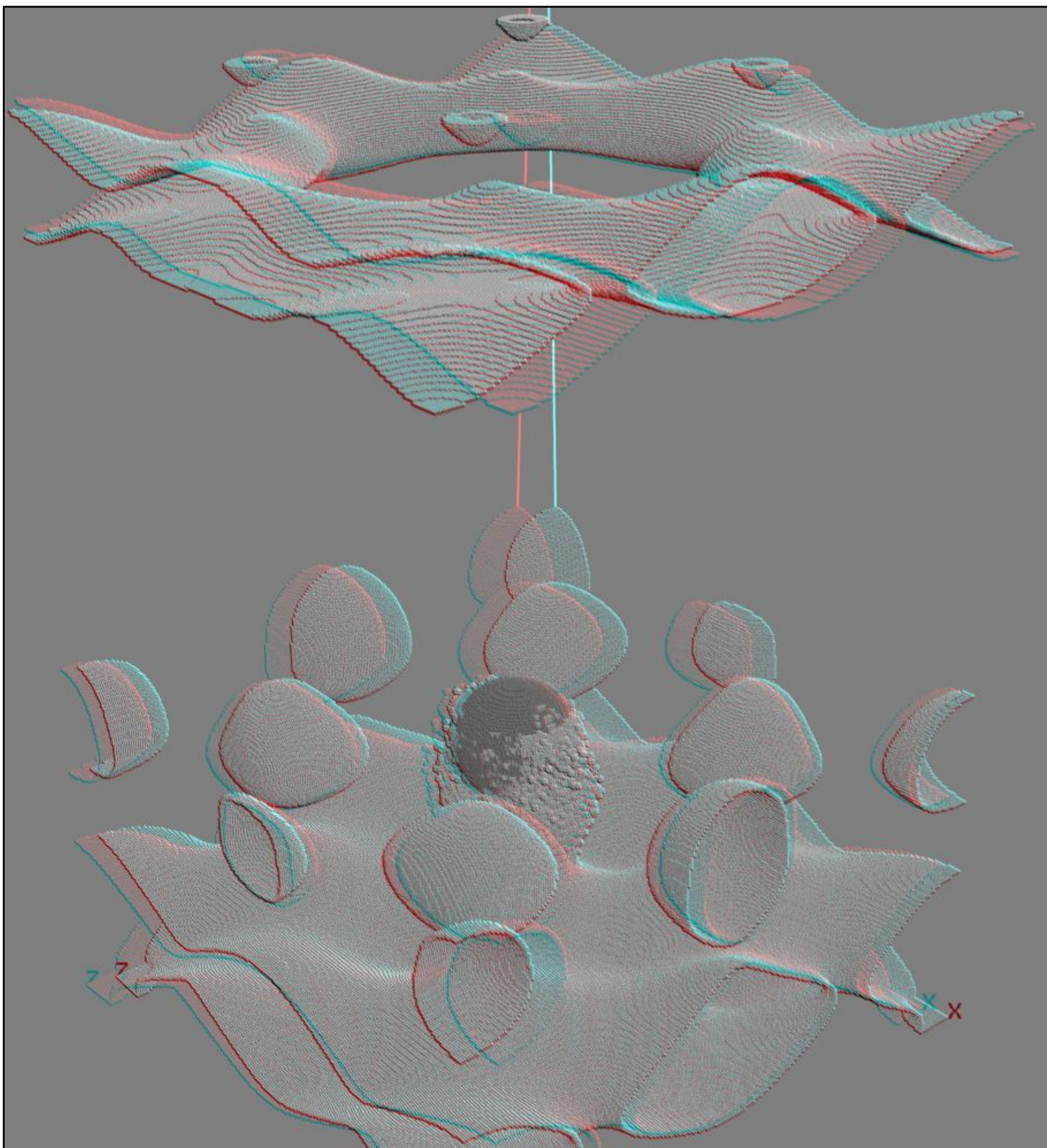
Den endelige løsningen ble å forkaste marching tetrahedron og gå for en helt annen teknikk. Ved å skrive sin egen spesialiserte vertex- og fragment-shader kan man bruke et enkelt kall til en openGL funksjon som egentlig er beregnet til å tegne ensfargede firkanter av uniform størrelse, til å tegne skyggelagte kuler i et 3D-rom. Om man så tegner en slik kule i hver eneste kube i rommet som inneholder en liten del av isoflaten, vil den kumulative effekten bli noe som ligner ganske nært på selve isoflaten bare oppløsningen er høy nok.

Dette vil redusere antall nødvendige kall til en openGLvertex-funksjon til maksimalt én per kube. Tidligere var det i beste fall 6, og det kunne fort bli oppimot 20. Det blir også mindre jobb for grafikkortet å tegne resultatet etterpå.

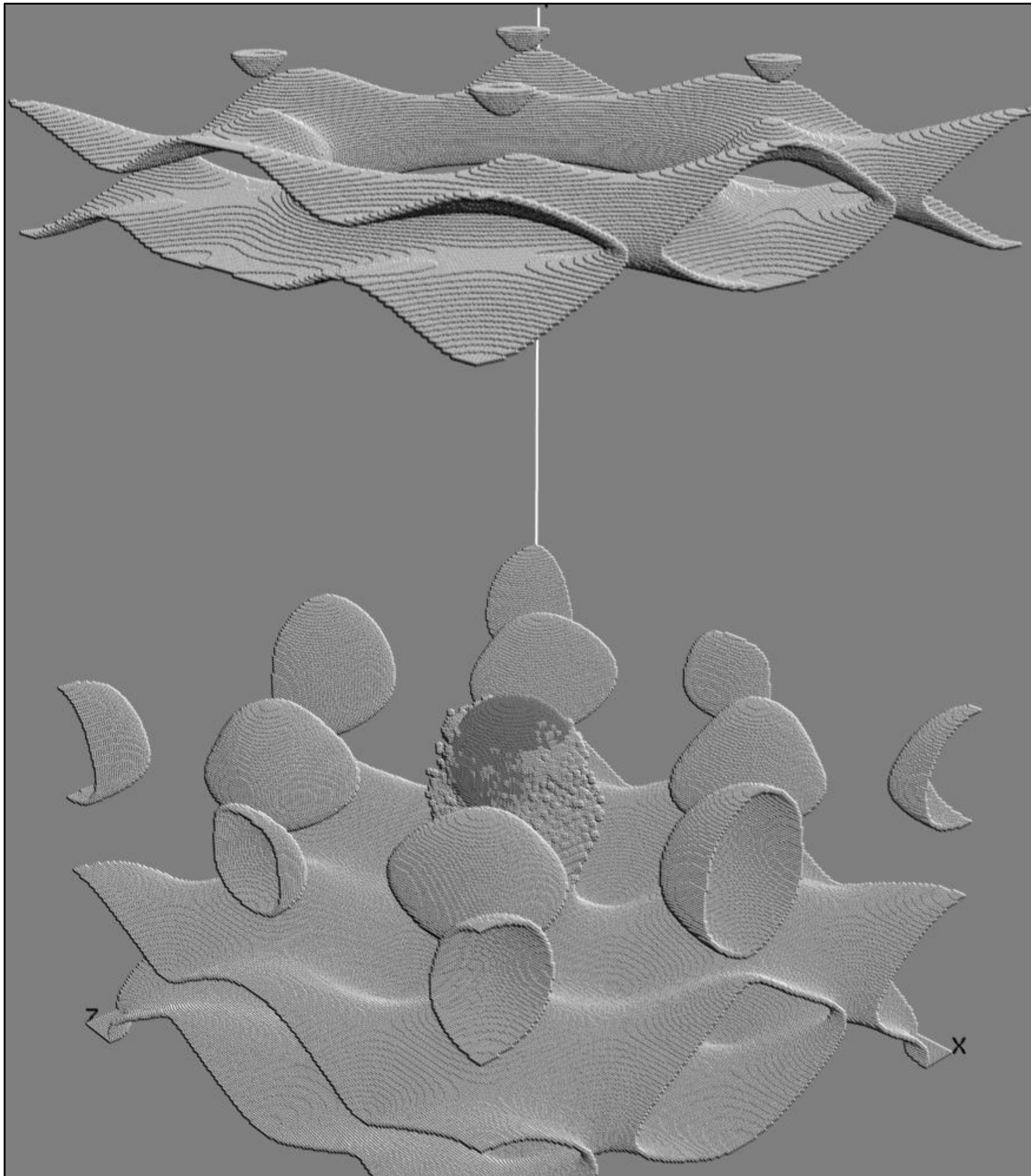
Fremdeles er dette en jobb som kunne gjøres langt raskere av grafikkortet uten noen hjelp fra CPU, siden hver kube kan klassifiseres uavhengig, men det ble som sagt ikke tid til den funksjonaliteten. Det fungerer nå tålelig bra på mellomstore grid.

En tilleggsbonus ved å bruke denne metoden er at den utmerket godt kan brukes til å visualisere linjer så vel som flater, og det var en smal sak å modifisere koden til å tegne strømlinjer også.

Figur 13 og Figur 14 viser den samme flaten visualisert henholdsvis stereoskopisk og ikke-stereoskopisk med kuleteknikken på en [256 x 400 x 256] lattice. Bildet er ikke valgt av noen annen grunn enn at det ser pent ut. Bildet er tatt før strømningen har stabilisert seg og det man ser er antageligvis en slags sjokkbølgeeffekt. Merk at ghost-image effekten typisk vil være verre på trykk enn på skjerm.



Figur 13: Stereoskopisk trykk-isoflate for kule i uniform strøm ved Reynoldstall 500, og Machtall 0.1



Figur 14: Ikke-stereoskopisk trykk-isoflate for kule i uniform strøm ved Reynoldstall 500, og Machtall 0.1

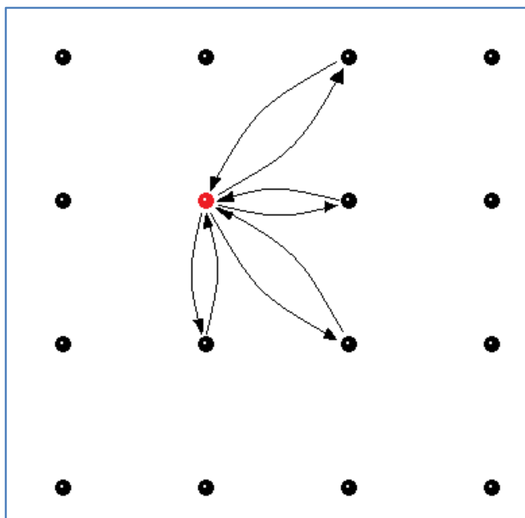
Lattice-Boltzmann på GPU

Det er som nevnt i kapittelet "Hvordan få god ytelse i CUDA" tre ting som er oppgitt som særskilt viktige for ytelsen i CUDAprogrammering, og dette kapittelet vil gå gjennom alle tre og si noe om hvordan det har påvirket algoritmedesignet.

(1) Maksimere parallell utførelse

Lattice-Boltzmann er svært godt egnet for parallellisering, og det er nettopp grunnen til at denne metoden er valgt. Det naturlige i GPUprogrammering er en såkalt "datastyrt parallellisme". Det vil si at hver tråd tar seg av sin lille del av datasettet, og det er datastørrelsen, ikke antall kjerner som styrer hvor mange tråder man bruker. Lattice-Boltzmann egner seg ypperlig for dette. Hvert eneste latticepunkt er et datasett som kan behandles uavhengig av resten. Unntaket er i propagate-fasen der data skal flyttes til nabonoder. Der har man to muligheter for å oppnå full parallellitet. Den ene bruker halvparten så mye minne som den andre, men dobbelt så lang tid.

Den raske og minnekrevende algoritmen bruker to tabeller til å lagre latticedata. For hvert tidssteg leser hver kernel data fra en tabell. Dataene leses på en slik måte at innkommende hastigheter leses fra nabonoder, slik at propagate-steget gjøres implisitt. Så går hver kernel gjennom relax-fasen på de innleste data, og skriver resultatet til den andre tabellen. Både propagate og relax gjøres da i samme omgang og man klarer seg med å skrive og lese alle data en gang per tidsiterasjon.



Figur 15: Propagate swap

Den minnebesparende og trege algoritmen bruker bare en tabell til å lagre latticedata. For å garantere konsistent resultat uavhengig av kernel-rekkefølge gjøres propagate-fasen eksplisitt unna først, slik at hver node bytter utgående/inngående data med hverandre som vist i Figur 15. Kernelen leser altså sin egen nedoverhastighet, og sin nedenforliggende nabos oppoverhastighet fra RAM. Begge verdiene ligger nå lagret i trådens lokale register. Så skrives naboens oppoverhastighet i RAM der nedoverhastigheten lå, og omvendt. Når denne prosessen er utført for alle nodene, har man utført en modifisert propagate-fase slik at hastighetene ligger i riktig node, men på omvendt side av der de skal ligge, altså at venstre ligger på plassen til høyre og omvendt, opp ligger på plassen til ned og omvendt. Dette er imidlertid bare en fordel. Så lenge man passer på å lese fra riktig sted under relaxfasen er det ikke noe problem, og om man bare lar verdiene stå urørt, har man i praksis utført bounceback-grensebetingelse for den noden. Siden kun en tabell brukes til å lagre data har denne metoden bare halve minnebruken, men prisen man betaler er dobbelt så mange lese/skrive operasjoner til RAM.

Så lenge en enkelt lattice ikke bruker over halvparten av minnet, eventuelt at man ikke kjører andre operasjoner på kortet samtidig som krever mye minne, burde man konsekvent velge den raske men minnekrevende metoden. Siden litt av målet med oppgaven har vært å beregne størst mulig grid på

GPU, og RAM ofte vil være en begrensende faktor for problemstørrelse, er derimot den minnebesparende algoritmen valgt. Se forøvrig Appendiks B - Minneoptimering.

(2) Optimalisere minnetilgang

Minne-coalescing

Antallet minneaksesser er rimelig bestemt i Lattice-Boltzmann, så på den fronten er det ikke så mye å hente av optimalisering, men det man kan gjøre er å sørge for mest mulig samlet minnetilgang.

De fleste lese/skriveoperasjonene i Lattice-Boltzmann gjøres internt i hver node, og hver node har flere tall assosiert til seg. Så det første som er gjort for å samle minnebruken er å legge alle data tilhørende hver vektorretning samlet. Altså først ligger alle verdiene i retning #0 for alle nodene, så alle verdiene for retning #1, så for retning #2 osv.

Om minneaksessen nå er av typen en node per tråd, så er dette nok til å få perfekt minnebåndbredde. Hver half-warp leser først verdiene for retning #0, og siden de ligger etter hverandre, får man full coalescing. Så kan man gå over til retning #1 og det samme gjelder der.

Problemet ligger i propagate fasen når naboinformasjon skal leses. For å optimalisere her er latticestørrelsen i x-retning begrenset til multiplum av 32, samt mindre eller lik 512. Merk at 512 ikke er en veldig streng begrensing på x-dimensjonen. Om man skulle lagre en full lattice på 512x512x512 noder ville man trenge 9 GB RAM, og det er ennå en stund fram i tid før man har det på enkle skjermkort.

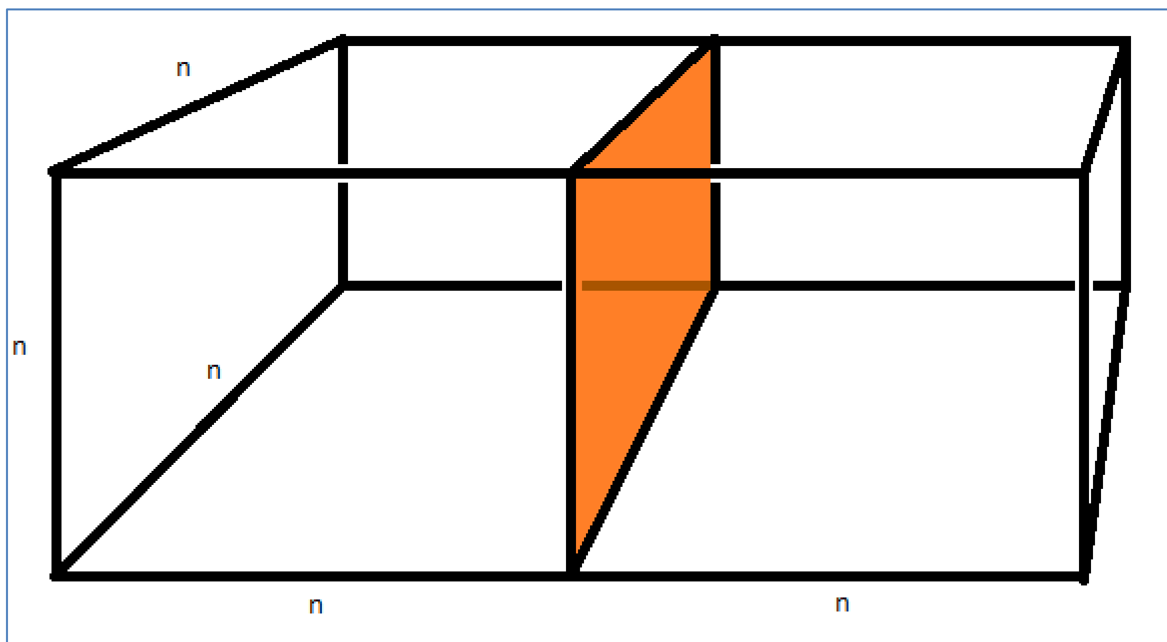
Siden lengden i x-retning ikke overskrider 512 har man muligheten til å la en hel x-rad betjenes av en enkelt kernel-blokk. Siden latticestørrelsen i x-retning er multiplum av 32, vil minneaksesser til naboer som ligger på samme høyde i x-retning passe inn på samme relative sted i et minnesegment. En tråd vil altså ha perfekt minnetilgang for alle verdier i "sin" node, samt alle naboer som ligger på samme høyde i x-retning.

