UNIVERSITETET I OSLO
Institutt for informatikk

Forsøk med parallell sortering på flerkjerne-CPU og GPU

Masteroppgave

Cato Morholt

26. mai 2010

Forord

Etter mange år på IFI er jeg nå snart ved veis ende for mine studier ved Universitetet i Oslo. Det har vært en lang og spennende vei, og jeg håper vi kanskje ses igjen.

Først og fremst vil jeg takke min veileder Arne Maus som har bidratt med mye av sin tid og kunnskap gjennom arbeidet med denne masteroppgaven. Arne har vist stort engasjement, og på mesterlig vis klart å motivere meg når jeg selv har "stått helt fast".

Takk til min gode venn Audun for uvurderlig innsikt i CUDA og GPU-programmering.

Takk til Dag Langmyhr for råd og veiledning med LaTeX og til Knut Hegna for hjelp med å finne litteratur og kontrollere referanser.

En spesielt stor takk går til mine gode venner Kent William og Kristoffer for verdifulle tilbakemeldinger på denne oppgaven.

Til slutt ønsker jeg å takke Maria, Ingrid, Anne og alle andre som har hjulpet, støttet og motivert meg underveis.

Cato Morholt
Mai 2010

Innhold

Oversettelser og forkortelser

Oversettelser

Norske oversettelser benyttet i denne oppgaven:

Bitonisk	Bitonic
Delt minne	Shared memory
Enhet	Device
Flerkjerne	Multicore
Forgrening	Branch
Grensesnitt	Interface
Høyoppløselig	High resolution
Koalesert	Coalesced
Medprosessor	Coprocessor
Odde-par-flettesortering	Odd-even mergesort
Permutasjonssykel	Permutation cycle
Rekursjonstre	Recursion tree
Rutenett	Grid
Sammenlikningselement	Comparison element
Skjev	Unaligned
Sorteringsnettverk	Sorting network
Stakk	Stack
Stykke	Slice
Tremaskin	Tree machine
Trådsamling	Thread pool
Trådveksler	Thread scheduler
Varp	Warp
Vert	Host

Forkortelser

API	Application Programming Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
FIFO	First In First Out
FLOPS	Floating Point Operations Per Second
FSB	Front-Side Bus
GPGPU	General Purpose Computing on GPU

GPU	Graphics Processing Unit
I/O	Input/Output
JDK	Java Development Kit
JNI	Java Native Interface
JVM	Java Virtual Machine
MSB	Most Significant Bit
PCIe	Peripheral Component Interconnect Express
RAM	Random Access Memory
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SP	Scalar Processor
SM	Streaming Multiprocessor
STL	C++ Standard Template Library
TPC	Texture/Processor Cluster

Kapittel 1

Introduksjon

Moderne datamaskiner blir stadig mer parallelle. De fleste nye datamaskiner i dag kommer med en CPU som har to, fire eller åtte kjerner. Allerede i år er det planlagt en CPU med 12 kjerner, og neste år en med 16. Denne trenden stiller nye krav til oss som utviklere, og vi blir i økende grad nødt til å ta hensyn til flerkjerne CPU-er dersom vi ønsker god ytelse i applikasjonene våre.

Det blir også stadig mer vanlig å bruke skjermkortet som en medprosessor for beregninger i vanlige programmer. Moderne skjermkort er svært godt egnet til vektorberegninger[1], og kan utføre visse operasjoner svært mye raskere enn en CPU. Dette blir spesielt tydelig når vi ser at Apple har valgt å inkludere OpenCL, et rammeverk som gir direkte tilgang til GPU-en, som en integrert del av sitt nye operativsystem, MacOS X 10.6 (Snow Leoprad).

I denne oppgaven ønsker vi å se nærmere på parallell programmering for flerkjerne-CPU og GPU. Vi er interessert i å se hvordan man kan gå fra en tradisjonell sekvensiell algoritme over til en parallell algoritme på disse plattformene. Finnes det generelle metoder man kan benytte, eller må man finne opp hjulet på nytt med hver algoritme? Hvor vanskelig er det? Og hvor mye kan man tjene på det?