For naboer som ligger foran, og bak, i x-retning vil minneaksessen ha en offset på 1 eller -1, og det fører som nevnt til en 42% reduksjon i effektiv båndbredde. For å bøte på dette er det tatt i bruk et delt cacheminne som befinner seg på hver GPUmultiprosessor. Hver blokk kjøres av en enhet på GPU som kalles en multiprosessor, og hver multiprosessor har flere kjerner, samt et delt cacheminne. Alle trådene i en blokk kan snakke med hverandre ved hjelp av dette cacheminnet. Det finnes også en synkroniseringsfunksjon som trådene kan bruke for å sikre konsistens i kommunikasjonen. Dette er den eneste formen for mellomtråd-kommunikasjon man kan få til i CUDA. Dette cacheminnet er mye raskere enn global RAM og det anbefales i "best practices guiden" **[3]** til CUDA å bruke det til å bedre minneaksessmønsteret om mulig. Det man kan gjøre er at hver blokk allokerer en tabell i cache med samme lengde som blokken. Så leser hver tråd inn data fra sin naturlige plass i RAM, men skriver det inn i cache med offset. Så synkroniserer man for å sikre at alle trådene er ferdige før hver tråd så leser data fra sin naturlige plass i cache. Istedenfor at hver blokk leser fra RAM med offset, leser hver tråd uten offset og bytter data gjennom cache minnet. Minnetilgangen er nå perfekt for alle operasjoner.

Cacheminnet er selvsagt begrenset i størrelse og på GPUer med compute capability 1.3 er det 16KB per multiprosessor. Dette setter en øvre begrensing på gridstørrelse i x-retning. For en D3Q19 lattice trengs det 10 tabeller per kernel-blokk, hver med samme lengde som blokken. For enkelpresisjondata betyr det at størrelsen på nødvendig cacheminne er 40 byte ganger blokkstørrelsen. Det gir en maksimal gridstørrelse på 384 i x-retning. For 384x384x384 grid er total datamengde over 4GB for en full lattice. Et vanlig forbrukergrafikkort har rundt 1 GB, og større GPGPUkort har opp mot 4 GB, så cacheminnet vil ikke være den begrensende faktor på gridstørrelse enda.

GPU-CPU overføring

Lattice-Boltzmann kan i utgangspunktet kjøres helt alene av GPU og man trenger kun å overføre data når man vil lagre et resultat. For å kjøre på flere GPUer samtidig derimot, kreves det kommunikasjon ved hvert tidssteg. Det er heldigvis bare overflatedata som skal overføres og om man partisjonerer beregningsdomenet sitt riktig, vil mengden overførte data være en størrelsesorden mindre enn total mengde data.



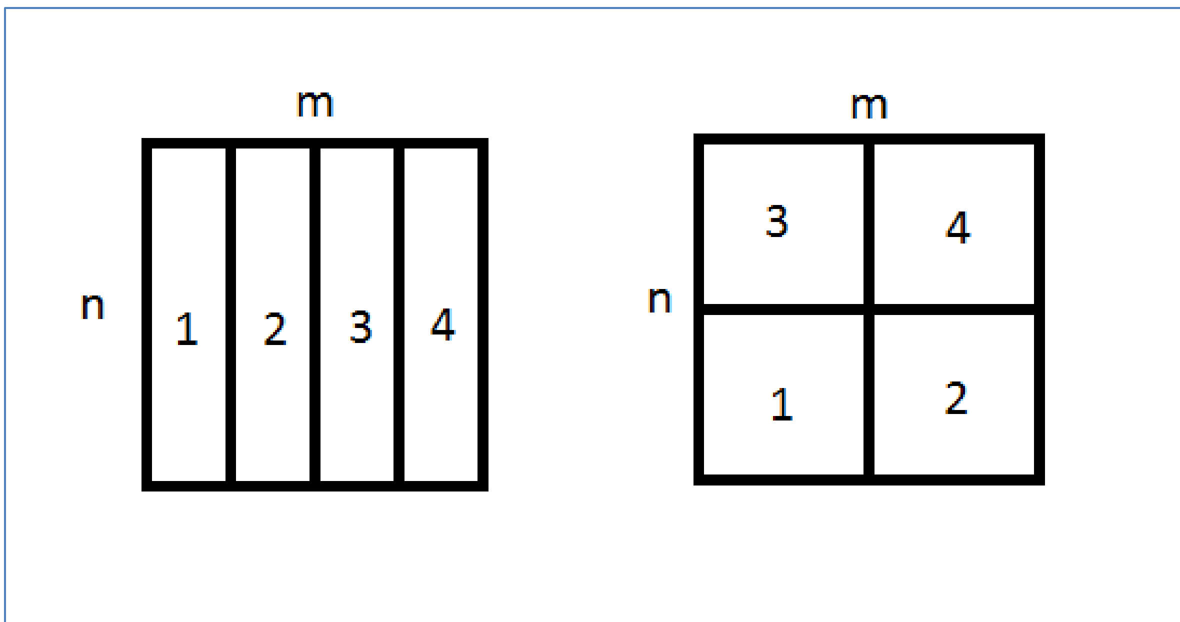
Figur 16: 1x2 Domenepartisjonering

Figur 16 viser en optimal todelt partisjonering av et domene med dimensjoner nxnx2n. Hver partisjon har da lik størrelse nxnxn, og overflatedataene som skal overføres mellom partisjoner, markert med oransje farge har størrelse nxn, eller 1/n av total mengde. Så lenge alle tre dimensjoner er omtrent like store, så vil overflatedata utgjøre en marginal del av totalen for store nok verdier av n. Når man går til det steget at man vil bruke flere GPUer til å løse Lattice-Boltzmann, er det gitt at n er relativt stor og typiske verdier er fra n = 256 og oppover.

Jeg har valgt å utvikle opp mot 4 GPUer siden det er så mange det største Teslakortet fra NVIDIA har, og jeg har vært så heldig å få tilgang til et fra HPC-laben på NTNU. For å redusere antallet overføringer er overflatedataene pakket sammen i en sammenhengende tabell før overføring. For at

denne pakkingen skal kjøre med best mulig minnebåndbredde, er partisjonsoverflatene begrenset til å gjelde xy-, og xz-planet. Pakking av data fra yz-planet ville uunngåelig ført til en worst case scenario minneaksess, der hver eneste tråd leser fra ulike minnesegment.

I situasjonen med 4 GPUer har man valget mellom å legge partisjonene på rekke etter hverandre eller i et 2x2 mønster. Implementeringen av partisjoneringsalgoritmen er basert på en topologitabell som inneholder informasjon om nabopartisjoner. Denne tabellen brukes så til å rute overflatedata til riktig partisjon, og kan hanskes med både rekkepartisjonering og 2x2. En kjapp retrospektiv analyse viser derimot at en rekkepartisjonering ville fungere bedre, og det ville vært mye enklere å implementere.



Figur 17: 2x2 vs. 1x4 partisjonering

Figur 17 viser en 1x4 og en 2x2 partisjonering av et domene med størrelse nxm. For 1x4 partisjoneringen må hver partisjon sende 'n' data til forrige, og neste partisjon. Det gir 2n data per partisjon og 8n til sammen. For 2x2 må hver partisjon sende n/2 data mot høyre, n/2 data mot venstre, m/2 data opp, og m/2 data ned. Til sammen 4(n+m). Så lenge n og m er like store så blir det det samme for begge, men om m er større vil 1x4-partisjonen vinne, og om m er mindre enn n så redefinerer man bare partisjonsretningen og 1x4 vinner uansett. 2x2 har også flere mindre overføringer. Et annet poeng er at i Lattice-Boltsmann så propagerer data på skrått, så 2x2 er nødt til å sende litt data til alle hjørnene også. Man har alle til alle kommunikasjon som kompliserer datarutingen og øker antallet overføringer unødig. For 1x4 vil ikke skråkommunikasjon komplisere bildet i det hele tatt.

Occupancy

Problemet med occupancy var ikke høyt prioritert i utviklingen før kjøretidssammenligninger med en tidligere utviklet D2Q9-implementasjon viste seg å yte nesten dobbelt så godt som D3Q19, når man målte tidsforbruk for å beregne en lattice på samme mengde byte. D3Q19-koden har 2 lese- og 2

skriveoperasjoner per byte per iterasjon, og D2Q9-koden har 1 lese- og 1 skriveoperasjon per byte per iterasjon. D2Q9 har derimot dårlig coalescing så den teoretiske båndbredden skal være ganske lik for begge to. Begge gjør også tilnærmet samme mengde arbeid per innleste byte, så i utgangspunktet burde begge to yte like godt. De er riktignok ikke testet på samme hardware, 2D-koden er kjørt på et GTX260 kort og 3D-koden på et Tesla C1060, men disse kortene er relativt like. GTX260 har litt smalere men raskere minnebuss (til sammen 10% raskere minnehastighet) og 10% færre GPUkjerner. Ikke nok til å forklare en ytelsesdobling altså.

For D3Q19 trenger hver kernel 19 register til å lagre hastighetsvektorer, samt 5 til å lagre makroskopisk hastighet og tetthet. Om man vil holde seg under 32, har man 8 register igjen til alle beregninger, pluss 7 flyttall som kan lagres i cache. (Det burde vært 8, men kernelen satte av litt cache til seg selv). For D2Q9 derimot trengs bare 9+5 tall til minimal datalagring, så sjansen er stor for at occupancy er bedre for D2Q9, og det kan være årsaken til at den yter bedre.

En annen ting som sannsynliggjør at occupancy kan spille inn, er at tester på D3Q19 mellom to kort med samme GPU, men der det ene kortet har 40% høyere minnebåndbredde, viste en ytelsesforskjell på ca. 40%. Den totale ytelsen er under det man ville forvente om minnebussen gikk for fullt 100% av tiden. Det er derfor grunn til å tro at man kan tjene på flere tråder som kan køe opp minneforespørsler.

Relax kernelen viste seg å bruke minimalt med cacheminne, men trengte til gjengjeld 62 register. 62 register er bare godt nok til 25% occupancy **[11]** (faktisk enda mindre for enkelte blokk-størrelser), og det setter i tillegg en øvre begrensing på størrelsen på kernelblokkene (og ergo latticen) på 256. For å komme opp til 50% som er neste hakk på stigen kreves det at man kutter ned til 32 register. Forsøk på å lagre mest mulig i cache, omskriving av kernelen slik at den gjorde seg ferdig med variable raskest mulig så de kunne gjenbrukes og mest mulig forenkling av ligninger, førte til en nedgang til 48. Nok til å kunne kjøre litt større latticer, men ikke nært nok til å få bedre occupancy annet enn helt marginalt for enkelte blokkstørrelser. Den endelige løsningen ble å markere en bestemt variabel som "volatile". Det vil si at kompilatoren tvinges til å undersøke registerverdien til en variabel hver gang den brukes i et uttrykk, istedenfor å bruke ekstra registerplasser på prekalkulering for å spare tid. Dette fikk registerbruken umiddelbart ned på 32 uten andre optimaliseringer overhodet.

Kernelen som tar seg av propagate-fasen viste seg å bruke 27 register og 40 byte cache. 27 register er godt nok til en occupancy på 50%. 40 byte cache er derimot bare godt nok til 25%, og strengeste begrensing er det som gjelder. Ved å bruke trikset med å markere variable som "volatile" gikk registerbruken ned til 16 som er godt nok til 100% occupancy. Det er to muligheter til å redusere cachebruken. Godta litt uncoalesced minneaksess, eller gjenbruke noe av cachen. Uncoalesced minneaksess vil føre til halv båndbredde på flere aksesser, og gjenbruk vil føre til at man må kjøre flere synkroniseringsbarrierer for å garantere at alle trådene er ferdige før data overskrives.

Resultatet av disse optimaliseringene var mellom 10 til 20% forbedring som er noe mindre enn jeg hadde håpet på. Med unntak av uncoalesced minneaksess i propagate kernelen ga alle endringene positive resultat. Resultatene av forbedringene er vist og diskutert nærmere i kapittelet "Ytelse".

(3) Optimalisere instruksjoner og kontrollflyt

Det er ikke så mye man kan gjøre for å øke instruksjonsflyten i Lattice-Boltzmann. Heltallsmodulus-operasjoner er unngått i kritisk kode og isteden brukes noen reduserte instruksjoner for å oppnå samme resultat der det trengs. Noen steder er ligninger skrevet om til å bruke multiplikasjon istedenfor divisjon, og for-løkker er rullet ut.

Med tanke på flytkontroll er det to situasjoner som krever særskilt behandling, og som derfor kan gjøre at en if-blokk går i ulik retning. Det ene er trykkgrensebetingelse og det andre er bounceback-grensebetingelse. Koden er designet slik at trykkbetingelsen innføres på innkommende xz-plan, og siden warpen går lineært med x vil alle trådene i samme warp gå samme vei i den if-blokken. Når det gjelder bounceback er propagatefasen utført på en slik måte at noden allerede ligger i ferdig bounceback tilstand. Om alle nodene tilhørende en half-warp er bounceback-noder sparer man litt tid siden ingen data trenger overføres. Om bare noen noder er bounceback-noder, fører det til at noen tråder står på vent mens resten utfører relax fasen, og man verken vinner eller taper ytelse.

Ytelse

Den vanligste måleenheten for ytelse når det kommer til Lattice-Boltzmannløsere er Mega Lattice Updates Per Second (MLUPS) som vil si #[millioner latticenoder oppdatert per sekund]. Den gir en god intern sammenligning mellom Lattice-Boltzmannmetoder av samme dimensjon og type. Ytelsen er også uttrykt som forholdet mellom effektiv og maksimal teoretisk minnebåndbredde for å få en pekepinn på hvor godt hardwaren utnyttes. Tester er kjørt for noen ulike gridstørrelser på 3 ulike grafikkort.

- Ett Nvidia Tesla C1060: 4GB RAM, 240 kjerner på 602 MHz
- En Nvidia Tesla S1070 server, som i praksis er 4 stk C1060 kort i en boks
- Ett Nvidia GTX 280 kort, som har akkurat samme chip som C1060, men 40% raskere klokkefrekvens på minnebuss og bare 1GB RAM

Teslakortene er dedikerte GPGPUkort. De har ikke skjermtilkobling, mye mere RAM enn vanlige skjermkort og koster mye mere enn vanlige forbrukerkort. GTX280 er et helt standard konsument-grafikkort beregnet på spill.

Resultatene er sammenlignet med en egenprodusert CPUimplementasjon. CPUimplementasjonen er en standard trippel-for-løkke implementasjon. Den er testet for både en og fire CPUkjerner på en 2.83 GHz Intel Q9550.

Målingene er gjort ved å ta tiden på 100 fulle iterasjoner på noen ulike latticestørrelser. Unntaket er for små latticer på GPU og store latticer på CPU, der det er kjørt henholdsvis flere og færre iterasjoner. Grunnen til det er målenøyaktighet på GPU og tidsbesparing på CPU. Resultatene fra de målingene er normert for 100 iterasjoner og skrevet inn i tabellen.

For tidsmålingene på Tesla S1070 kortet er resultatet skrevet inn i raden som samsvarer med størrelsen på sub-latticen som behandles av hver enkelt GPU, og ikke størrelsen på den totale latticen som er 4x større (dobbelt så stor i y og z retning). Dette er gjort for å enklere se hvor stor kommunikasjonskostnaden er og fordi forskjellen i RAM størrelse gjør at det blir unaturlig å kjøre like problemer på 1 GPU som på 4 GPUer. I tabellen for MLUPS derimot er tallet beregnet ut fra den reelle latticestørrelsen.

Alle GPU resultatene er for enkeltpresisjons flyttall, mens for CPU er det testet for både enkel og dobbel presisjon. De er da markert med henholdsvis **32** og **64**.

To av kolonnene er merket med **opt**. Det står for optimalisering og omhandler de endringene som er beskrevet i kapittelet "Lattice-Boltzmann på GPU - (2) Optimalisere minnetilgang" for å forbedre occupancy. Ved å tvinge fram gjenbruk av register gikk occupancy opp fra 25% til 50% på relax-kernelen. På propagate kernelen gikk det fra 25% til 100% ved å framtvinge registergjenbruk og spare på cache ved å bruke det samme minnet om igjen.

For de radene der det mangler resultater er det fordi det ikke var nok RAM til å kjøre hele problemet.

Tid i ms for 100 tidsiterasjoner på ulike latticestørrelser

X-grid	Y-grid	Z-grid	1xCPU 64	1xCPU 32	4xCPU 64	1xTesla	GTX 280	1xTesla opt	GTX 280 opt	4xTesla*
			Tid for 100 iterasjoner [ms]							
128	128	64	52000	50000	18000	715	500	694	411	895
128	256	64	149000	120000	36000	1414	975	1370	806	1692
256	256	64	298000	230000	80000	3014	2225	2731	1766	3454
256	256	128	560000	650000	132000	5956	4464	5490	3540	6527
256	512	128				13792		12342		14652
256	512	256								28571

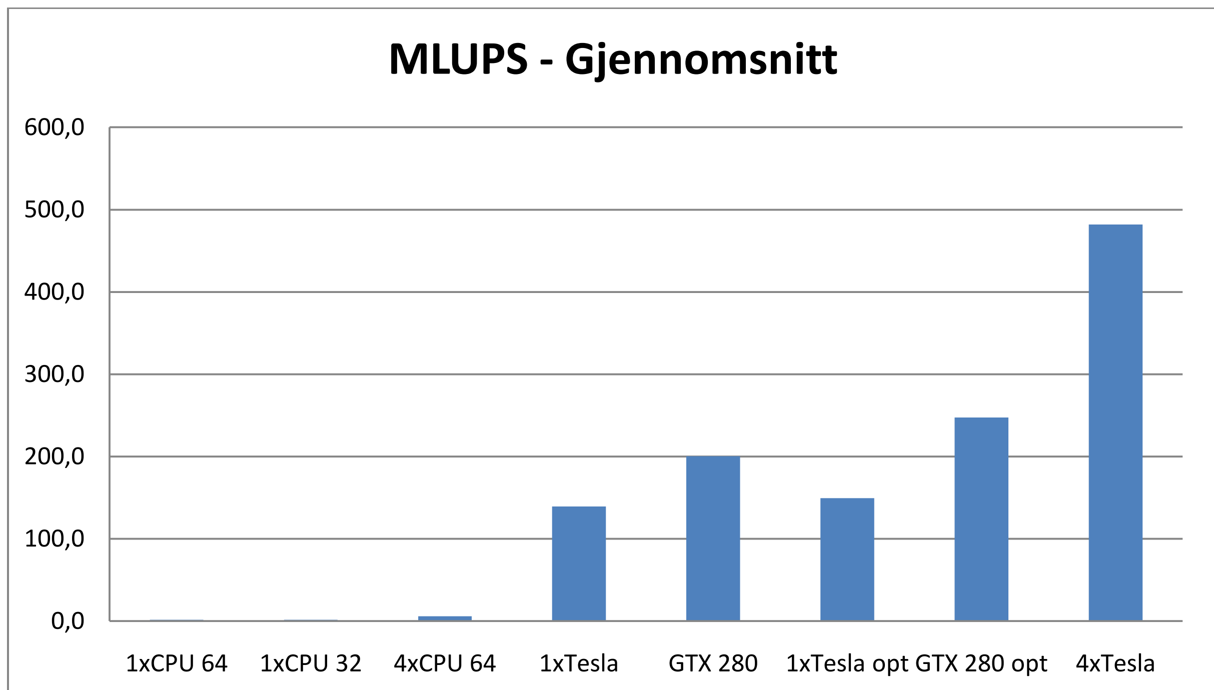
Gjennomsnittlig MLUPS ytelse for ulike latticestørrelser

X-grid	Y-grid	Z-grid	1xCPU 64	1xCPU 32	4xCPU 64	1xTesla	GTX 280	1xTesla opt	GTX 280 opt	4xTesla*
			MLUPS [Mega Lattice Updates Per Second]							
128	128	64	2.0	0.2	5.8	147	210	151	255	469
128	256	64	1.4	1.7	5.8	148	215	153	260	496
256	256	64	1.4	1.8	5.2	139	189	154	238	486
256	256	128	1.5	1.3	6.4	141	188	153	237	514
256	512	128				122		136		458
256	512	256								470

Relativ båndbreddeutnyttelse for ulike latticestørrelser

X-grid	Y-grid	Z-grid	1xTesla 102GB/s	GTX 280 142GB/s	1xTesla opt 102GB/s	GTX 280 opt 142GB/s	4xTesla 4x102GB/s
			Minnebåndbreddeutnyttelse [%]				
128	128	64	41 %	42 %	42 %	51 %	33 %
128	256	64	41 %	43 %	43 %	52 %	34 %
256	256	64	39 %	38 %	43 %	47 %	34 %
256	256	128	39 %	38 %	42 %	47 %	36 %
256	512	128	34 %		38 %		32 %
256	512	256					33 %

** - Tidsmålinger for 4xTesla er angitt for størrelse på sublattice per GPU. MLUPS er angitt for total latticestørrelse*



Figur 18: Gjennomsnittlig MLUPS ytelse

GPU-optimalisering

Testene viser at den occupancy-optimaliserte koden gir en marginal ytelsesforbedring for Teslakortet på rundt 10%. For GTX280-kortet derimot var tallet over 20%. Den eneste forskjellen på disse kortene bortsett fra at Tesla har 4x ganger mere minne, mangler skjermutgang og er dyrere, er at Tesla har 102GB/s minnebåndbredde mens GTX280 har 142 GB/s, (samme buss-bredde men høyere klokkefrekvens) og om man ser på den uoptimaliserte koden så ser man da også ca. 40% ekstra ytelse på GTX280. På den optimaliserte koden derimot yter GTX280 nesten 70% bedre. Dette er tall jeg har vansker med å forstå. Om flaskehalsen i beregningene var flyttallsytelse ville man ikke se noen forskjell på Tesla og GTX280 siden disse har akkurat samme chip. Man ville heller ikke se noen særlig forbedring på å øke occupancy. Om flaskehalsen var minnelatens ville man gjerne tjene mere på å øke occupancy på det kortet med mest latens. Om flaskehalsen var minnebåndbredde, hvilket man forøvrig ser av tabellen for båndbreddeutnyttelse at det ikke er, ville man se en 40% forskjell på de to kortene uansett kode.

Relativ utnyttelse av minnebåndbredde indikerer at det kanskje kan være mere å hente. I en oppgave av Eirik Aksnes **[1]** har han oppnådd maksimal ytelse på 180 MLUPS på en lignende GPU (Quadro FX5800), så det er grunn til å tro at koden kan optimaliseres ytterligere. Det var derimot for en litt annen type problem med mange flere kollisjonsnoder, hvilket gjør en direkte sammenligning vanskelig.

CPU vs. GPU

For enkeltkjerneimplementasjonen er hastighetsøkningen fra CPU til GPU rundt 100 (ganger, ikke prosent!) for Teslakortet. GTX280 klarer 180 ganger i best case. For utstyr til tilnærmet samme pris

er dette mildt sagt imponerende. Lattice-Boltzmann er like godt egnet for parallellisering til CPU som GPU og om man tar alle 4 kjerner i bruk er man nede på henholdsvis 25 og 45 ganger ytelsesforbedring for Tesla og GTX280, som fremdeles er svært bra. Det skal sies at koden min ikke har utnyttet kraften i CPUen til fulle. I en oppgave av Peter Bailey **[2]** henvises det til quad-core CPUkode som oppnår 9 MLUPS. Da er ytelsen nede fra imponerende 100 ganger til svært respektable 17 for et enkelt Teslakort.

Direkte sammenligning mellom CPU og 4xTesla er litt urettferdig siden det dreier seg om en standard forbruker-CPU vs. en GPU-server som kostet rundt 80 000 kr i sin tid. Men for ordens skyld, 4xTesla greier 340 ganger mot enkeltkjerne, 80 ganger mot 4 kjerner og 55 ganger mot 4 kjerner optimert.

GPU vs. multiGPU

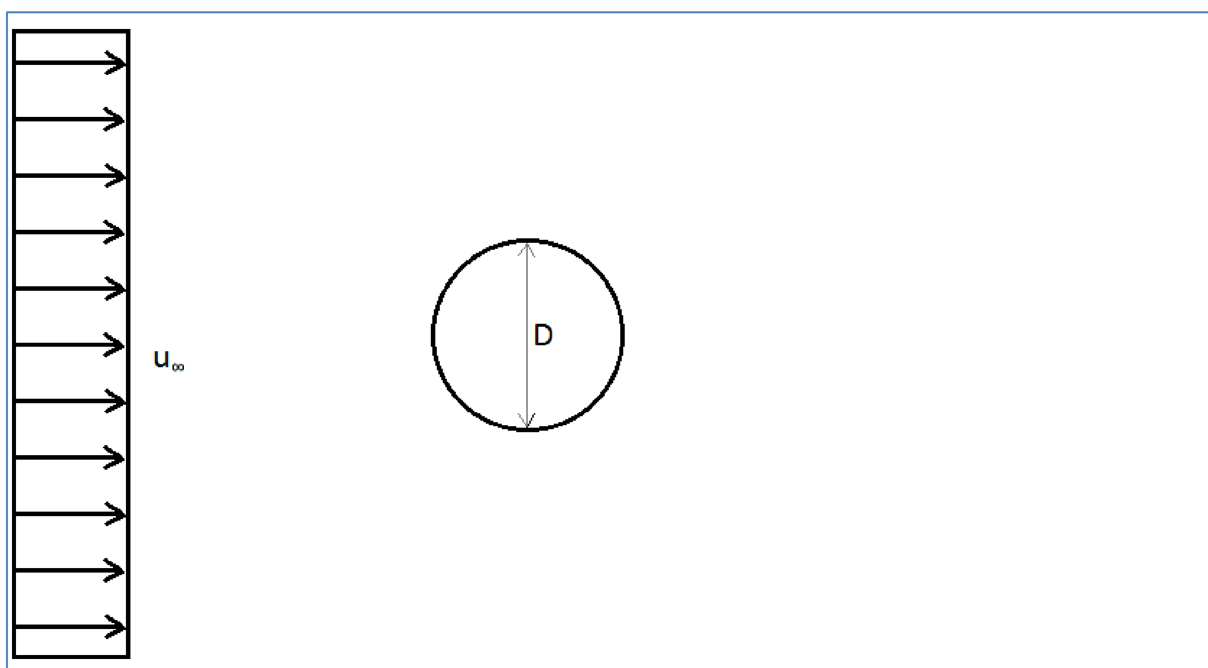
Et av hovedpoengene med oppgaven var å se hvor egnet GPUprogrammering var for å ta i bruk flere GPUer. NVIDIAS Best Practices Guide **[3]** oppgir dataoverføringer mellom GPU og CPU som svært kostbart og det var et åpent spørsmål hvor bra det ville gå. Lattice-Boltzmann har derimot alle forutsetninger for effektiv parallellisering, der mengden data som må deles per tidssteg er asymptotisk liten i forhold til beregningene som må gjøres. Sammenligning mellom 1xTesla og 4xTesla viser en kommunikasjonsoverhead på 25% for de minste latticene. Deretter synker den forutsigbart med 5% for hver dobling av latticen, og ender opp på 6%. Jeg fikk ikke data for de største latticene siden enkeltGPUversjonen ikke ville kjøre på så stor lattice. Hvorfor det gikk med fire GPUer og ikke med en har jeg ingen god forklaring på. Både teori og praksis er i alle fall klar på at overheaden nok ville blitt enda mindre for enda større latticer.

25% overhead er ikke veldig bra skalering, men man kjører ikke på flere GPUer om problemene er små, så jeg regner 6% som det endelige resultatet. Om tendensen fortsetter for enda større latticer ville den endt opp på 1-2%.

Et endelig resultat på rundt 500 MLUPS må kunne sies å være et godt resultat. Jeg har ikke sett noe tilsvarende i annen litteratur jeg har lest under arbeidet med denne oppgaven, men det har kun vært snakk om enkeltGPUimplementasjoner. Et interessant regnestykke man kan gjøre er å ta differansen i kjøretid for enkeltGPU og multiGPU for å isolere kommunikasjonskostnaden. Om man så legger til den kostnaden til kjøretiden for GTX280 og antar at man hadde hatt 4 kort til å kjøre samtidig ville man endt opp med en ytelse på 800 MLUPS. Da er man ganske nært å bryte GLUPS barrieren. Ikke dårlig for forrige generasjons hardware. Et googlesøk på "GLUPS Lattice Boltzmann" ga forøvrig kun 2 treff, begge urelaterte.

Validering av koden

Validering av koden er gjort ved å sammenligne enkelte nøkkeltall for strømningen med et velstudert og kjent strømningstilfelle. Uniform innstrømning mot en sirkulær sylinder (Se Figur 19). Ifølge Kjetil K. Heggernes **[12]** er det fire ulike kvalitative strømregimer for dette tilfellet, som avhenger av Reynoldstallet. Ved Reynoldstall fra 0 til under ~5 har man laminær strømning som følger sylinderen langs kanten. Ved Reynoldstall ~5 utvikles to symmetriske virvler bak sylinderen. Disse virvlene vil vare helt opp til Reynoldstall ~45. Da vil virvlene bli ustabile, den ene vil overta den andre, og man får en periodisk virvelavløsning bak sylinderen. Fra Reynoldstall ~45 til ~190 vil Strouhaltallet stige jevnt fra 0,12 til 0,19. Mellom Reynoldstall ~190 og ~260 er det en overgangsfase der tredimensjonale effekter begynner å spille inn og fra Reynoldstall ~260 og opp til minst $O(10^5)$ har man periodisk virvelavløsning med Strouhaltall på ca. 0,2.

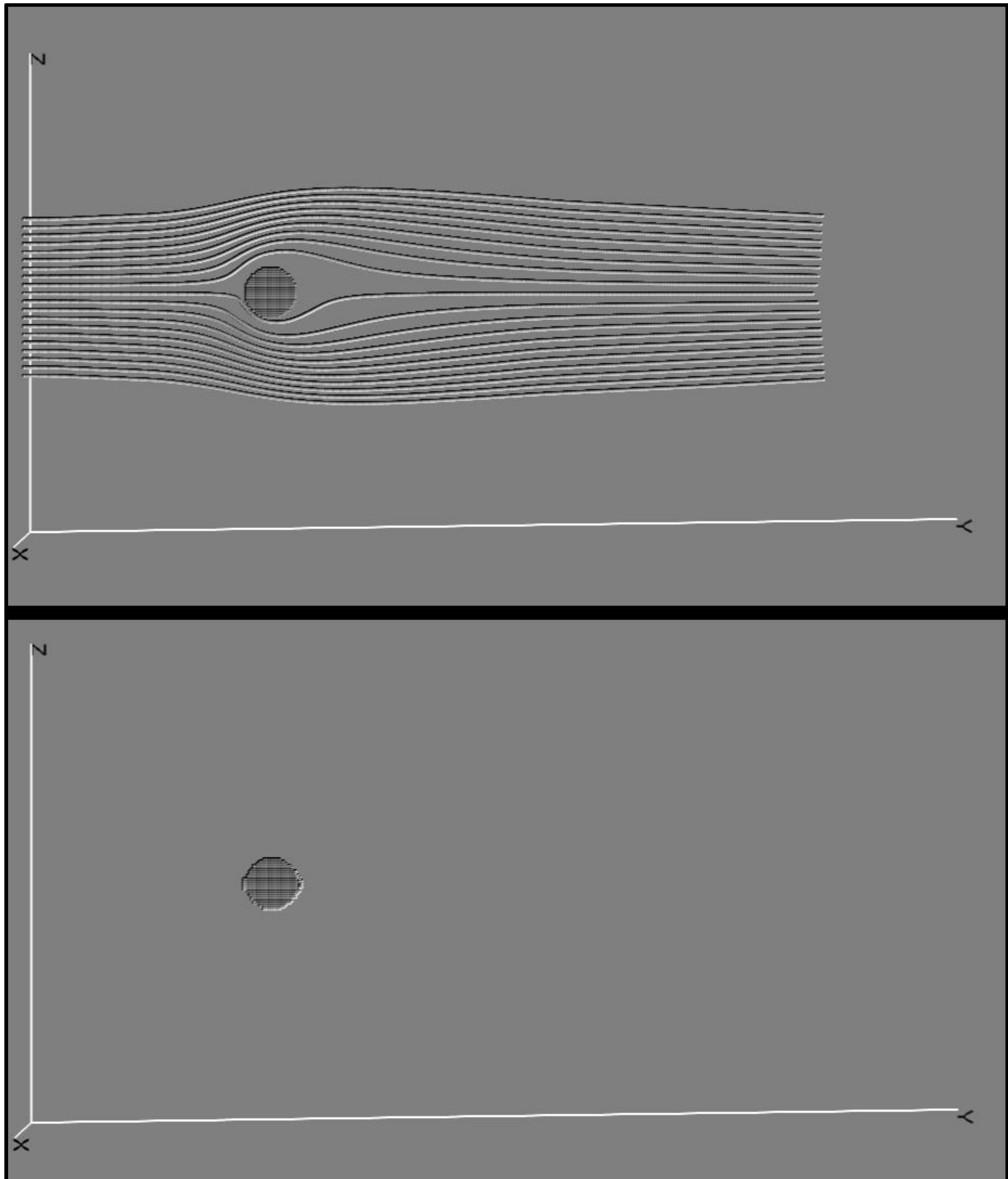


Figur 19: Sylinder i uniform strøm

Om koden min genererer den samme makroskopiske oppførselen er det grunn til å tro at koden, og metoden er korrekt. For detaljer rundt enhetsregning i Lattice-Boltzmann se paper av Jonas Latt **[6]**.

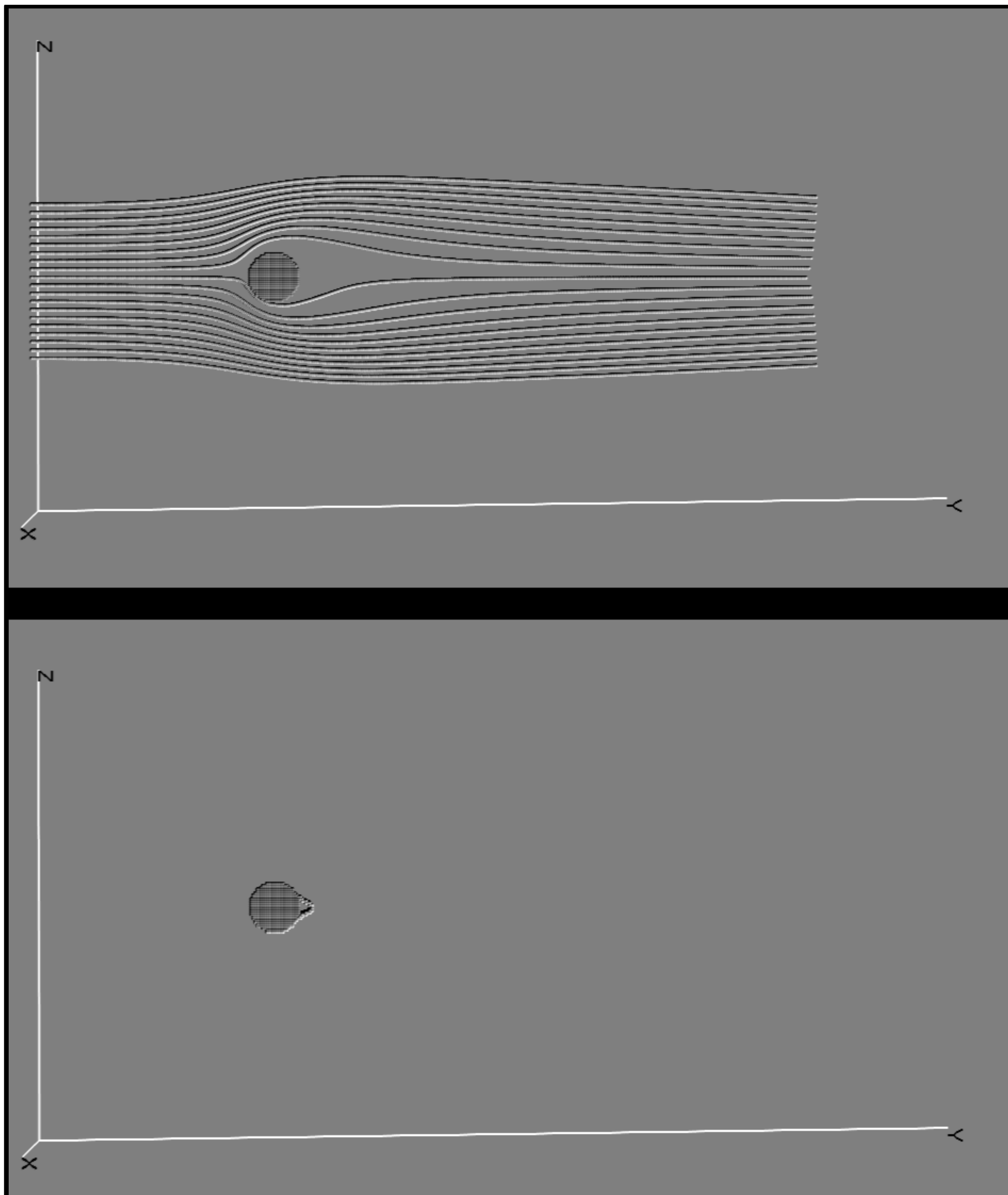
Reynoldstallregimer

Figur 20 viser strømlinjer og Vy-isoflate for Reynoldstall 5, på en [64 x 512 x 256] lattice. Simuleringen er kjørt helt til strømningsbildet ikke lengre viste synlige tegn til å utvikle seg videre. Av strømlinjene ser man at strømningen henger litt bak sylinderen. Isoflaten for y-hastigheten er tegnet for laveste Vy-verdi. Siden laveste Vy-verdi er positiv viser det at man ikke har tilbakefallsstrømning bak sylinderen, og ergo ikke virvling.