For å undersøke dette trenger vi algoritmer som er egnet for parallellisering, men heller ikke trivielt parallelliserbare. Vi endte tidlig opp med å velge sortering som utgangspunkt. Dette er et felt hvor det allerede er gjort mye forskning, men som fortsatt er spennende å utforske. Det at både veileder og kandidat har interesse for emnet spilte også en stor rolle i avgjørelsen.

I kapittel 2 vil vi se nærmere på sortering. Vi ser sortering i et historisk perspektiv, og undersøker hva som allerede finnes av forskning. Spesielt vil vi se på hvilke algoritmer som er utviklet for flerkjerne-CPU og for GPU, hvordan disse yter og hvilke lærdommer vi kan ta med oss videre.

Kapittel 3 introduserer algoritmene vi ønsker å parallellisere; flettesortering og venstre-radix-sortering. Vi går igjennom referanseimplementasjonene av disse, og ser nærmere på testmiljøet vi kommer til å benytte.

Kapittel 4 og 5 tar for seg de parallelle algoritmene vi har utviklet på henholdsvis CPU og GPU. Begge kapitlene starter med en introduksjon til parallell programmering på den

[1] Vektorberegning: Å utføre samme beregning på mange elementer i parallell

aktuelle plattformen. Deretter beskrives implementasjonen av algoritmene og resultatene vi målte underveis.

I kapittel 6 sammenligner vi resultatene fra hver algoritme og ser nærmere på hva disse betyr. Vi ser på hva som gikk bra, og hva som gikk galt underveis. Til slutt tar vi med våre ideer for videre forbedring av algoritmene.

Kapittel 7 oppsummerer vi hva vi har gjort i oppgaven, hvordan våre resultater presterer i forhold til eksisterende algoritmer og hva vi kunne gjort annerledes.

Kapittel 2

Bakgrunn

I denne oppgaven skal vi gjøre forsøk med parallelle sorteringsalgoritmer på flerkjerne-CPU og GPU. I dette kapittelet vil vi se nærmere på sortering i historisk sammenheng, og hva slags forskning som er gjort på emnet. Til slutt vil vi se på hvilken lærdom vi kan ta med oss og bruke videre i oppgaven.

2.1 Om sortering

Behovet for effektive sorteringsalgoritmer er vesentlig eldre enn oppfinnelsen av datamaskinen. Før dette sorterte man for hånd og benyttet ofte radix-lignende algoritmer. På 1880-tallet hadde man begynt å samle inn så mye informasjon at man ikke lenger kunne sortere den manuelt. Dette inspirerte H. Hollerith ved Census Bureau til å konstruerte en elektrisk maskin for å bistå med sorteringsarbeidet [Knuth, 1973, side 382]. Da den første programmerbare datamaskinen ble laget på 1940-tallet var det naturlig at sortering var ett av de første bruksområdene som ble undersøkt. Faktisk tyder mye på at det første programmet som noensinne ble skrevet for en programmerbar datamaskin, var et program for sortering [Knuth, 1973, side 384]. Innen 1960-tallet hadde man begynt å vurdere sorteringsnettverk for å kunne sortere tall raskere enn den kjente $O(n \log n)$-grensen. Dette oppnådde man ved å gjøre flere sammenlikninger i parallell. I 1968 publiserte K. Batcher sin rapport om sorteringsnettverk og deres anvendelser [Batcher, 1968], hvor han presenterer algoritmer for *Odde-par-flettesortering* (Odd-even mergesort) og *Bitonisk sortering* (Bitonic sort). Begge disse står fortsatt som to av de beste og mest elegante sorteringsnettverkene man kjenner.

Fra 1970-tallet og fram til i dag har man forsket på parallell sortering på en rekke forskjellige arkitekturer. Mye av denne forskningen har vært rettet mot superdatamaskiner. Tidlige modeller fra 1980-tallet kunne ha 4 til 16 prosessorer. Moderne superdatamaskiner består av tusenvis av datamaskiner koblet sammen med høyhastighetsnettverk i store klynger. En slik klynge kan ha flere hundre tusen kjerner. Med så mange kjerner blir effektiv kommunikasjon mellom dem viktig for å oppnå god ytelse og mange arkitekturer har blitt laget for å oppnå dette. Eksempler på slike arkitekturer er blant annet *lineære array-er*, *rutenett* og *tremaskiner*. En generell oversikt over algoritmer på slike arkitekturer finner vi i [Miller og Boxer, 2000] og [JáJá, 1992], mens [Akl, 1985] ser nærmere på sortering på disse.

Cell er en flerkjernet prosessor som opprinnelig ble designet for spillkonsoller. Den fikk

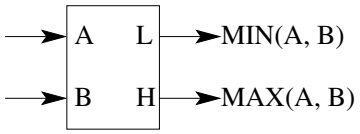
stor oppmerksomhet da Sony valgte å bruke den i sin nyeste konsoll, PlayStation 3. Men på grunn av prosessorens unike egenskaper har den også vakt stor interesse innenfor vitenskapelige miljøer. CellSort [Gedik et al., 2007] er en algoritme spesielt utviklet for Cell-prosessoren som demonstrerer at denne plattformen også kan egne seg til sortering.

I nyere tid observerer vi at også vanlige konsumentmaskiner har en stor grad av parallellitet. Vi ser derfor at det blir stadig viktigere å designe applikasjoner for parallelle arkitekturer. Spesielt flerkjerne-CPU og GPU er nå blitt en standard i alle datamaskiner, og trenden tyder på at disse kommer bare til å bli mer parallelle. Vi ønsker derfor i denne oppgaven å se nærmere på disse plattformene, og hvordan man kan utnytte potensialet i dem.

2.2 Sorteringsnettverk

Sorteringsnettverk er spesialbygde kretser beregnet på å sortere tall. De stammer fra 1950-tallet, da datamaskiner var svært dyre og hadde begrensede ressurser. I dag er ikke lengre sorteringsnettverk en praktisk måte å sortere tall, men teorien og algoritmene bak dem er fortsatt meget relevante. Som vi vil vise i kapittel 2.4.3 på side 10, benytter mange moderne algoritmer seg av disse metodene for å oppnå høyere ytelse.

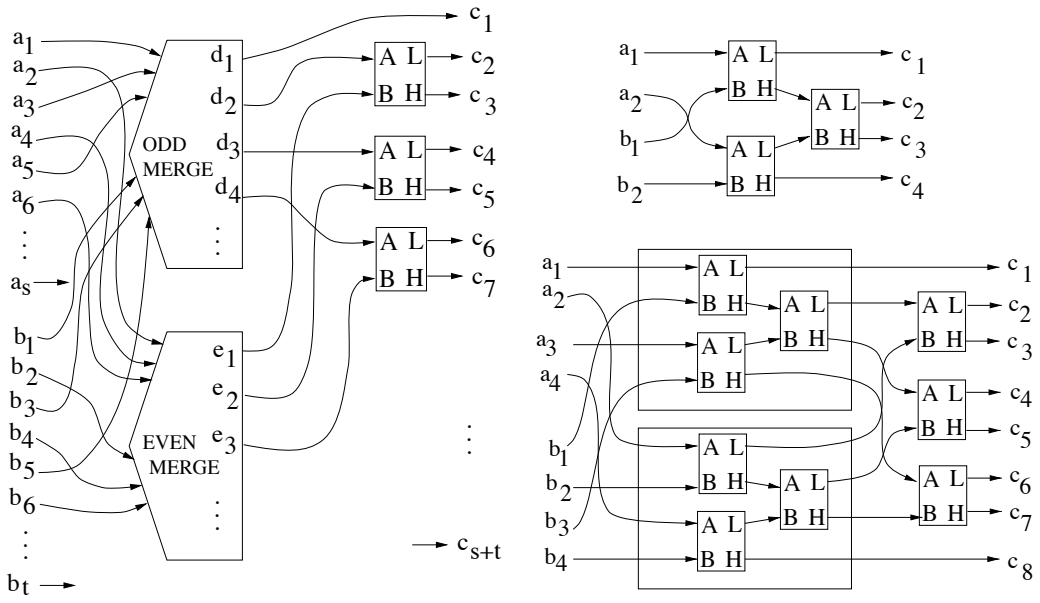
Den opprinnelige ideen bak sorteringsnettverk var å undersøke muligheten til å sortere raskere ved å sammenligne flere tall i parallell. Siden datamaskiner på den tiden var svært dyre, var det mer praktisk å bygge egne kretser spesialberegnet for sortering. Disse kretsene bestod av en rekke *sammenlikningselementer* (herifra referert til som SE) koblet sammen i et nettverk. En skjematisk oversikt av et SE er vist i figur 2.1. Vi ser at hvert SE har to innganger (A og B) og to utganger (L og H). Den ene utgangen (L) gir det laveste tallet av A og B, mens den andre utgangen (H) gir det høyeste. Ved å sette sammen tilstrekkelig mange SE-er, kan vi lage et nettverk som sorterer n tall.