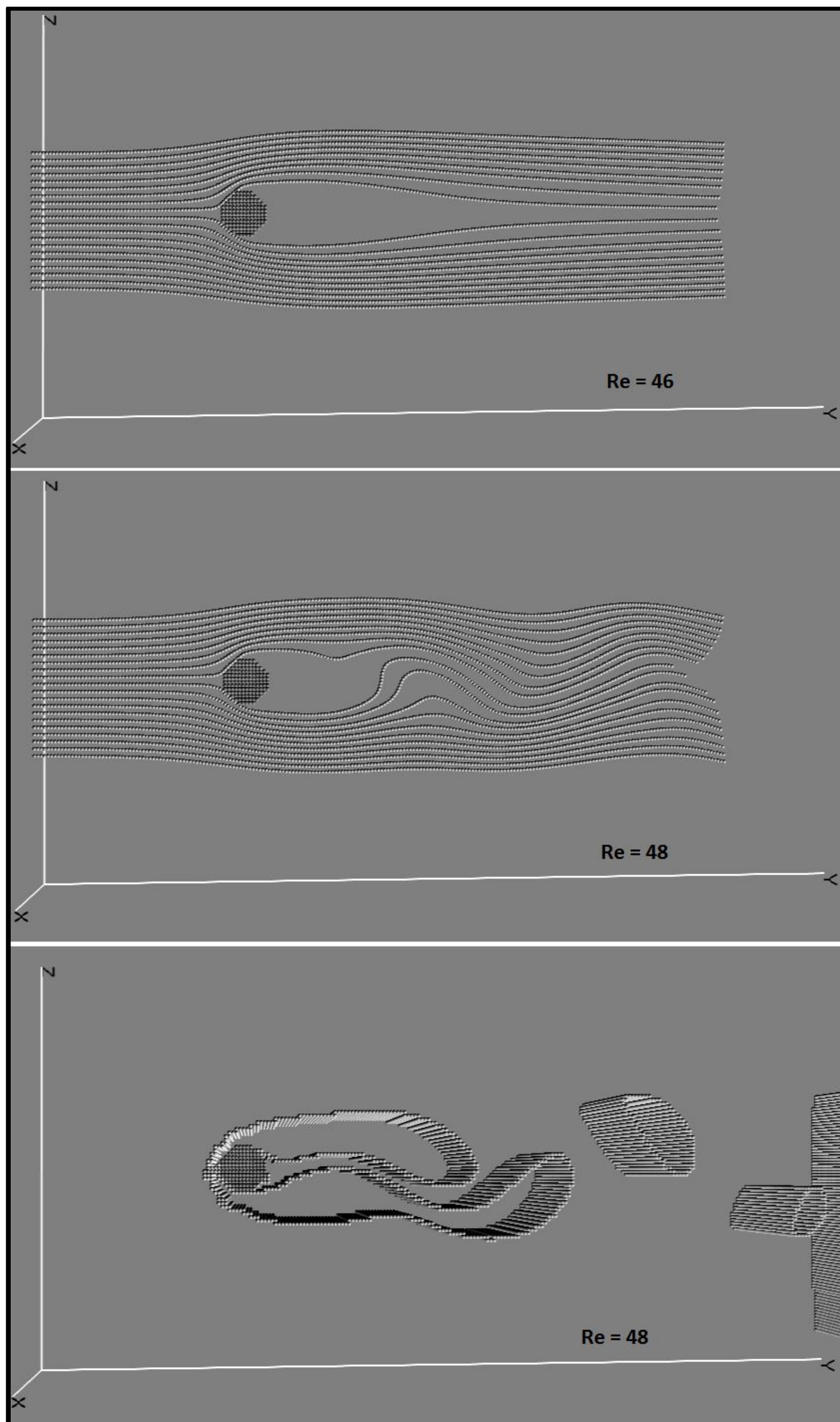


Figur 20: Strømlinjer og Vy isoflate for Re 5

Figur 21 viser strømlinjer og Vy-isoflate for Reynoldstall 10, på en [64 x 512 x 256] lattice. Strømlinjene henger litt lengre bak enn for Re 5. Isoflaten for Vy er tegnet for en negativ verdi, og den viser at man har en liten tilbakefallsstrøm som betyr at virvlingen har begynt. At den symmetriske virvlingen begynner for et høyere Reynoldstall enn teorien forutsier, kan ha å gjøre med at grensesjiktet rundt sylinderen ikke er godt nok oppløst. Et vanlig problem med fluidberegninger på regulære grid.



Figur 21: Strømlinjer og Vy isoflate for Re 10



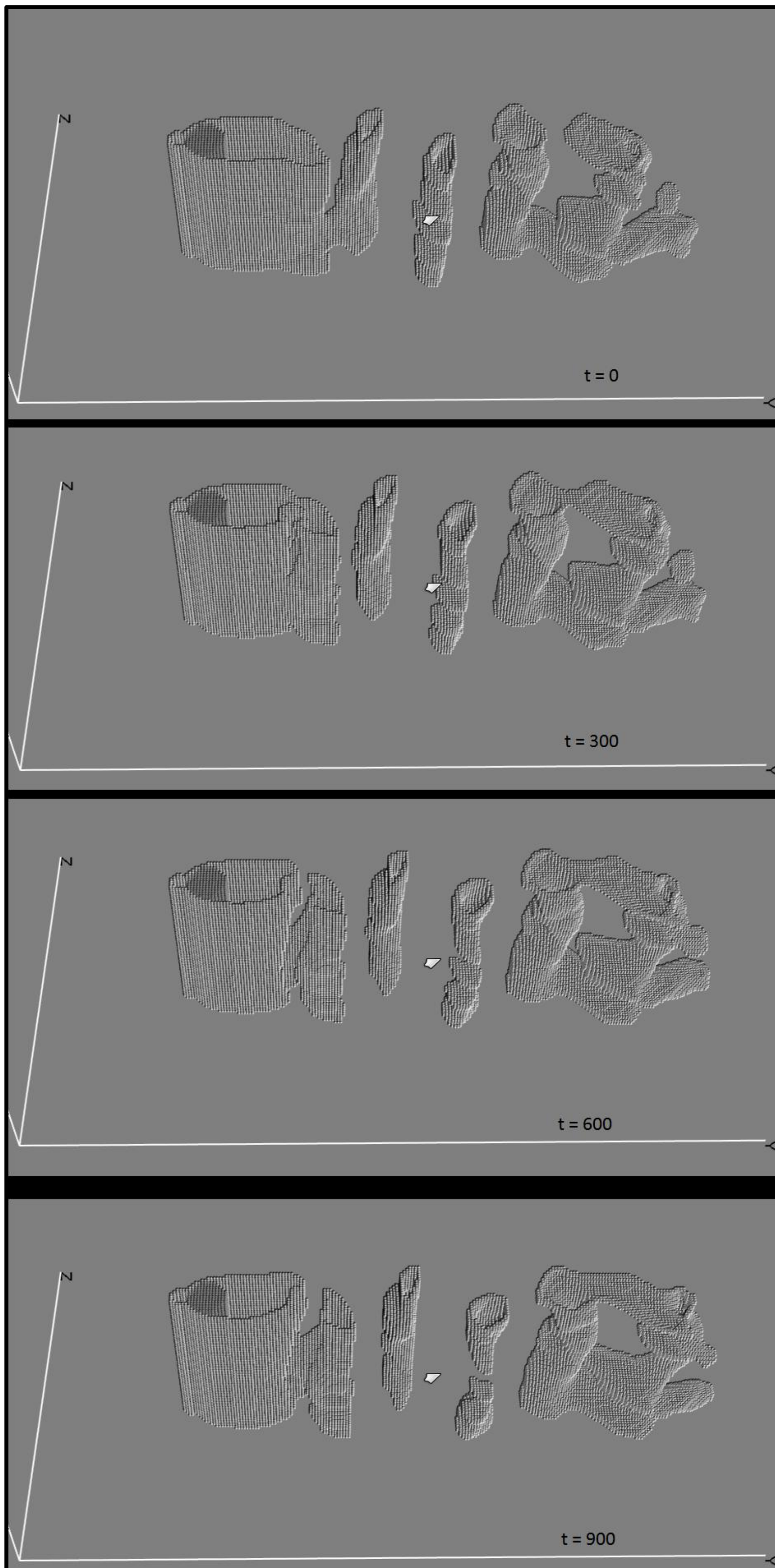
Figur 22: Strømlinjer og skalarcurl-isoflate for Reynoldstall 46 og 48

Figur 22 viser strømlinjer og skalarcurl-isoflate for henholdsvis Reynoldstall 46 og 48, på en [64 x 256 x 128] lattice. Ved Re 46 ser man at strømningen er stasjonær og stabil. Først for Re 48 starter virvelavløsningen. Igjen ser man at koden undervurderer Reynoldstallet litt.

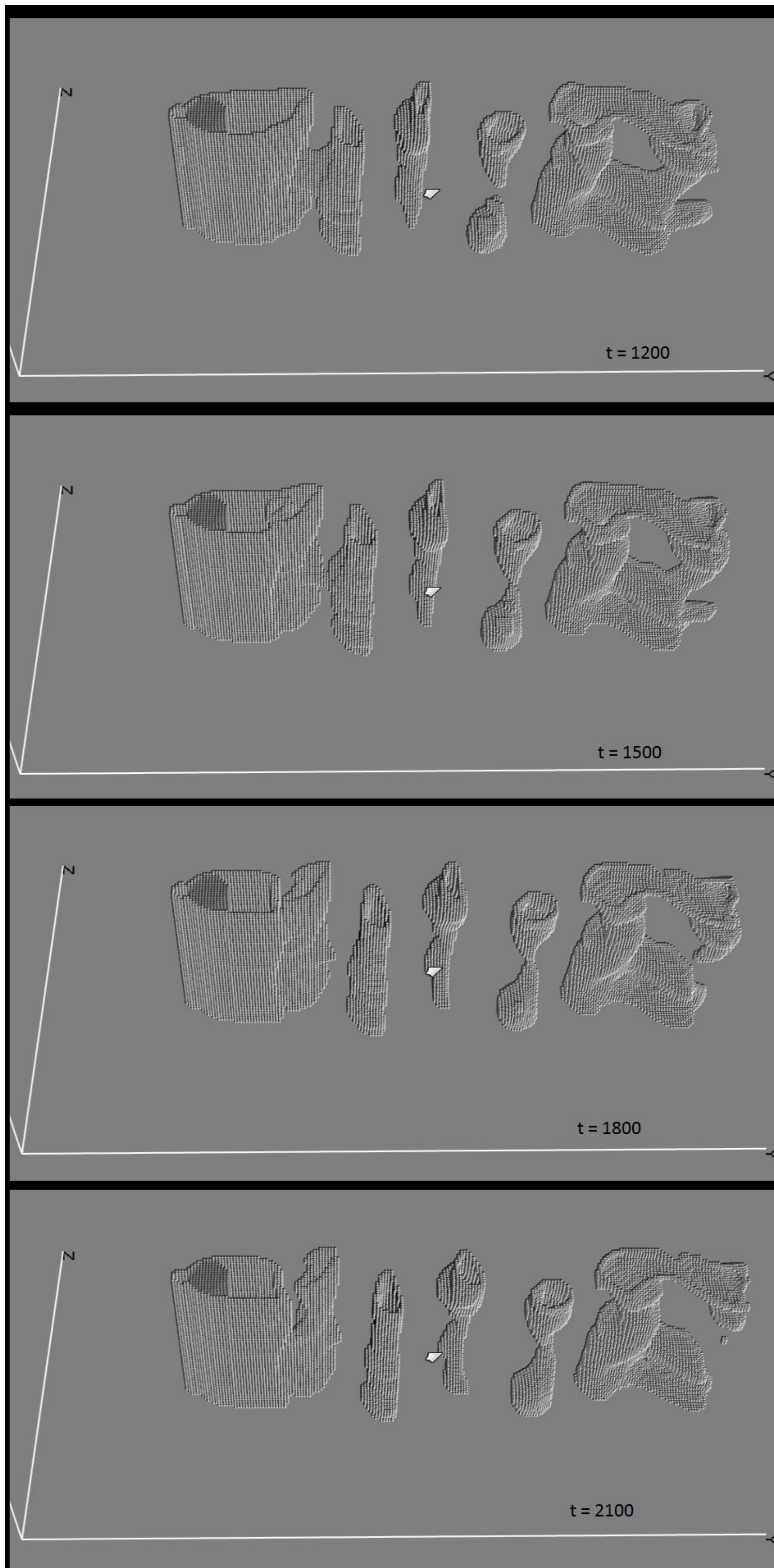
Strouhaltall

Numerisk og teoretisk verdi for virvelavløsningsfrekvens er sammenlignet for å se om de er sammenfallende. Tilfellet som er undersøkt er for en D = 12.8 δ_x sylinder på en latticeoppløsning [64 δ_x x 256 δ_x x 128 δ_x]. Kinematisk viskositet $\nu = 0.00083752\ \delta_x^2/\delta_t$ og innstrømshastighet u_0 = 0.0209 δ_x/δ_t. Dette skal gi et Reynoldstall Re = 320. Selv medregnet tendensen til at dårlig grensesjiktoppløsning trekker ned det praktiske Reynoldstallet, kan man regne med å befinne seg i sonen der Strouhaltallet er lik 0,2 uavhengig av Reynoldstall. Den teoretiske virvelavløsningsperioden skal da være lik 3000 δ_t.

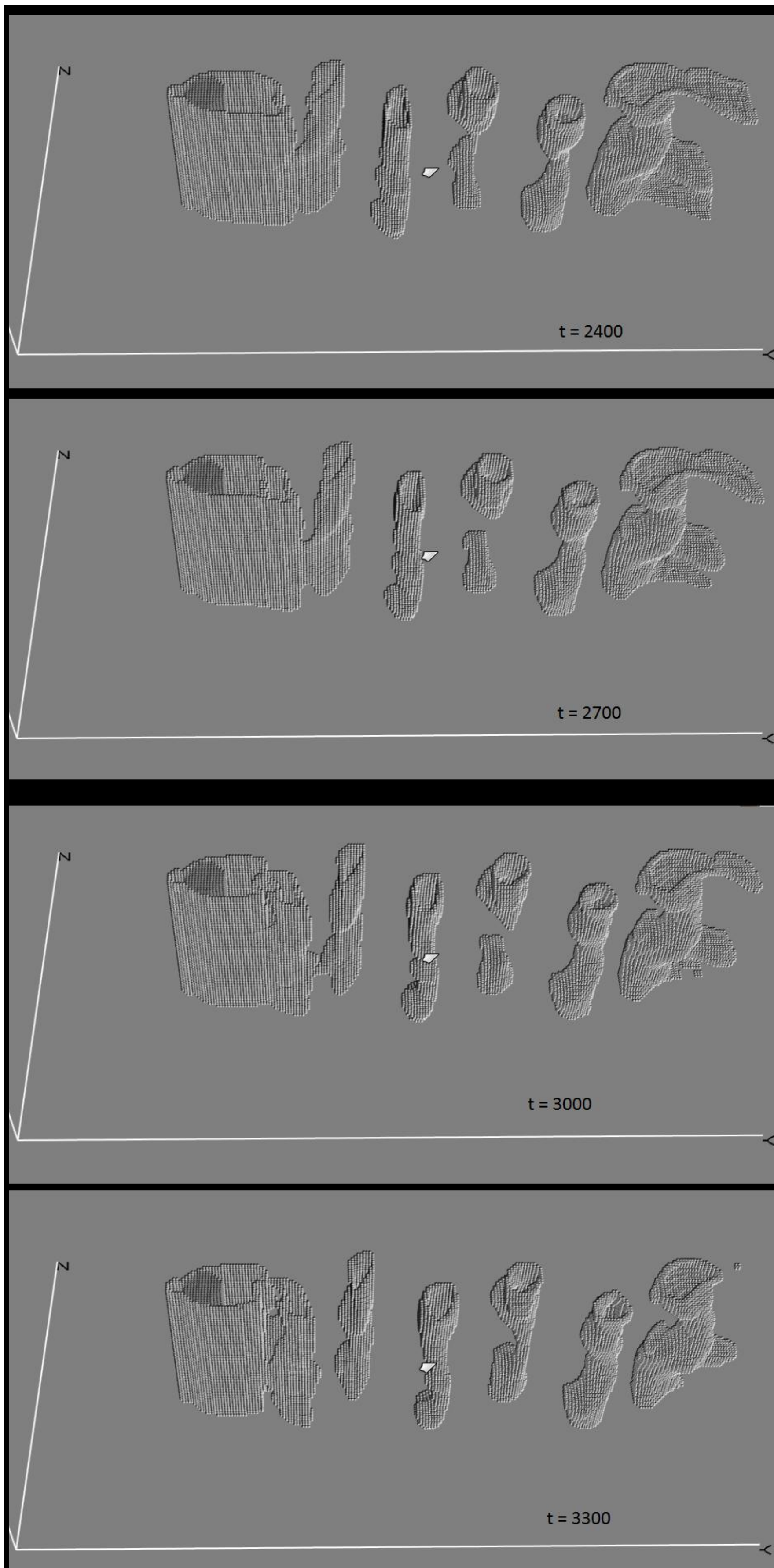
Figur 23, Figur 24 og Figur 25 på de neste sidene viser en tidsserie med isoflater for absoluttverdien av hastigheten. Man ser at tredimensjonale strømningseffekter kommer klart til uttrykk, og man kan ergo regne med å befinne seg i riktig Reynoldsregime. Bildene er tatt med 300 δ_t mellomrom. Om man følger med på det stasjonære referansepunktet midt i hvert bilde, ser man at det tar ganske nøyaktig 11 bilder fra en virvel forlater referansepunktet til en ny virvel fra samme side er på samme sted. Dette svarer til en periode på 3300 δ_t, som er 10% over det teoretiske estimatet. Om man tar høyde for litt usikkerhet i Strouhaltallet, og diskretiseringsunøyaktighet som kommer av å uttrykke en sirkulær sylinder som diskrete punkter i et uniformt grid, er det et relativt godt treff.



Figur 23: Virvelavløsningstidsserie av absolutthastighet-isoflate med 300 tidsstegsintervall ved Re 311



Figur 24: Virvelavløsningstidsserie av absolutthastighet-isoflate med 300 tidsstegsintervall ved Re 311



Figur 25: Virvelavløsningstidsserie av absolutthastighet-isoflate med 300 tidsstegsintervall ved Re 311

Konklusjon

Det er 4 tall som på en enkel måte oppsummerer denne oppgaven.

500 MLUPS maksytelse, 6% parallelliseringsoverhead, 340 ganger speedup i forhold til naiv CPU-implementasjon og 55 ganger speedup i forhold til optimalisert CPU-implementasjon. Det viser at GPGPU er verdt å satse på, og at man også kan få gode resultater ved å bruke flere GPUer sammen. Det krever litt mere planlegging for å skrive et godt program for GPU, men så lenge man unngår de verste fallgruvene får man mye igjen for det.

Se forøvrig "Appendiks B - Minneoptimering" for resultatet av noen optimeringer som grunnet tidspress ikke ble redigert inn i selve rapporten.

Alt er derimot ikke udelt positivt og kapitlet om validering antydet noen typiske svakheter med både GPGPU og Lattice-Boltzmann. God oppløsning av grensesjikt krever enten irregulære grid, eller ekstremt høy oppløsning av hele domenet. Både GPGPU og Lattice-Boltzmann er dårlig egnet for irregulære grid, og det hjelper ikke at GPGPU regner mye raskere om man er nødt til å ha mye finere grid for å oppnå samme nøyaktighet som en CPUkode med variabelt grid.

Et alternativ for å løse dette problemet er å la GPUen regne på et høyoppløst grid umiddelbart rundt objektet, og så bruke CPU (eller en annen GPU?) til å regne på et grovere grid rundt. En annen mulighet er å bruke en vanlig CPUbasert Navier-Stokesløser i grensesjiktet, og så koble resultatet fra grensesjiktberegningene til en GPUløser som regner på det grovere gridet rundt.

Litteraturliste

[1] "Porous Rock Simulations and Lattice Boltzmann on GPUs" - Eirik O. AKSNES og Anne C. ELSTER

[2] "Accelerating Lattice Boltzmann Fluid FlowSimulations Using Graphics Processors" - Peter Bailey, Joe Myre, Stuart D. C. Walsh, David J. Lilja, Martin O. Saar

[3] "NVIDIA CUDA C Programming Best Practices Guide CUDA Toolkit 2.3"

[4] "Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation" - Xiauyi He og Li-Shi Luo

[5] "anb.f" - Lattice-Boltzmann kildekode i fortran av Joerg Bernsdorf

[6] "Choice of units in lattice Boltzmann simulations" - Jonas Latt

[7] "NVIDIA CUDA Programming Guide Version 2.3.1"

[8] "A Practical Introduction to the Lattice Boltzmann Method" Alexander J. Wagner

[9] "Lattice Boltzmann Model for the Incompressible Navier-Stokes Equation" - Xiaoyi He og Li-Shi Luo

[10]"http://users.polytech.unice.fr/~lingrand/MarchingCubes/accueil.html" - Diane LINGRAND

[11]"CUDA GPU Occupancy Calculator v1.5"

[12]"Numerical Simulations of three-dimensional viscous flow around marine structures" - Kjetil K. Heggernes

Appendiks A - D3Q19 Lattice-konstanter

$$c_s = \frac{1}{\sqrt{3}}, \qquad w_a = \begin{Bmatrix} 1/3 & a = 0 \\ 1/18 & a = [1,6] \\ 1/36 & a = [7,18] \end{Bmatrix}$$

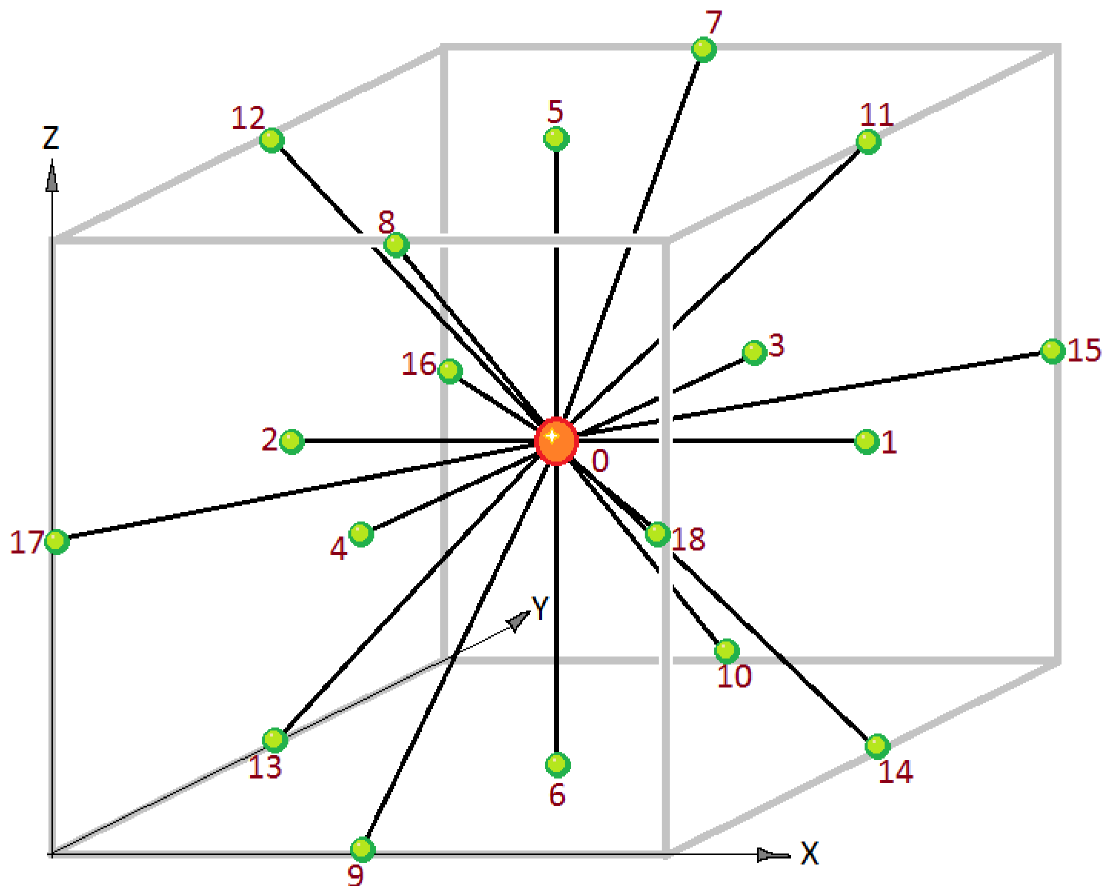
$\boldsymbol{e_0} = (0,0,0)$, $\boldsymbol{e_1} = (c,0,0)$, $\boldsymbol{e_2} = (-c,0,0)$, $\boldsymbol{e_3} = (0,c,0)$, $\boldsymbol{e_4} = (0,-c,0)$

$\boldsymbol{e_5} = (0,0,c)$, $\boldsymbol{e_6} = (0,0,-c)$, $\boldsymbol{e_7} = (0,c,c)$, $\boldsymbol{e_8} = (0,-c,c)$, $\boldsymbol{e_9} = (0,-c,-c)$

$\boldsymbol{e_{10}} = (0,c,-c)$, $\boldsymbol{e_{11}} = (c,0,c)$, $\boldsymbol{e_{12}} = (-c,0,c)$, $\boldsymbol{e_{13}} = (-c,0,-c)$

$\boldsymbol{e_{14}} = (c,0,-c)$, $\boldsymbol{e_{15}} = (c,c,0)$, $\boldsymbol{e_{16}} = (-c,c,0)$, $\boldsymbol{e_{17}} = (-c,-c,0)$

$\boldsymbol{e_{18}} = (c,-c,0)$

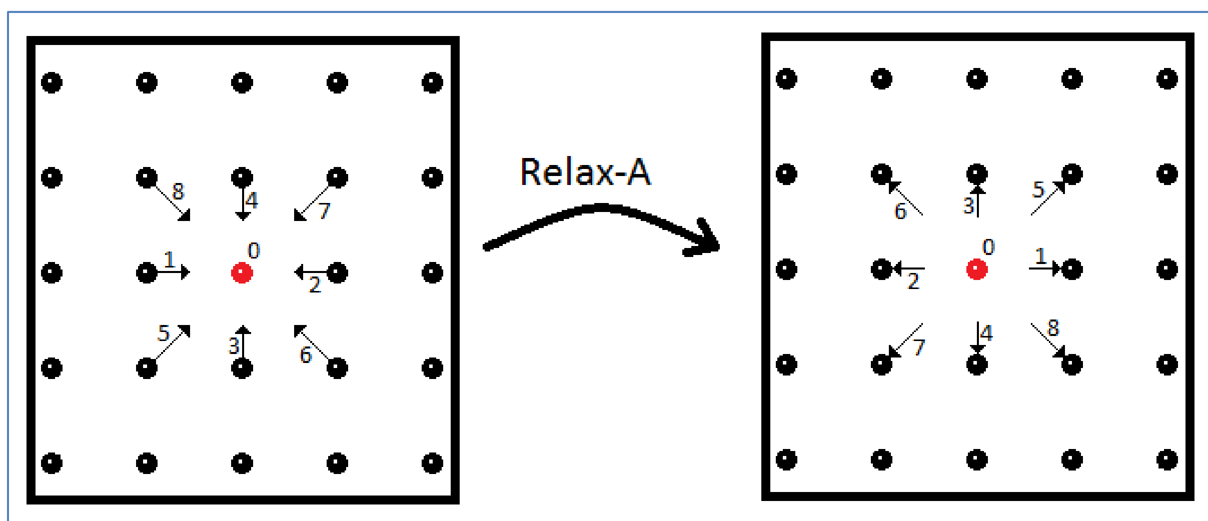


Appendiks B - Minneoptimering

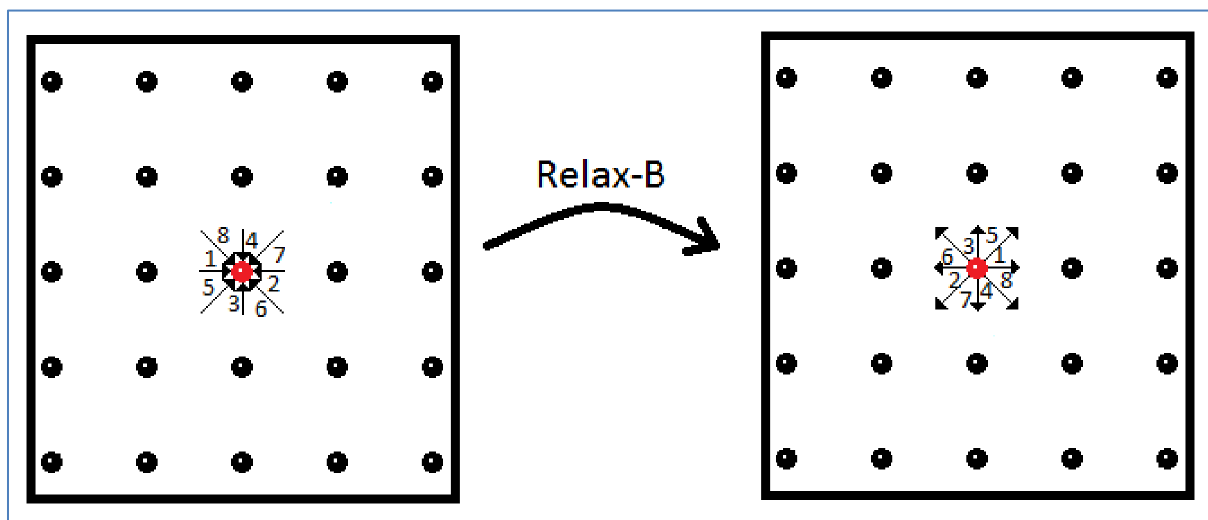
I kapittelet "Lattice-Boltzmann på GPU - (1) Maksimere parallell utførelse" diskuteres to ulike datastrukturer med ulike fordeler og ulemper. Den ene krever 2 tabeller for å lagre latticedata og 2 aksesser til globalt minne per tidssteg. Den andre trenger kun 1 tabell for latticedata, men trenger til gjengjeld 4 globale minneaksesser. Dette appendikset beskriver en teknikk for å hente det beste fra begge metodene. En latticetabell og to minneaksesser per tidssteg. På grunn av kort tid til leveringsfristen har jeg valgt å legge det til et eget appendiks framfor å gjøre store endringer i rapporten.

Alternerende relax-optimering

Ved å alternere mellom av to ulike relax-rutiner som leser/skriver i ulike mønster i latticen kan man klare seg uten et eksplisitt propagate-steg. Relax A leser og skriver som vist i Figur 26.



Figur 26: Relax A



Figur 27: Relax B

Intuitivt kan man si at Relax_A utfører "propagate-relax-propagate", mens Relax_B utfører "relax". Relax_B er den samme rutinen som allerede er i bruk så alt man trenger å gjøre er å skrive Relax_A.

Når begge rutinene er ferdige har man lest og skrevet til sammen 4 ganger til globalt minne, og utført 2 tidssteg. Denne optimeringen har potensiale til å doble ytelsen sammenlignet med vanlig alternerende propagate-relax.

Ytelse for alternerende relax-optimering

Jeg har kun testet alternerende relax-optimering på en GPU. Grunnen til det er at multiGPUimplementasjonen krever kommunikasjon mellom partisjonene mellom hvert tidssteg. Siden minnelayouten er annerledes etter Relax_A og Relax_B ville det kreve to litt ulike rutiner for å bytte overflatedata og det har det ikke blitt tid til.

X-grid	Y-grid	Z-grid	1xTesla	GTX 280	1xTesla	GTX 280
			Tid for 100 iterasjoner [ms]		MLUPS	
128	128	64	343	251	306	418
128	256	64	686	507	306	414
256	256	64	1471	1108	285	379
256	256	128	2943	2219	285	378
256	512	128	6749		249	
256	512	256				

Sammenlignet med resultatene fra kapittelet "Ytelse" har Teslakortet fått en dobling i ytelse fra rundt 150 MLUPS til 300 MLUPS, mens GTX280 har en noe knapp dobling fra 250 MLUPS til 400 MLUPS.

Kildekode for propagateRelaxPropagate

```
#define real float

__global__ void propagateRelaxPropagate(real* lattice,bool* obstacle,int
NNN,int yo) {
      int I  = threadIdx.x;
      int J  = blockIdx.x*lx;
      int K  = blockIdx.y*lx*ly;
      if(!obstacle[I+J+K]){
            int Ip = (i+1)*(i != lx-1);
            int Im = (i-1)+(i == 0)*lx;
            int Jp = ((j+1)*(j != ly-1))*lx;
            int Jm = ((j-1)+(j == 0)*ly)*lx;
            int Kp = ((k+1)*(k != lz-1))*lx*ly;
            int Km = ((k-1)+(k == 0)*lz)*lx*ly;

            __shared__ real V1[lx];
            __shared__ real V2[lx];
            __shared__ real V11[lx];
            __shared__ real V13[lx];
            __shared__ real V14[lx];
            __shared__ real V12[lx];
            __shared__ real V15[lx];
            __shared__ real V17[lx];
```

```
            __shared__ real V18[lx];
            __shared__ real V16[lx];

            V1[Ip] = lattice[I +J +K + 1*NNN];
            V2[Im] = lattice[I +J +K + 2*NNN];

            V11[Ip]= lattice[I +J +Km+11*NNN];
            V13[Im]= lattice[I +J +Kp+13*NNN];

            V14[Ip]= lattice[I +J +Km+14*NNN];
            V12[Im]= lattice[I +J +Kp+12*NNN];

            V15[Ip]= lattice[I +Jm+K +15*NNN];
            V17[Im]= lattice[I +Jp+K +17*NNN];

            V18[Ip]= lattice[I +Jp+K +18*NNN];
            V16[Im]= lattice[I +Jm+K +16*NNN];

            __syncThreads();

            real V0 = lattice[I +J +K + 0*NNN];
            real V3 = lattice[I +Jm+K + 3*NNN];
            real V4 = lattice[I +Jp+K + 4*NNN];
            real V5 = lattice[I +J +Km+ 5*NNN];
            real V6 = lattice[I +J +Kp+ 6*NNN];
            real V7 = lattice[I +Jm+Km+ 7*NNN];
            real V8 = lattice[I +Jp+Km+ 8*NNN];
            real V9 = lattice[I +Jp+Kp+ 9*NNN];
            real V10= lattice[I +Jm+Kp+10*NNN];

            real D =
V0+V1[I]+V2[I]+V3+V4+V5+V6+V7+V8+V9+V10+V11[I]+V12[I]+V13[I]+V14[I]+V15[I]+
V16[I]+V17[I]+V18[I];
            real vx = (V1[I]-V2[I]+V11[I]-V12[I]+V14[I]-V13[I]+V15[I]-
V16[I]+V18[I]-V17[I])/D;
            real vy = (V3-V4+V7-V8+V10-V9+V15[I]-V17[I]+V16[I]-V18[I])/D;
            real vz = (V5-V6+V7-V9+V8-V10+V11[I]-V13[I]+V12[I]-V14[I])/D;
            real v2 = vx*vx + vy*vy + vz*vz;

            real un1 =  vx;
            real un2 = -vx;
            real un3 =  vy;
            real un4 = -vy;
            real un5 =  vz;
            real un6 = -vz;
            real un7 =  vy+vz;
            real un8 = -vy+vz;
            real un9 = -vy-vz;
            real un10=  vy-vz;
            real un11=  vx+vz;
            real un12= -vx+vz;
            real un13= -vx-vz;
            real un14=  vx-vz;
            real un15=  vx+vy;
            real un16= -vx+vy;
            real un17= -vx-vy;
            real un18=  vx-vy;

            real t0 = 1.0f/3.0f;
            real t1 = 1.0f/18.0f;
            real t2 = 1.0f/36.0f;
```

```
V0 = V0 +     omega*(D*t0*(1.0f                              - v2*1.5f)-V0);
V1[I]=V1[I]+  omega*(D*t1*(1.0f+un1 *3.0f+un1 * un1*4.5f-v2*1.5f)-V1[I]);
V2[I]=V2[I]+  omega*(D*t1*(1.0f+un2 *3.0f+un2 * un2*4.5f-v2*1.5f)-V2[I]);
V3 = V3 +     omega*(D*t1*(1.0f+un3 *3.0f+un3 * un3*4.5f-v2*1.5f)-V3);
V4 = V4 +     omega*(D*t1*(1.0f+un4 *3.0f+un4 * un4*4.5f-v2*1.5f)-V4);
V5 = V5 +     omega*(D*t1*(1.0f+un5 *3.0f+un5 * un5*4.5f-v2*1.5f)-V5);
V6 = V6 +     omega*(D*t1*(1.0f+un6 *3.0f+un6 * un6*4.5f-v2*1.5f)-V6);
V7 = V7 +     omega*(D*t2*(1.0f+un7 *3.0f+un7 * un7*4.5f-v2*1.5f)-V7);
V8 = V8 +     omega*(D*t2*(1.0f+un8 *3.0f+un8 * un8*4.5f-v2*1.5f)-V8);
V9 = V9 +     omega*(D*t2*(1.0f+un9 *3.0f+un9 * un9*4.5f-v2*1.5f)-V9);
V10[I]=V10[I]+omega*(D*t2*(1.0f+un10*3.0f+un10*un10*4.5f-v2*1.5f)-V10[I]);
V11[I]=V11[I]+omega*(D*t2*(1.0f+un11*3.0f+un11*un11*4.5f-v2*1.5f)-V11[I]);
V12[I]=V12[I]+omega*(D*t2*(1.0f+un12*3.0f+un12*un12*4.5f-v2*1.5f)-V12[I]);
V13[I]=V13[I]+omega*(D*t2*(1.0f+un13*3.0f+un13*un13*4.5f-v2*1.5f)-V13[I]);
V14[I]=V14[I]+omega*(D*t2*(1.0f+un14*3.0f+un14*un14*4.5f-v2*1.5f)-V14[I]);
V15[I]=V15[I]+omega*(D*t2*(1.0f+un15*3.0f+un15*un15*4.5f-v2*1.5f)-V15[I]);
V16[I]=V16[I]+omega*(D*t1*(1.0f+un16*3.0f+un16*un16*4.5f-v2*1.5f)-V16[I]);
V17[I]=V17[I]+omega*(D*t2*(1.0f+un17*3.0f+un17*un17*4.5f-v2*1.5f)-V17[I]);
V18[I]=V18[I]+omega*(D*t2*(1.0f+un18*3.0f+un18*un18*4.5f-v2*1.5f)-V18[I]);

            if(blockIdx.x+yo == 0){
                  real w1 = t1*accel;
                  real w2 = t2*accel;
                  if(V4>w1 && V8>w2 && V9>w2 && V18[I]>w2 && V17[I]>w2){
                        V4-=w1;
                        V3+=w1;
                        V8-=w2;
                        V10+=w2;
                        V9-=w2;
                        V7+=w2;
                        V18[I]-=w2;
                        V16[I]+=w2;
                        V17[I]-=w2;
                        V15[I]+=w2;
                  }
            }

            __syncThreads();

            lattice[I +J +K + 0*NNN] = V0;
            lattice[I +J +K + 1*NNN] = V1[Im];
            lattice[I +J +K + 2*NNN] = V2[Ip];
            lattice[I +Jp+K + 3*NNN] = V3;
            lattice[I +Jm+K + 4*NNN] = V4;
            lattice[I +J +Kp+ 5*NNN] = V5;
            lattice[I +J +Km+ 6*NNN] = V6;
            lattice[I +Jp+Kp+ 7*NNN] = V7;
            lattice[I +Jm+Kp+ 8*NNN] = V8;
            lattice[I +Jm+Km+ 9*NNN] = V9;
            lattice[I +Jp+Km+10*NNN] = V10;
            lattice[I +J +Kp+11*NNN] = V11[Im];
            lattice[I +J +Kp+12*NNN] = V12[Ip];
            lattice[I +J +Km+13*NNN] = V13[Ip];
            lattice[I +J +Km+14*NNN] = V14[Im];
            lattice[I +Jp+K +15*NNN] = V15[Im];
            lattice[I +Jp+K +16*NNN] = V16[Ip];
            lattice[I +Jm+K +17*NNN] = V17[Ip];
            lattice[I +Jm+K +18*NNN] = V18[Im];
      }
}
```