Figur 2.1: Sammenlikningselement [Batcher, 1968]

Da sorteringsnettverk ble utviklet var dette en fornuftig teknisk løsning for rask sortering av data. I forhold til datidens datamaskiner var det å bygge et SE både enkelt og billig, og man hadde teknologi til å bygge et stort antall SE-er på én integrert krets [Knuth, 1973, side 222].

Den største ulempen med sorteringsnettverk er at det kreves ekstremt mange SE-er for selv å sortere små mengder tall. For eksempel vil et sorteringsnettverk for 1024 elementer krever 24 063 SE-er med odde-par-flettesortering eller 28 160 SE-er med bitonisk sortering [Batcher, 1968]. Generelt krever disse nettverkene rund $\frac{1}{4}n \log n$ SE-er for å sortere n tall [Batcher, 1968]. Dette er naturligvis ikke praktisk gjennomførbart i dag når man skal sortere millioner av tall.



(a) Iterativ regel [Batcher, 1968] (b) 2x2- og 4x4-flettekrets [Batcher, 1968]

Figur 2.2: Sammensetning av odde-par-flettenettverk

2.2.1 Odde-par-flettesortering

Odde-par-flettesortering var den første av de to algoritmene presentert i [Batcher, 1968]. Det var den første virkelig vellykkede algoritmen for et sorteringsnettverk som var både gjennomførbar og rask.

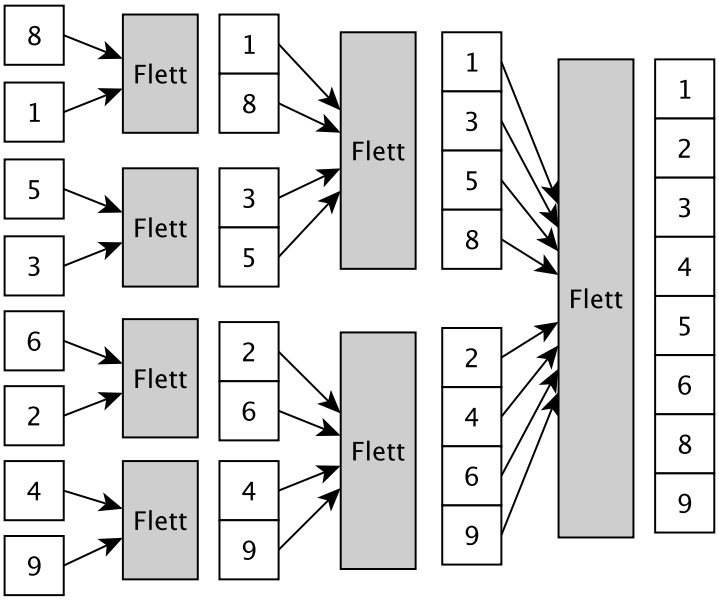
Algoritmen baserer seg på et nettverk bygd opp av flettekretser. Hver flettekrets tar to sorterte lister og fletter dem til én sortert liste. Batcher observerte at man gitt to flettekretser for n elementer, kan bruke disse til å bygge en ny flettekrets for $2n$ elementer. Figur 2.2a viser hvordan dette gjøres ved å flette elementene med oddetall-indeks og elementene med partall-indeks hver for seg (derav navnet på algoritmen).

Legg merke til at et enkelt SE kan ses på som en flettekrets for to elementer. Den tar to lister med ett element, og lager en sortert liste med to elementer. Med utgangspunkt i den iterative regelen beskrevet over, kan vi derfor bygge flettekretser av vilkårlig størrelse. Figur 2.2b viser flettekretser for henholdsvis 4 og 8 elementer.