Appendiks C - CUDA Kildekode

```
// All funksjonalitet relatert til lagring av resultat er fjernet av hensyn
// til leselighet.

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda_runtime.h>
#include <omp.h>


#define accel 0.00002f
// accel: Parameter for å justere akselerasjon på innstrømskant

#define omega 1.99f
// omega: Invers av dimensionless relaxation time.
// Bestemmer kinematisk viskositet

#define lx 64
#define ly 512
#define lz 256
// lx,ly,lz: Lattice størrelse i x,y og z retning for hver latticepartisjon



#define real float
#define k0 0.5f
#define k1 1.0f
#define k2 3.0f
#define k3 4.5f
#define k4 1.5f


__global__ void relax(real* V, bool* obstacle, volatile int NNN, int yo){
      int i = threadIdx.x + blockIdx.x*lx + blockIdx.y*lx*ly;
      if(!obstacle[i]){ // Dersom obstacle[x + y*lx + z*lx*ly] = true,
                        // er node(x,y,z) en bounceback-node, og trenger
                        // derfor ikke behandles av relax.
      real V0 =  V[i];
      real V1 =  V[i +  2*NNN];
      real V2 =  V[i +  1*NNN];
      real V3 =  V[i +  4*NNN];
      real V4 =  V[i +  3*NNN];
      real V5 =  V[i +  6*NNN];
      real V6 =  V[i +  5*NNN];
      real V7 =  V[i +  9*NNN];
      real V8 =  V[i + 10*NNN];
      real V9 =  V[i +  7*NNN];
      real V10 = V[i +  8*NNN];
      real V11 = V[i + 13*NNN];
      real V12 = V[i + 14*NNN];
      real V13 = V[i + 11*NNN];
      real V14 = V[i + 12*NNN];
      real V15 = V[i + 17*NNN];
      real V16 = V[i + 18*NNN];
      real V17 = V[i + 15*NNN];
      real V18 = V[i + 16*NNN];

real D = V0+V1+V2+V3+V4+V5+V6+V7+V8+V9+V10+V11+V12+V13+V14+V15+V16+V17+V18;
```

```
// lokal masse
      real vx = (V1-V2+V11-V12+V14-V13+V15-V16+V18-V17)/D;
// lokal hastighet x
      real vy = (V3-V4+V7-V8+V10-V9+V15-V17+V16-V18)/D;
// lokal hastighet y
      real vz = (V5-V6+V7-V9+V8-V10+V11-V13+V12-V14)/D;
// lokal hastighet z
      real v2 = vx*vx + vy*vy + vz*vz;

      real un1 =  vx;
      real un2 = -vx;
      real un3 =  vy;
      real un4 = -vy;
      real un5 =  vz;
      real un6 = -vz;
      real un7 =  vy+vz;
      real un8 = -vy+vz;
      real un9 = -vy-vz;
      real un10=  vy-vz;
      real un11=  vx+vz;
      real un12= -vx+vz;
      real un13= -vx-vz;
      real un14=  vx-vz;
      real un15=  vx+vy;
      real un16= -vx+vy;
      real un17= -vx-vy;
      real un18=  vx-vy;

      real t0 = 1.0f/3.0f;
      real t1 = 1.0f/18.0f;
      real t2 = 1.0f/36.0f;

      V0  = V0+ omega*(   D*t0*(k1                          - v2*k4) - V0);
      V1  = V1+ omega*(   D*t1*(k1 + un1 *k2 + un1 * un1*k3 - v2*k4) - V1);
      V2  = V2+ omega*(   D*t1*(k1 + un2 *k2 + un2 * un2*k3 - v2*k4) - V2);
      V3  = V3+ omega*(   D*t1*(k1 + un3 *k2 + un3 * un3*k3 - v2*k4) - V3);
      V4  = V4+ omega*(   D*t1*(k1 + un4 *k2 + un4 * un4*k3 - v2*k4) - V4);
      V5  = V5+ omega*(   D*t1*(k1 + un5 *k2 + un5 * un5*k3 - v2*k4) - V5);
      V6  = V6+ omega*(   D*t1*(k1 + un6 *k2 + un6 * un6*k3 - v2*k4) - V6);
      V7  = V7+ omega*(k0*D*t1*(k1 + un7 *k2 + un7 * un7*k3 - v2*k4) - V7);
      V8  = V8+ omega*(k0*D*t1*(k1 + un8 *k2 + un8 * un8*k3 - v2*k4) - V8);
      V9  = V9+ omega*(k0*D*t1*(k1 + un9 *k2 + un9 * un9*k3 - v2*k4) - V9);
      V10= V10+omega*(k0*D*t1*(k1 + un10*k2 + un10*un10*k3 - v2*k4) - V10);
      V11= V11+omega*(k0*D*t1*(k1 + un11*k2 + un11*un11*k3 - v2*k4) - V11);
      V12= V12+omega*(k0*D*t1*(k1 + un12*k2 + un12*un12*k3 - v2*k4) - V12);
      V13= V13+omega*(k0*D*t1*(k1 + un13*k2 + un13*un13*k3 - v2*k4) - V13);
      V14= V14+omega*(k0*D*t1*(k1 + un14*k2 + un14*un14*k3 - v2*k4) - V14);
      V15= V15+omega*(k0*D*t1*(k1 + un15*k2 + un15*un15*k3 - v2*k4) - V15);
      V16= V16+omega*(k0*D*t1*(k1 + un16*k2 + un16*un16*k3 - v2*k4) - V16);
      V17= V17+omega*(k0*D*t1*(k1 + un17*k2 + un17*un17*k3 - v2*k4) - V17);
      V18= V18+omega*(k0*D*t1*(k1 + un18*k2 + un18*un18*k3 - v2*k4) - V18);


      if(blockIdx.x+yo == 0){
            // akselerasjonsgrensebetingelse på innstrømskant
            real w1 = t1*accel;
            real w2 = t2*accel;
            if(V4 > w1 && V8 > w2 && V9 > w2 && V18 > w2 && V17 > w2){
                  V4-=w1;
                  V3+=w1;
                  V8-=w2;
```

```
                V10+=w2;
                V9-=w2;
                V7+=w2;
                V18-=w2;
                V16+=w2;
                V17-=w2;
                V15+=w2;
          }
      }

      V[i+ 0*NNN] = V0;
      V[i+ 1*NNN] = V1;
      V[i+ 2*NNN] = V2;
      V[i+ 3*NNN] = V3;
      V[i+ 4*NNN] = V4;
      V[i+ 5*NNN] = V5;
      V[i+ 6*NNN] = V6;
      V[i+ 7*NNN] = V7;
      V[i+ 8*NNN] = V8;
      V[i+ 9*NNN] = V9;
      V[i+10*NNN] = V10;
      V[i+11*NNN] = V11;
      V[i+12*NNN] = V12;
      V[i+13*NNN] = V13;
      V[i+14*NNN] = V14;
      V[i+15*NNN] = V15;
      V[i+16*NNN] = V16;
      V[i+17*NNN] = V17;
      V[i+18*NNN] = V18;
      }
}

__global__ void propagate(real* V,volatile int NNN) {
      int i = threadIdx.x;
      int j = blockIdx.x;
      int k = blockIdx.y;
      int I  = i;
      int Ip = (i+1)*(i != lx-1);          // Ip = (i+1)%lx
      int Im = (i-1)+(i == 0)*lx;          // Im = (i+lx-1)%lx
      int J  = j*lx;
      int Jp = ((j+1)*(j != ly-1))*lx;     // Jp = ((j+1)%ly)*lx;
      int K  = k*lx*ly;
      int Kp = ((k+1)*(k != lz-1))*lx*ly; // Kp = ((k+1)%lz)*ly*lx;

//    ### Delt cacheminne for å bedre minnecoalescing ###
      __shared__ real swapTableA[lx];
      __shared__ real swapTableB[lx];

      __shared__ real swapTableC[lx];
      __shared__ real swapTableD[lx];
//    ### /Delt cacheminne for å bedre minnecoalescing ###

      swapTableA[Ip] = V[I + J + K + 15*NNN];
      swapTableB[Im] = V[I + Jp+ K + 17*NNN];

      swapTableC[Ip] = V[I + Jp+ K + 18*NNN];
      swapTableD[Im] = V[I + J + K + 16*NNN];

      __syncthreads(); // synkroniseringsbarriere for å sikre at alle
                  // trådene i blokken er ferdige med det delte cacheminnet
```

```
      V[I + J + K +15*NNN] = swapTableB[I];
      V[I + Jp+ K +17*NNN] = swapTableA[I];

      V[I + Jp+ K +18*NNN] = swapTableD[I];
      V[I + J + K +16*NNN] = swapTableC[I];

      swapTableC[I] = V[I + J + K +  1*NNN];
      swapTableD[I] = V[I + J + K +  2*NNN];

      swapTableB[I] = V[I + J + K + 11*NNN];
      swapTableA[I] = V[I + J + Kp+ 13*NNN];

      __syncthreads(); // synkroniseringsbarriere for å sikre at alle
                  // trådene i blokken er ferdige med det delte cacheminnet

      V[I + J + K + 1*NNN] = swapTableD[Ip];
      V[I + J + K + 2*NNN] = swapTableC[Im];

      V[I + J + K +11*NNN] = swapTableA[Ip];
      V[I + J + Kp+13*NNN] = swapTableB[Im];

      swapTableC[Im] = V[I + J + Kp+ 14*NNN];
      swapTableD[Ip] = V[I + J + K + 12*NNN];

      __syncthreads(); // synkroniseringsbarriere for å sikre at alle
                  // trådene i blokken er ferdige med det delte cacheminnet

      V[I + J + Kp+14*NNN] = swapTableD[I];
      V[I + J + K +12*NNN] = swapTableC[I];

      real V3 = V[I + J + K +  3*NNN];
      real V4 = V[I + Jp+ K +  4*NNN];

      real V5 = V[I + J + K +  5*NNN];
      real V6 = V[I + J + Kp+  6*NNN];

      real V7 = V[I + J + K +  7*NNN];
      real V9 = V[I + Jp+ Kp+  9*NNN];

      real V8  = V[I + Jp+ K +  8*NNN];
      real V10 = V[I + J + Kp+  10*NNN];

      V[I + J + K + 3*NNN] = V4;
      V[I + Jp+ K + 4*NNN] = V3;

      V[I + J + K + 5*NNN] = V6;
      V[I + J + Kp+ 6*NNN] = V5;

      V[I + J + K + 7*NNN] = V9;
      V[I + Jp+ Kp+ 9*NNN] = V7;

      V[I + Jp+ K + 8*NNN] = V10;
      V[I + J + Kp+10*NNN] = V8;
}



__global__ void initialize(real* lattice, bool* obstacle, int xo, int yo,
int zo, int gLX, int gLY, int gLZ){
int NNN = lx*ly*lz;
```

```
int i = threadIdx.x + blockIdx.x*lx + blockIdx.y*lx*ly;
int x = threadIdx.x+xo; int y = blockIdx.x+yo; int z = blockIdx.y+zo;
real ry = (1.0*y)/(1.0*gLZ);
real rz = (1.0*z)/(1.0*gLZ);

obstacle[i] = pow(pow(rz-0.5,2.0) + pow(ry-0.5,2.0),0.5) < 0.05;

real t0 = 1.0/3.0;
real t1 = 1.0/18.0;
real t2 = 1.0/36.0;

lattice[i+ 0*NNN] = t0;
lattice[i+ 1*NNN] = t1;
lattice[i+ 2*NNN] = t1;
lattice[i+ 3*NNN] = t1;
lattice[i+ 4*NNN] = t1;
lattice[i+ 5*NNN] = t1;
lattice[i+ 6*NNN] = t1;
lattice[i+ 7*NNN] = t2;
lattice[i+ 8*NNN] = t2;
lattice[i+ 9*NNN] = t2;
lattice[i+10*NNN] = t2;
lattice[i+11*NNN] = t2;
lattice[i+12*NNN] = t2;
lattice[i+13*NNN] = t2;
lattice[i+14*NNN] = t2;
lattice[i+15*NNN] = t2;
lattice[i+16*NNN] = t2;
lattice[i+17*NNN] = t2;
lattice[i+18*NNN] = t2;

// randomize fluidfield
if(!(i%23)){
      lattice[i + i%19*NNN] = lattice[i + i%19*NNN]*0.98;
}
}

__global__ void unpackForSwap56(real* lattice, real* swapTab) {
// skriver overflatedata fra nabopartisjoner i retning #5 og #6 fra swapTab
til lattice
      int i = threadIdx.x;
      int j = blockIdx.x;
      int NN = lx*ly;
      int NNN = lx*ly*lz;
      int Is5 = i+lx*j + 5*lx*lz;
      int Is6 = i+lx*j + 5*lx*ly + 10*lx*lz;
      int Il5 = i+(ly-1-j)*lx+(lz-1)*lx*ly;
      int Il6 = i+j*lx;

      lattice[Il5+ 7*NNN] = swapTab[Is5       ];
      lattice[Il5+ 5*NNN] = swapTab[Is5 +   NN];
      lattice[Il5+11*NNN] = swapTab[Is5 + 2*NN];
      lattice[Il5+12*NNN] = swapTab[Is5 + 3*NN];
      lattice[Il5+ 8*NNN] = swapTab[Is5 + 4*NN];
      lattice[Il6+ 9*NNN] = swapTab[Is6       ];
      lattice[Il6+ 6*NNN] = swapTab[Is6 +   NN];
      lattice[Il6+13*NNN] = swapTab[Is6 + 2*NN];
      lattice[Il6+14*NNN] = swapTab[Is6 + 3*NN];
      lattice[Il6+10*NNN] = swapTab[Is6 + 4*NN];
}
```

```
__global__ void unpackForSwap34(real* lattice, real* swapTab) {
// skriver overflatedata fra nabopartisjoner i retning #3 og #4 fra swapTab
til lattice
      int i = threadIdx.x;
      int j = blockIdx.x;
      int NN = lx*lz;
      int NNN = lx*ly*lz;
      int Is3 = i+lx*j;
      int Is4 = i+lx*j+5*lx*(ly+lz);
      int Il3 = i+(ly-1)*lx+j*lx*ly;
      int Il4 = i+         +(lz-1-j)*lx*ly;

      lattice[Il3+10*NNN] = swapTab[Is3       ];
      lattice[Il3+ 3*NNN] = swapTab[Is3 +   NN];
      lattice[Il3+15*NNN] = swapTab[Is3 + 2*NN];
      lattice[Il3+16*NNN] = swapTab[Is3 + 3*NN];
      lattice[Il3+ 7*NNN] = swapTab[Is3 + 4*NN];
      lattice[Il4+ 8*NNN] = swapTab[Is4       ];
      lattice[Il4+17*NNN] = swapTab[Is4 +   NN];
      lattice[Il4+18*NNN] = swapTab[Is4 + 2*NN];
      lattice[Il4+ 4*NNN] = swapTab[Is4 + 3*NN];
      lattice[Il4+ 9*NNN] = swapTab[Is4 + 4*NN];
}

__global__ void packForSwap56(real* lattice, real* swapTab) {
// skriver overflatedata fra nabopartisjoner i retning #5 og #6 fra lattice
til swapTab
      int i = threadIdx.x;
      int j = blockIdx.x;
      int NN = lx*ly;
      int NNN = lx*ly*lz;
      int Is5 = i+lx*j + 5*lx*lz;
      int Is6 = i+lx*j + 5*lx*ly + 10*lx*lz;
      int Il5 = i+(ly-1-j)*lx+(lz-1)*lx*ly;
      int Il6 = i+j*lx;

      swapTab[Is5       ] = lattice[Il5+ 7*NNN];
      swapTab[Is5 +   NN] = lattice[Il5+ 5*NNN];
      swapTab[Is5 + 2*NN] = lattice[Il5+11*NNN];
      swapTab[Is5 + 3*NN] = lattice[Il5+12*NNN];
      swapTab[Is5 + 4*NN] = lattice[Il5+ 8*NNN];
      swapTab[Is6       ] = lattice[Il6+ 9*NNN];
      swapTab[Is6 +   NN] = lattice[Il6+ 6*NNN];
      swapTab[Is6 + 2*NN] = lattice[Il6+13*NNN];
      swapTab[Is6 + 3*NN] = lattice[Il6+14*NNN];
      swapTab[Is6 + 4*NN] = lattice[Il6+10*NNN];
}

__global__ void packForSwap34(real* lattice, real* swapTab) {
// skriver overflatedata fra nabopartisjoner i retning #3 og #4 fra lattice
til swapTab
      int i = threadIdx.x;
      int j = blockIdx.x;
      int NN = lx*lz;
      int NNN = lx*ly*lz;
      int Is3 = i+lx*j;
      int Is4 = i+lx*j+5*lx*(ly+lz);
      int Il3 = i+(ly-1)*lx+j*lx*ly;
      int Il4 = i+         +(lz-1-j)*lx*ly;

      swapTab[Is3       ] = lattice[Il3+10*NNN];
```

```
        swapTab[Is3 +   NN] = lattice[Il3+ 3*NNN];
        swapTab[Is3 + 2*NN] = lattice[Il3+15*NNN];
        swapTab[Is3 + 3*NN] = lattice[Il3+16*NNN];
        swapTab[Is3 + 4*NN] = lattice[Il3+ 7*NNN];
        swapTab[Is4       ] = lattice[Il4+ 8*NNN];
        swapTab[Is4 +   NN] = lattice[Il4+17*NNN];
        swapTab[Is4 + 2*NN] = lattice[Il4+18*NNN];
        swapTab[Is4 + 3*NN] = lattice[Il4+ 4*NNN];
        swapTab[Is4 + 4*NN] = lattice[Il4+ 9*NNN];
}


int main(int argc, char* argv[]){
// #####  Brukerdefinerte variable  #####
int nGPU = 4;      // antall GPUer. Bør, men trenger ikke å samsvare med
                   // fysisk antall GPUer på systemet

int nIterations = 10000;// antall iterasjoner

int topology[nGPU][11]; // topology bestemmer koblingen mellom
                        // latticepartisjoner. topology[i][j] er
                        // naboen til partisjon #i, i retning #j
                        // der retningsnummer er basert på
                        // latticenummereringen
topology[0][3] = 1;
topology[0][5] = 2;
topology[1][3] = 0;
topology[1][5] = 3;
topology[2][3] = 3;
topology[2][5] = 0;
topology[3][3] = 2;
topology[3][5] = 1;
// ##### /Brukerdefinerte variable  #####


// #####  Initialize  #####
int deviceCount;
cudaGetDeviceCount(&deviceCount);

//¤¤¤¤¤ automatisk komplettering av topology, basert på brukerspesifiserte
data om retning #3 og #5 ¤¤¤¤¤
for(int i = 0;i < nGPU;i++){
        topology[topology[i][3]][4] = i;
        topology[topology[i][5]][6] = i;
}
for(int i = 0;i < nGPU;i++){
        topology[i][7] = topology[topology[i][3]][5];
        topology[i][8] = topology[topology[i][4]][5];
        topology[i][9] = topology[topology[i][4]][6];
        topology[i][10]= topology[topology[i][3]][6];
}
//¤¤¤¤¤ /automatisk komplettering av topology, basert på brukerspesifiserte
data om retning #3 og #5 ¤¤¤¤¤


//¤¤¤¤¤ beregning av global størrelse på lattice og globalt origo for hver
partisjon, basert på data fra topology ¤¤¤¤¤
int origo[nGPU][3];     // origo[i][j] er global koordinat for lokalt origo
                        // i partisjon #i, for retning %j, der j = 0,1,2
                        // svarer til x,y,z
for(int i = 1;i < nGPU;i++){
```

```
        origo[i][0] = -1;origo[i][1] = -1;origo[i][2] = -1;
}
origo[0][0] = 0;origo[0][1] = 0;origo[0][2] = 0;
int globalX = 0; // antall partisjoner i X-retning
int globalY = 0; // antall partisjoner i Y-retning
int globalZ = 0; // antall partisjoner i Z-retning

for(int i = 0;i < nGPU;i++){
        for(int n = 1;n < nGPU;n++){
                origo[i][0] = 0;
                int zn = topology[n][4];
                if(zn != n && origo[n][1] == -1 && origo[zn][1] >= 0)
{origo[n][1] = origo[zn][1]+ly;}
                if(zn != n && origo[n][2] == -1 && origo[zn][2] >= 0)
{origo[n][2] = origo[zn][2];}
                int yn = topology[n][6];
                if(yn != n && origo[n][2] == -1 && origo[yn][2] >= 0)
{origo[n][2] = origo[yn][2]+lz;}
                if(yn != n && origo[n][1] == -1 && origo[yn][1] >= 0)
{origo[n][1] = origo[yn][1];}
                if(origo[n][1]/ly > globalY){globalY = origo[n][1]/ly;}
                if(origo[n][2]/lz > globalZ){globalZ = origo[n][2]/lz;}
        }
}
globalX++;
globalY++;
globalZ++;
//¤¤¤¤¤ /beregning av global størrelse på lattice og globalt origo for hver
partisjon, basert på data fra topology ¤¤¤¤¤


real* hostSwapTab;        // lokal tabell for mellomlagring av overflatedata
                          // under bytting mellom partisjoner
real* deviceLattice[nGPU];    // minnepekertabell til latticedata per GPU
real* deviceSwapTab[nGPU];    // minnepekertabell til overflatedata per GPU
bool* deviceObstacle[nGPU];   // minnepekertabell til obstacledata per GPU
hostSwapTab = (real*)malloc(sizeof(real)*lx*10*(ly+lz)*nGPU);


// ¤¤¤¤¤ alloker minne på GPU. initialiser verdier i lattice og obstacle
¤¤¤¤¤
dim3 dimBlock(lx);
dim3 dimGrid(ly,lz);
omp_set_num_threads(nGPU);
#pragma omp parallel
{
        int ID = omp_get_thread_num();
        cudaSetDevice(ID%deviceCount);
        cudaMalloc((void**)&deviceLattice[ID] ,sizeof(real)*lx*ly*lz*19);
        cudaMalloc((void**)&deviceSwapTab[ID] ,sizeof(real)*lx*10*(ly+lz));
        cudaMalloc((void**)&deviceObstacle[ID],sizeof(bool)*lx*ly*lz);
        initialize<<<dimGrid,dimBlock>>>(deviceLattice[ID],
deviceObstacle[ID], origo[ID][0], origo[ID][1],
origo[ID][2],globalX*lx,globalY*ly,globalZ*lz);
}
// ¤¤¤¤¤ /alloker minne på GPU. initialiser verdier i lattice og obstacle
¤¤¤¤¤


int nPoints = lx*globalX*ly*globalY*lz*globalZ;
```

```
printf("virtual #Gpu: %i; hardware #Gpu: %i\n",nGPU,deviceCount);
printf("grid: %i,%i,%i;   total #latticepoints:
%i\n",lx*globalX,ly*globalY,lz*globalZ,nPoints);
printf("subgrid: %i,%i,%i;   #latticepoints pr partition:
%i\n",lx,ly,lz,lx*ly*lz);
printf("total memory consumption: %iMB;   pr partition:
%iMB\n",(int)(nPoints/1024*(4*19+4*4+1)/1024),(int)(lx*ly*lz/1024*(4*19+4*4
+1)/1024));
printf("running a total of %i iterations\n",nIterations);
// ##### /Initialize  #####


// #####  Main loop  #####
#pragma omp parallel
{
      int ID = omp_get_thread_num();
      cudaSetDevice(ID%deviceCount);
      cudaEvent_t start,stop;
      float time;
      cudaEventCreate(&start);
      cudaEventCreate(&stop);
      cudaEventRecord(start,0);
      for(int iLoop = 0;iLoop < nIterations;iLoop++){
            // ¤¤¤¤¤¤¤¤¤ pack data for swap ¤¤¤¤¤¤¤¤¤
            // overflatedata som skal byttes mellom partisjoner skrives i
            // en sammenhengende tabell
            packForSwap34<<<lz,lx>>>(deviceLattice[ID],deviceSwapTab[ID]);
            packForSwap56<<<ly,lx>>>(deviceLattice[ID],deviceSwapTab[ID]);

            // ¤¤¤¤¤¤¤¤¤ copy from device to host ¤¤¤¤¤¤¤¤¤
            // overflatedata som skal byttes mellom partisjoner kopieres
            // fra GPURAM til CPURAM

      cudaMemcpy(hostSwapTab+ID*lx*10*(ly+lz),deviceSwapTab[ID],sizeof(real
)*lx*10*(ly+lz),cudaMemcpyDeviceToHost);

            // ¤¤¤¤¤¤¤¤¤ host corner repack ¤¤¤¤¤¤¤¤¤
            // overflatedata i hjørner ompakkes for å redusere antall
            // nødvendige overføringer
            #pragma omp barrier
            int target = topology[topology[ID][7]][6];
            if(target != ID){
                  //printf("relocate7:%i -> %i\n",ID,target);
                  int sI = ID    *lx*10*(ly+lz) + 5*lx*lz-lx;
                  int rI = target*lx*10*(ly+lz) + 5*lx*lz;
                  for(int i = 0;i < lx;i++){
                        hostSwapTab[rI+i] = hostSwapTab[sI+i];
                  }
            }
            target = topology[topology[ID][8]][6];
                  //printf("relocate8:%i -> %i\n",ID,target);
            if(target != ID){
                  int sI = ID    *lx*10*(ly+lz) + 5*lx*lz + 5*lx*ly;
                  int rI = target*lx*10*(ly+lz) + 5*lx*lz + 5*lx*ly-lx;
                  for(int i = 0;i < lx;i++){
                        hostSwapTab[rI+i] = hostSwapTab[sI+i];
                  }
            }
            target = topology[topology[ID][9]][5];
                  //printf("relocate9:%i -> %i\n",ID,target);
            if(target != ID){
```

```
                int sI = ID    *lx*10*(ly+lz) + 10*lx*lz + 5*lx*ly-lx;
                int rI = target*lx*10*(ly+lz) + 10*lx*lz + 5*lx*ly;
                for(int i = 0;i < lx;i++){
                      hostSwapTab[rI+i] = hostSwapTab[sI+i];
                }
          }
          target = topology[topology[ID][10]][5];
                //printf("relocate10:%i -> %i\n",ID,target);
          if(target != ID){
                int sI = ID         *lx*10*(ly+lz);
                int rI = (target+1)*lx*10*(ly+lz)-lx;
                for(int i = 0;i < lx;i++){
                      hostSwapTab[rI+i] = hostSwapTab[sI+i];
                }
          }

          // ¤¤¤¤¤¤¤¤¤ copy from host to device ¤¤¤¤¤¤¤¤¤
          // overflatedata som skal byttes mellom partisjoner kopieres
          // fra CPURAM til GPURAM
          #pragma omp barrier

          int offset3 = 0;
          int offset4 = 5*lx*(ly+lz);
          int offset5 = 5*lx*lz;
          int offset6 = 5*lx*(2*lz+ly);


      cudaMemcpy(deviceSwapTab[ID]+offset3,hostSwapTab+topology[ID][4]*lx*1
0*(ly+lz)+offset3,sizeof(real)*5*lx*lz,cudaMemcpyHostToDevice);

      cudaMemcpy(deviceSwapTab[ID]+offset4,hostSwapTab+topology[ID][3]*lx*1
0*(ly+lz)+offset4,sizeof(real)*5*lx*lz,cudaMemcpyHostToDevice);

      cudaMemcpy(deviceSwapTab[ID]+offset5,hostSwapTab+topology[ID][6]*lx*1
0*(ly+lz)+offset5,sizeof(real)*5*lx*ly,cudaMemcpyHostToDevice);

      cudaMemcpy(deviceSwapTab[ID]+offset6,hostSwapTab+topology[ID][5]*lx*1
0*(ly+lz)+offset6,sizeof(real)*5*lx*ly,cudaMemcpyHostToDevice);


          // ¤¤¤¤¤¤¤¤¤ unpack data from swap ¤¤¤¤¤¤¤¤¤
          // overflatedata som skal byttes mellom partisjoner skrives fra
          // sammenhengende tabell, inn i latticen

      unpackForSwap34<<<lz,lx>>>(deviceLattice[ID],deviceSwapTab[ID]);

      unpackForSwap56<<<ly,lx>>>(deviceLattice[ID],deviceSwapTab[ID]);

          // ¤¤¤¤¤¤¤¤¤ propagate and relax ¤¤¤¤¤¤¤¤¤
          propagate<<<dimGrid,dimBlock>>>(deviceLattice[ID],lx*ly*lz);
          relax<<<dimGrid,dimBlock>>>(deviceLattice[ID],
deviceObstacle[ID], lx*ly*lz,origo[ID][1]);
      }
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time,start,stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
printf("total runtime: %g s, GPU#%i\n",time/1000.0,ID);
}
// ##### /Main loop  #####
```

```
// #####  Cleanup  #####
#pragma omp parallel
{
      int ID = omp_get_thread_num();
      cudaSetDevice(ID%deviceCount);
      cudaFree(deviceLattice[ID]);
      cudaFree(deviceSwapTab[ID]);
      cudaFree(deviceObstacle[ID]);
}

return 0;
// ##### /Cleanup  #####
}
```