Batcher viser også hvordan man med et nettverk av flettekretser enkelt kan sortere 2^p tall. Figur 2.3 på neste side viser et eksempel hvor man sorterer 8 tall. Her ser vi at det i første nivå sorteres to og to tall med fire enkeltstående SE-er. I andre nivå har vi nå fire sorterte lister med to tall i hver. Disse listene flettes videre med to $2x2$-flettekretser (se øverst i figur 2.2b). Til slutt har vi to sorterte lister med fire elementer som flettes til én sortert liste med én $4x4$-flettekrets (se nederst i figur 2.2b).

2.2.2 Bitonisk sortering

Bitonisk sortering var den andre algoritmen presentert i [Batcher, 1968]. Bitonisk sortering er en videreutvikling av odde-par-flettesortering, og har en mer regelmessig struktur som gir større fleksibilitet. Til gjengjeld krever den noen flere SE-er.



Figur 2.3: Sortering med odde-par-flettekretser

Byggestenen i bitoniske sorteringsnettverk, er en *bitonisk sorterer*. Dette er en krets som leser inn en bitonisk sekvens, og gir ut en sortert sekvens.

Definisjon: En bitonisk sekvens er en sekvens som består av to delsekvenser, en stigende og en synkende. En sekvens er fortsatt bitonisk om den kan roteres (deles på et vilkårlig sted, og bytte rekkefølge på delsekvensene) slik at den oppfyller det første kravet.

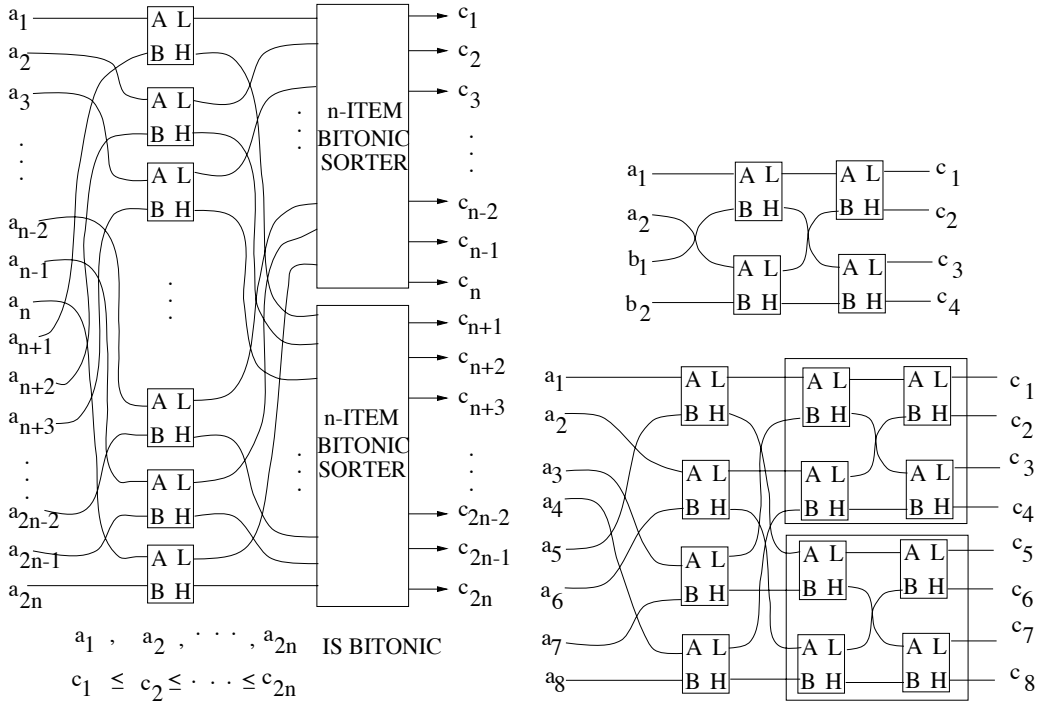
På samme måte som for odde-par-flettekretser, bygges en bitonisk sorterer for $2n$ elementer av to bitoniske sorterere for n elementer. Figur 2.4a på neste side viser hvordan dette gjøres. Legg også her merke til at et SE kan ses på som en bitonisk sorterer for 2 elementer. Figur 2.4b på neste side viser dermed hvordan vi kan bygge en bitonisk sorterer for henholdsvis 4 og 8 tall.