Appendiks D - Visualisering Kildekode

```
// Visualiseringsprogrammet er utviklet uten noen klar plan fra begynnelsen
// og funksjonalitet er lagt til, og tatt ut etter som det har vært
// nødvendig. Resultatet er blitt det som på fagspråket kalles
// "spaghettikode". Den er lagt ved hovedsaklig som en dokumentasjon på at
// jobben er gjort, og er følgelig ikke forsøkt forklart inngående. For de
// som er interessert i det som omhandler 3D stereografi skal det være
// tilstrekkelig å se på funksjonen "draw"


#include <GL/glut.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <omp.h>

#define degTOrad 0.0174532925199433
#define real float
#define bool int

static real* dataTab;
static real* velX;
static real* velY;
static real* velZ;
static real* pressure;
static int* obstacle;
static int obstacleDisplayList;
static int displayListSec;
static int displayList;
char filename[50];
static int nGPU;
static int mouseX = 0;
static int mouseY = 0;
static int mouseButtonState = 7;
double cameraPitch = 0;
double cameraHeading = 0;
const double screenDistance = 50.0;
const double eyeSpacing = 0.0;

static int visualCase = 6;
static int nVisualCases = 7;
static real maximum;
static real minimum;
static real average;
static real isoValue;
static real AverageIncomingVel = 0;
static real polySize = 25.0;
static int timestep;
static int xGrid;
static int yGrid;
static int zGrid;
static int partitionX;
static int partitionY;
static int partitionZ;
static int globalX;
static int globalY;
static int globalZ;
static int maxGrid;
```

```
static GLhandleARB sphereShader;
static int Height = 600;
static int Width = 600;
const double pixelsize = 0.028216;
static double near = 20;
static double far = 100;
static double angle = -30.0;

static int readTimestepdata(int t);

static double AA[] = {        -.0, .0,
                         .5, .5,
                         .8,-.1,
                        -.8,-.1,
                        -.8, .3, // 5
                         .2,-.4,
                        -.9,-.1,
                         .0, .9,
                        -.5, .7,
                         .9,-.0, // 10
                        -.0, .0,
                         .9, .9,
                        -.4, .6,
                         .3,-.7
              };



void renderBitmapString(float x, float y, float z, char *string) {
      char *c;
      glRasterPos3f(x, y,z);

      for (c=string; *c != '\0'; c++) {
            glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18 , *c);
      }
}

void drawCage(){
            glLineWidth(2);

            glBegin(GL_LINES);
            double BX = polySize*globalX*0.5/maxGrid;
            double BY = polySize*globalY*0.5/maxGrid;
            double BZ = polySize*globalZ*0.5/maxGrid;
            glVertex3f(-BX,-BY,-BZ);
            glVertex3f( BX,-BY,-BZ);
            glVertex3f(-BX,-BY,-BZ);
            glVertex3f(-BX, BY,-BZ);
            glVertex3f(-BX,-BY,-BZ);
            glVertex3f(-BX,-BY, BZ);
            glEnd();
            glColor3f(0.0,0.0,0.0);
            renderBitmapString( BX,-BY,-BZ, "X");
            renderBitmapString(-BX, BY,-BZ, "Y");
            renderBitmapString(-BX,-BY, BZ, "Z");
}
```

```
// Tegner
static void draw(){
glUseProgramObjectARB(sphereShader);
glEnable(GL_POINT_SPRITE);
glTexEnvf(GL_POINT_SPRITE, GL_COORD_REPLACE, GL_TRUE);
glEnable(GL_VERTEX_PROGRAM_POINT_SIZE_NV);
glEnable(GL_VERTEX_PROGRAM_POINT_SIZE_ARB);


glClearColor(0.0,0.0,0.0,1);
glClear(GL_ACCUM_BUFFER_BIT);
glMatrixMode (GL_PROJECTION);
glLoadIdentity();
gluPerspective(2*atan(Height*pixelsize/(2*50))/degTOrad,1.0*Width/Height,ne
ar,far);

glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
glEnable(GL_DEPTH_TEST);
int aa;
int AAq = 14-13*(mouseButtonState == 6);
for(aa = 0;aa < AAq;aa++){
      glPushMatrix();
            glClearColor (0.5, 0.0, 0.0, 1.0);
            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
            glTranslated(eyeSpacing,0,0);
            glRotated(atan(eyeSpacing/screenDistance)/degTOrad,0,1,0);
            glTranslated(0,0,-screenDistance);
            glRotated(cameraHeading,0,1,0);
      glRotated(cameraPitch,cos(cameraHeading*degTOrad),0,sin(cameraHeading
*degTOrad));
            glPointSize(10.0);
            glTranslated(AA[2*aa]*pixelsize,AA[2*aa+1]*pixelsize,0);
            glUseProgramObjectARB(sphereShader);
            glColor3f(0.7,0.0,0.0);
            glCallList(obstacleDisplayList);
            glColor3f(1.0,0.0,0.0);
            glCallList(displayList);
            glUseProgramObjectARB(0);
            drawCage();
            glAccum(GL_ACCUM,1.0/AAq);
      glPopMatrix();
      glPushMatrix();
            glClearColor (0.0, 0.5, 0.5, 1.0);
            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
            glTranslated(-eyeSpacing,0,0);
            glRotated(-atan(eyeSpacing/screenDistance)/degTOrad,0,1,0);
            glTranslated(0,0,-screenDistance);
            glRotated(cameraHeading,0,1,0);
      glRotated(cameraPitch,cos(cameraHeading*degTOrad),0,sin(cameraHeading
*degTOrad));
            glPointSize(10.0);
            glTranslated(AA[2*aa]*pixelsize,AA[2*aa+1]*pixelsize,0);
            glUseProgramObjectARB(sphereShader);
            glColor3f(0.0,0.7,0.7);
            glCallList(obstacleDisplayList);
            glColor3f(0.0,1.0,1.0);
            glCallList(displayList);
            glUseProgramObjectARB(0);
            drawCage();
            glAccum(GL_ACCUM,1.0/AAq);
```

```
        glPopMatrix();
}
glAccum(GL_RETURN,1.0);
glUseProgramObjectARB(0);
glMatrixMode (GL_PROJECTION);
glLoadIdentity();
glOrtho(0,Width,0,Height,-1,1);
glDisable(GL_DEPTH_TEST);
glColor3f(0,0,0);

char text[50];
sprintf(text,"isovalue : %g < %g < %g",minimum,isoValue,maximum);
renderBitmapString(2, Height-20, 0, text);
sprintf(text,"timestep = %i",timestep);
renderBitmapString(2, Height-40, 0, text);

switch (visualCase){
        case 0:
                sprintf(text,"plottype: iso- pressure");break;
        case 1:
                sprintf(text,"plottype: iso- absolute velocity");break;
        case 2:
                sprintf(text,"plottype: iso- X velocity");break;
        case 3:
                sprintf(text,"plottype: iso- Y velocity");break;
        case 4:
                sprintf(text,"plottype: iso- Z velocity");break;
        case 5:
                sprintf(text,"plottype: iso- scalar vorticity");break;
        case 6:
                sprintf(text,"plottype: streamlines");break;
}
renderBitmapString(2, Height-60, 0, text);
sprintf(text,"avg. incoming velocity = %g",AverageIncomingVel);
renderBitmapString(2, Height-80, 0, text);
glutSwapBuffers();
}



static void reshape(GLsizei w, GLsizei h){
glViewport(0,0,w,h);
Width = w;
Height = h;
}

static void setupPixelShader(){
GLhandleARB pixShader = glCreateShaderObjectARB(GL_FRAGMENT_SHADER);
int m = 10;
const GLcharARB *code[m];
int length[m];
int l = 0;
code[l++] = "void main(){";
code[l++] = "const vec3 lightDir = vec3(0.577, 0.577, 0.577);";
code[l++] = "vec3 N;";
code[l++] = "N.xy = gl_TexCoord[0].xy*vec2(2.0, -2.0) + vec2(-1.0, 1.0);";
code[l++] = "float mag = dot(N.xy, N.xy);";
code[l++] = "if (mag > 1.0) discard;";
code[l++] = "N.z = sqrt(1.0-mag);";
code[l++] = "float diffuse = max(0.0, dot(lightDir, N));";
code[l++] = "gl_FragDepth = gl_FragCoord.z-N.z/80;";
```

```
code[l++] = "gl_FragColor = gl_Color * diffuse;}";
int i;
for(i = 0; i < m;i++){
length[i] = strlen(code[i]);
}
glShaderSourceARB(pixShader,m,code,length);
glCompileShaderARB(pixShader);

GLhandleARB verShader = glCreateShaderObjectARB(GL_VERTEX_SHADER);
m = 7;
const GLcharARB *codeV[m];
int lengthV[m];
l = 0;
codeV[l++] = "void main(){";
codeV[l++] = "vec3 posEye = vec3(gl_ModelViewMatrix * vec4(gl_Vertex.xyz,
1.0));";
codeV[l++] = "float dist = length(posEye);";
codeV[l++] = "gl_PointSize = 200/dist;";
codeV[l++] = "gl_TexCoord[0] = gl_MultiTexCoord0;";
codeV[l++] = "gl_Position = gl_ModelViewProjectionMatrix *
vec4(gl_Vertex.xyz, 1.0);";
codeV[l++] = "gl_FrontColor = gl_Color;}";
for(i = 0; i < m;i++){
lengthV[i] = strlen(codeV[i]);
}

glShaderSourceARB(verShader,m,codeV,lengthV);
glCompileShaderARB(verShader);

sphereShader = glCreateProgramObjectARB();
glAttachObjectARB(sphereShader,verShader);
glAttachObjectARB(sphereShader,pixShader);
glLinkProgram(sphereShader);

GLint status;
glGetProgramiv(sphereShader,GL_LINK_STATUS,&status);

//printf("shader link status: %i\n",status);
}

real* getStreamLines(int *nCoords){
      int nSteps = 5*globalY/6;
      int nLines = 20;
      double h = 4000.0;
      nCoords[0] = nSteps*nLines;
      real* streamLines = (real*)malloc(nCoords[0]*sizeof(real)*3);
      int i;
      for(i = 0;i < nLines;i++){
            streamLines[3*i+0] = globalX/2;
            streamLines[3*i+1] = 1;
            streamLines[3*i+2] = globalZ/3+(globalZ/(nLines*3.0)*i);
      }
      for(i = 1;i < nSteps;i++){
            int j;
            for(j = 0;j < nLines;j++){
                  real x = streamLines[nLines*3*(i-1)+j*3+0];
                  real y = streamLines[nLines*3*(i-1)+j*3+1];
                  real z = streamLines[nLines*3*(i-1)+j*3+2];
                  int I = ((int)x) + ((int)y)*globalX +
((int)z)*globalX*globalY;
```

```
				double tempV =
sqrt(velX[I]*velX[I]+velY[I]*velY[I]+velZ[I]*velZ[I]);
				h = 1.0/tempV;
				if(tempV == 0){h = 0;}
				streamLines[nLines*3*i+j*3+0] = x + h*velX[I];
				streamLines[nLines*3*i+j*3+1] = y + h*velY[I];
				streamLines[nLines*3*i+j*3+2] = z + h*velZ[I];
			}
		}
		return streamLines;
}


real* getIsoCoordinates(int* nCoords,real c){
real delta = (maximum-minimum)/100.0;
omp_set_num_threads(4);
nCoords[0] = 0;
int counter[4];
#pragma omp parallel
{
		int ID = omp_get_thread_num();
		counter[ID] = 0;
		int i;
		for(i = ID;i < globalX*globalY*globalZ;i+=4){
			int x = i%globalX;
			int y = (i/globalX)%globalY;
			int z = i/(globalX*globalY);
			int I = x;
			int Ip= (x+1)%globalX;
			int J = y*globalX;
			int Jp= ((y+1)%globalY)*globalX;
			int K = z*globalX*globalY;
			int Kp= ((z+1)%globalZ)*globalX*globalY;

			int local = 0x7f*(dataTab[i]>c);
			int neighbors = (dataTab[I +J +Kp]>c)<<0 |
					(dataTab[I +Jp+K ]>c)<<1 |
					(dataTab[I +Jp+Kp]>c)<<2 |
					(dataTab[Ip+J +K ]>c)<<3 |
					(dataTab[Ip+J +Kp]>c)<<4 |
					(dataTab[Ip+Jp+K ]>c)<<5 |
					(dataTab[Ip+Jp+Kp]>c)<<6;
			counter[ID] += (local ^ neighbors) != 0;
		}
}

nCoords[0] = counter[0]+counter[1]+counter[2]+counter[3];
real* isoCoordinates = (real*)malloc(nCoords[0]*sizeof(real)*3);
counter[3] = counter[0]+counter[1]+counter[2];
counter[2] = counter[0]+counter[1];
counter[1] = counter[0];
counter[0] = 0;

#pragma omp parallel
{
		int ID = omp_get_thread_num();
		int secondaryCounter = 0;
		int i;
		for(i = ID;i < globalX*globalY*globalZ;i+=4){
			int x = i%globalX;
			int y = (i/globalX)%globalY;
```

```
            int z = i/(globalX*globalY);
            int I = x;
            int Ip= (x+1)%globalX;
            int J = y*globalX;
            int Jp= ((y+1)%globalY)*globalX;
            int K = z*globalX*globalY;
            int Kp= ((z+1)%globalZ)*globalX*globalY;

            int local = 0x7f*(dataTab[i]>c);
            int neighbors = (dataTab[I +J +Kp]>c)<<0 |
                        (dataTab[I +Jp+K ]>c)<<1 |
                        (dataTab[I +Jp+Kp]>c)<<2 |
                        (dataTab[Ip+J +K ]>c)<<3 |
                        (dataTab[Ip+J +Kp]>c)<<4 |
                        (dataTab[Ip+Jp+K ]>c)<<5 |
                        (dataTab[Ip+Jp+Kp]>c)<<6;
            if((local ^ neighbors) != 0){
                  isoCoordinates[(counter[ID] + secondaryCounter)*3+0] =
i%globalX;
                  isoCoordinates[(counter[ID] + secondaryCounter)*3+1] =
(i/globalX)%globalY;
                  isoCoordinates[(counter[ID] + secondaryCounter)*3+2] =
i/(globalX*globalY);
                  secondaryCounter++;
            }
      }
}
      return isoCoordinates;
}


static void makeIsosurf(int type, int time){
if(((type%nVisualCases != visualCase) && type != 6) || time != timestep){
      minimum = pow(10.0,10.0);
      maximum = -minimum;
      visualCase = type%nVisualCases;
      timestep = time;
      int x,y,z;
      int counter = 0;
      average = 0;
      for(z = 0;z < globalZ;z++){
            for(y = 0;y < globalY;y++){
                  for(x = 0;x < globalX;x++){
                        int I = x+y*globalX+z*globalY*globalX;
                        double value = 0;
                        double vx = velX[I];
                        double vy = velY[I];
                        double vz = velZ[I];
                        double p = pressure[I];
                        int Ixp,Iyp,Izp;
                        switch (visualCase){
                              case 0:
                              value = pressure[I];
                              break;
                              case 1:
                              value =
sqrt(velX[I]*velX[I]+velY[I]*velY[I]+velZ[I]*velZ[I]);
                              break;
                              case 2:
                              value = velX[I];
                              break;
```

```
                                        case 3:
                                        value = velY[I];
                                        break;
                                        case 4:
                                        value = velZ[I];
                                        break;
                                        case 5:
                                        Ixp = ((x+1)%globalX)+(y  )         *globalX+
(z  )         *globalY*globalX;
                                        Iyp =  (x  )+        ((y+1)%globalY)*globalX+
(z  )         *globalY*globalX;
                                        Izp =  (x  )+         (y  )
*globalX+((z+1)%globalZ)*globalY*globalX;
                                        double dVZdy = velZ[Iyp] - vz;
                                        double dVYdz = velY[Izp] - vy;
                                        double dVXdz = velX[Izp] - vx;
                                        double dVZdx = velZ[Ixp] - vz;
                                        double dVYdx = velY[Ixp] - vy;
                                        double dVXdy = velX[Iyp] - vx;
                                        value = sqrt(pow(dVZdy - dVYdz,2)+pow(dVXdz -
dVZdx,2)+pow(dVYdx - dVXdy,2));
                                        break;
                                }
                                dataTab[I] = value;
                                if(!obstacle[I]){
                                        if(value < minimum){minimum = value;}
                                        if(value > maximum){maximum = value;}
                                        average+=value;
                                        counter++;
                                }
                        }
                }
        }
        average/=counter;
}


if(isoValue < minimum){
isoValue = minimum;
}else if(isoValue > maximum){
isoValue = maximum;
}
double c = isoValue;
real* isoCoordinates;
int nCoords;
if(type != 6){
isoCoordinates = getIsoCoordinates(&nCoords,c);
}else{
visualCase = 6;
isoCoordinates = getStreamLines(&nCoords);
}

glNewList(displayListSec, GL_COMPILE);
glBegin(GL_POINTS);
int i;
for(i = 0;i < nCoords;i++){
        double cx = polySize*(isoCoordinates[3*i+0]-globalX*0.5)/maxGrid;
        double cy = polySize*(isoCoordinates[3*i+1]-globalY*0.5)/maxGrid;
        double cz = polySize*(isoCoordinates[3*i+2]-globalZ*0.5)/maxGrid;
        glVertex3f(cx,cy,cz);
}
```

```
glEnd();
glEndList();
int temp = displayList;
displayList = displayListSec;
displayListSec = temp;
}

static int readTimestepdata(int t){
int n;
int counter = 0;
AverageIncomingVel = 0;
int z,y,x;
for(n = 0;n < nGPU;n++){
	char file[50];
	sprintf(file,"%s%i-%i.out",filename,n,t);
	FILE *fp;
	fp = fopen(file, "r");
	if(fp != NULL){
		printf("%s\n",file);
		int xo; int yo; int zo;
		fscanf (fp, "%i %i %i", &xo, &yo, &zo);

		for(z = 0;z < zGrid;z++){
			for(y = 0;y < yGrid;y++){
				for(x = 0;x < xGrid;x++){
					real vx; real vy; real vz; real p;
					fscanf (fp, "%g %g %g %g", &vx, &vy, &vz, &p);

	velX[(x+xo)+(y+yo)*globalX+(z+zo)*globalY*globalX] = vx;

	velY[(x+xo)+(y+yo)*globalX+(z+zo)*globalY*globalX] = vy;

	velZ[(x+xo)+(y+yo)*globalX+(z+zo)*globalY*globalX] = vz;

	pressure[(x+xo)+(y+yo)*globalX+(z+zo)*globalY*globalX] = p;

	if(!obstacle[(x+xo)+(y+yo)*globalX+(z+zo)*globalY*globalX] && y ==
globalY/50){
					AverageIncomingVel += vy;
					counter++;
				}
				}
			}
		}
		fclose(fp);
	}else{
		readTimestepdata(0);
		return;
	}
}
AverageIncomingVel /= counter;
makeIsosurf(visualCase,t);
}

static void mouseMotion(int x, int y){
	switch (mouseButtonState){
		case 6:
			cameraHeading += (x-mouseX)*0.2;
			cameraPitch += (y-mouseY)*0.2;
			break;
	}
```

```
        mouseX = x; mouseY = y;
}

void keyboard (unsigned char key, int x, int y){
        makeIsosurf(visualCase+1,timestep);
}


static void mouseButton(int button, int state, int x, int y){
        mouseX = x; mouseY = y;
        mouseButtonState = (mouseButtonState & ~(1<<button))|(state<<button);
        if(mouseButtonState == 5) {readTimestepdata((timestep+1));}
        if(mouseButtonState == 3) {isoValue = y/(1.0*Height)*(maximum-
minimum) + minimum; makeIsosurf(visualCase,timestep);}
}

int main(int argc, char *argv[]){
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH | GLUT_ACCUM);
glutCreateWindow("VBO");
glutInitWindowSize(600,600);
glutDisplayFunc(draw);
glutIdleFunc(draw);
glutReshapeFunc(reshape);
glutMouseFunc(mouseButton);
glutMotionFunc(mouseMotion);
glutKeyboardFunc(keyboard);
setupPixelShader();
FILE *fp;
        fp = fopen("init.out", "r");
        if(fp != NULL){

                fscanf(fp, "%s", filename);
                fscanf(fp, "%i %i %i", &partitionX, &partitionY, &partitionZ);
                fscanf(fp, "%i %i %i", &xGrid, &yGrid, &zGrid);
                fscanf(fp, "%i",&nGPU);
                fclose(fp);
                globalX = partitionX*xGrid;
                globalY = partitionY*yGrid;
                globalZ = partitionZ*zGrid;
                displayListSec = glGenLists(1);
                displayList = glGenLists(1);
                obstacleDisplayList = glGenLists(1);
                glNewList(obstacleDisplayList, GL_COMPILE);
                fp = fopen("obstacle.out", "r");
                int nPoints = globalX*globalY*globalZ;
                obstacle = (bool*)malloc(sizeof(bool)*nPoints);
                int i;
                for(i = 0;i < nPoints;i++){
                        obstacle[i] = 0;
                }
                maxGrid = globalX;
                if(globalY > maxGrid) {maxGrid = globalY;}
                if(globalZ > maxGrid) {maxGrid = globalZ;}
                int x; int y; int z;
                int end = fscanf (fp, "%i %i %i", &x, &y, &z);
                glBegin(GL_POINTS);
                while(end > 0){
                        double cx = polySize*(x-globalX*0.5)/maxGrid;
                        double cy = polySize*(y-globalY*0.5)/maxGrid;
                        double cz = polySize*(z-globalZ*0.5)/maxGrid;
```

```
                glVertex3f(cx,cy,cz);
                obstacle[x + y*globalX + z*globalX*globalY] = 1;
                end = fscanf (fp, "%i %i %i", &x, &y, &z);
            }
            glEnd();

            fclose(fp);

            glEndList();

            dataTab = (real*)malloc(xGrid*yGrid*zGrid*sizeof(real)*nGPU);
            velX = (real*)malloc(xGrid*yGrid*zGrid*sizeof(real)*nGPU);
            velY = (real*)malloc(xGrid*yGrid*zGrid*sizeof(real)*nGPU);
            velZ = (real*)malloc(xGrid*yGrid*zGrid*sizeof(real)*nGPU);
            pressure = (real*)malloc(xGrid*yGrid*zGrid*sizeof(real)*nGPU);
            readTimestepdata(4);
            glutMainLoop();
            return 1;
        }else{
            return 0;
        }
    glutMainLoop();
    }
```