På samme måte som odde-par-sorteringsnettverk, kan et bitonisk sorteringsnettverk for 2^p tall bygges opp av p nivåer av mindre bitoniske sortererere. Eneste forskjell er når to sorterte sekvenser kobles til en større bitonisk sorterer, reverseres den andre listen for å lage en ny bitonisk sekvens.

2.3 Sortering på flerkjerne-CPU

2.3.1 Algoritmene

Selv om mye av forskning på parallell sortering har vært rettet mot større datamaskiner, er det også interesse for vanlig flerkjerne-CPU-er. Her tar vi for oss tre nylig publiserte artikler, og presenterer kort hvilke resultater de har kommet frem til.



(a) Iterativ regel [Batcher, 1968]

(b) Bitonisk sorterer for 4 og 8 tall [Batcher, 1968]

Figur 2.4: Sammensetning av bitoniske sorteringsnettverk

AA-Sort

Aligned-Access Sort (AA-Sort) [Inoue et al., 2007] er en algoritme spesielt designet for moderne flerkjerne-CPU-er. Algoritmen forsøker å kombinere trådparallellitet i form av flere kjerner, med dataparallellitet i form av SIMD-instruksjoner (Single Instruction Multiple Data). For å dra fullt utbytte av SIMD-parallelliteten, er det lagt stor vekt på å unngå *skjeve* (unaligned) minne-operasjoner (derav navnet på algoritmen).

AA-Sort er delt i to algoritmer. Den første av disse er basert på Combsort[1]. Denne klarer fullstendig å unngå skjeve minne-operasjoner ved å transponere array-et før og etter det sorteres. Ved å bruke SIMD-operasjoner klarer også algoritmen fullstendig å unngå forgreninger. Dermed yter algoritmen svært godt på array-er som får plass prosessorens cache-minne.

Den andre algoritmen er designet for å utnytte trådparallellitet. Den deler først tallene som skal sorteres opp i blokker som passer i prosessorens cache-minne. Deretter sorteres hver blokk i parallell med algoritmen beskrevet over. De sorterte blokkene flettes med en variant av flettesortering. Denne varianten bruker odde-par-flettenettverk (se avsnitt 2.2.1 på side 5) implementert med SIMD-operasjoner for å flette fire og fire tall. Dette sikrer at ingen minne-operasjoner er skjeve og kun én if-test for hvert fjerde element som flettes.

I artikkelen blir algoritmen testet på en 4-kjerners PowerPC 970MP-prosessor. Når algoritmen kjøres på 1 kjerne rapporteres den å være 3,0 ganger raskere enn *STL sort()* (C++ sin standard sorteringsmetode). Kjører man den på 4 kjerner, er den hele 8,1 ganger

[1] Combsort [Lacey og Box, 1991] er en variant av boblesortering, men oppnår en tidskompleksitet på $n \log n$ i gjennomsnitt.

raskere.

Quicksort

[Parikh, 2008] presenterer en enkel metode for å parallellisere quicksort på en Pentium 4-prosessor med Hyper-Threading. Dette er en teknologi hvor hver fysisk kjerne fremstår som to virtuelle kjerner for operativsystemet. Når tråder kjører på begge de virtuelle kjernene, vil maskinvaren håndtere vekslingen mellom dem. På denne måten oppnår man bedre ressursutnyttelse av CPU-en.

Algoritmen som presenteres går i korte trekk ut på å dele array-et som skal sorteres i to. Hver av de to halvdelene sorteres i parallell med quicksort. Til slutt flettes de to halvdelene.

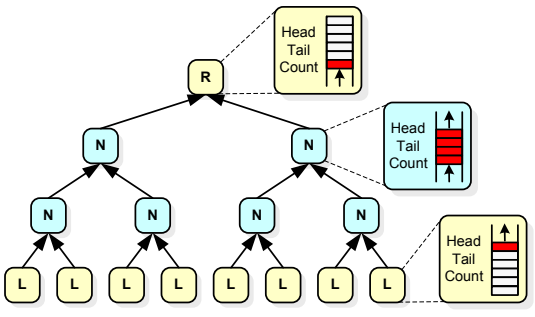
Med denne enkle metoden oppnådde Parikh opp til 58 prosent forbedret ytelse over vanlig quicksort.

Flettesortering

Den nyeste algoritmen, presentert i [Chhugani et al., 2008], er en variant av flettesortering spesielt tilpasset SIMD-arkitektur.

Som AA-Sort er også denne algoritmen delt i to. Den første algoritmen deler arrayet i blokker av størrelse M, hvor M er antall elementer som får plass i prosessorens L2-cache-minne. Hver blokk deles igjen i k biter (hvor k er antall kjerner) som sorteres på hver sin kjerne. Her benytter man bitonisk sortering implementert med SIMD-instruksjoner for å oppnå dataparallellitet. Deretter flettes de k bitene i parallell til én sortert blokk.

Når alle blokkene er sortert, flettes disse i parallell gjennom et tre av små køer (se figur 2.5). Hver av løvnodene, markert L i figuren, representerer en sortert blokk. Hver intern node, markert N, representerer en tråd med en kort FIFO-kø. Køene har plass til ca. 100 elementer hver. Størrelsen på køene er justert slik at de tilsammen får plass i prosessorens L2-cache-minne. Til slutt skriver rotnoden, markert R, de flettede elementene tilbake i resultat-array-et. Målet med denne metoden er å redusere båndbredden mot minnet ved å gjøre all fletting av data i cache-minnet.



Figur 2.5: Parallell fletting ved hjelp av køer [Chhugani et al., 2008]

Denne kombinasjonen viser seg å være svært effektiv. På en Intel Q9550 Quad-prosessor klarer man å sortere 32 millioner tall på 0,2429 sekunder. Dette er 3,1 ganger raskere enn CellSort, 2,1 ganger raskere enn AA-Sort og 1,7 ganger raskere enn radix-algoritmen

presentert i [Satish et al., 2008]. Sistnevnte algoritme beskrives i avsnitt 2.4.3 på neste side.

2.3.2 Fellesnevnere

Ut ifra artiklene vi har studert, virker det som trenden innen forskning på flerkjernet sortering ligger i å unytte SIMD-arkitektur i størst mulig grad. Algoritmene har gjerne en indre og en ytre delalgoritme. Den indre delen er gjerne optimalisert for å sortere blokker i størrelser som passer i prosessorens cache-minne. Deretter blir disse blokkene flettet sammen til en sortert liste av en ytre algoritme som utnytter trådparallellitet.

2.4 Sortering på skjermkort

2.4.1 Hva er GPU

Skjermkort har blitt en viktig del av alle moderne datamaskiner. Slike kort tar seg av det meste som har å gjøre med generering av 2D- og 3D-grafikk, og avlaster dermed CPU-en betraktelig. For å håndtere disse oppgavene er skjermkortene utstyrt med en spesialdesignet prosessor for nettopp disse formålene. Denne prosessoren kalles en *Graphics Processing Unit* (GPU) [Wikipedia, 2010a].

Den viktigste drivkraften bak utviklingen av moderne skjermkort er spillmarkedet. Nye spill setter stadig større krav til skjermkortet, og resultatet er at disse kortene over tid har utviklet en enorm regnekraft. I den siste tiden har regnekraften til skjermkortene, målt i antall piksler beregninger per sekund, doblet seg hver sjette måned [Fernando, 2004, forord].

Skjermkort er først og fremst beregnet på å manipulere bilder. Som regel betyr dette at de får en enkel operasjon som skal utføres på alle piksler samtidig. De er derfor organisert i en *Stream Processing*-arkitektur, som betyr at de kan håndtere svært mange tråder samtidig. Bytting mellom trådene håndteres i maskinvaren. Derfor kan moderne skjermkort håndtere millioner av tråder, uten tap av ytelse.

2.4.2 Hva er GPGPU

Moderne skjermkort har demonstrert enorm kapasitet til å gjøre parallelle beregninger. Samtidig er de i første omgang beregnet på konsumentmarkedet, og er derfor relativt billige. Denne kombinasjonen har gjort at forskere har vist stor interesse for å bruke GPU som medprosessor til vitenskapelige beretninger [Coombe og Harris, 2005, side 493]. I nyere tid har mye forskning dreiet seg om å finne effektive metoder for å utnytte regnekraften i GPU-er. Det generelle begrepet for denne typen forskning er *General Purpose Computing on GPU* (GPGPU).

I de første forsøkene på å gjøre vanlige beregninger med skjermkort hadde man kun tilgang til skjermkortene gjennom grafikk-API-et. Den vanligste måten å programmere GPGPU var gjennom OpenGL. GPUSort [Govindaraju et al., 2005] og GPU-ABiSort [Greß og Zachmann, 2006] er eksempler på algoritmer som er implementert ved hjelp av grafikk-API.

Etterhvert som interessen for GPGPU økte, kom det også flere rammeverk for å gjøre det enklere å programmere mot skjermkort. Disse rammeverkene gjør det mye enklere å utnytte kraften i skjermkortene ved at de gir tilgang til GPU-en gjennom et normalt programmeringsgrensesnitt. Ofte tilbyr de en utvidet versjon av programmeringsspråk som C eller C++. Ved kompilering oversettes disse språkene til kode som kan kjøre på skjermkortet. På den måten slipper programmereren å forholde seg til grafikk-API-et som en kompliserende faktor. Her er en kort liste over de mest kjente rammeverkene:

CUDA [NVIDIA, 2010]: Rammeverk for programmering av NVIDIA-skjermkort. Er tilgjengelig både som utvidet syntaks av C++ og som et rent C-API. Gir direkte tilgang til maskinvaren på NVIDIA-kort uten å gå gjennom grafikk-API-er.

Stream [AMD, 2010]: Rammeverk for programmering av AMD/ATI-skjermkort. Baserer seg på BrookGPU, men spesielt optimalisert for AMD/ATI-kort. Gir direkte tilgang til maskinvaren.

BrookGPU [Stanford University Graphics Lab, 2010]: Et generelt rammeverk for GPU-programmering utviklet av Stanford University. Programmer skrives i språket *Brook* som er en utvidet versjon av standard C. Koden kompileres til enten OpenGL v1.3+, DirectX v9+ eller AMD/ATI Stream (se over).

Sh [Intel Corporation, 2010]: C++-bibliotek for GPU-programmering utviklet av RapidMind. Programmer kompileres av en vanlig C++-kompilator og oversettes til GPU-kode under kjøring. OpenGL-standarden benyttes for å kjøre kode på skjermkort.

OpenCL [Khronos Group, 2010]: En åpen standard for GPU-programmering. Opprinnelig utviklet av Apple, men nå vedlikeholdt av Khronos gruppen. Er en åpen industristandard på lik linje med OpenGL. Støttet av flere store aktører som AMD, NVIDIA og VIA.

DirectCompute [Microsoft Corporation, 2010]: Standard for GPU-programmering utviklet av Microsoft. DirectCompute er en del av DirectX-samlingen med API-er. Støttet av både NVIDIA og AMD.

2.4.3 Sortering på GPU

Mens skjermkort først og fremst egner seg til å løse regneintensive problemer, er sortering først og fremst I/O-intensivt. Data leses og skrives til minnet flere ganger, og beregningene som gjøres er stort sett svært enkle.

Allikevel har mye forskning gått med til å finne effektive sorteringsalgoritmer for GPU. En av årsakene til dette er at skjermkort har opp til 10 ganger større båndbredde mot minnet enn vanlige CPU-er, og kan også gjøre svært mange beregninger i parallell. De er derfor verdt å se nærmere på.

2.4.4 Algoritmene

Et stor antall sorteringsalgoritmer har blitt utviklet for GPU. I de neste avsnittene har vi plukket ut noen av de viktigste og mest kjente algoritmene, og tatt med en kort beskrivelse av hver.

Bitonisk sortering

[Purcell et al., 2003] presenterer en av de første sorteringsalgoritmene for GPU. Denne algoritmen var basert på bitoniske sorteringsnettverk (se kapittel 2.2.2 på side 5). På denne tiden kunne man kun programmere skjermkort gjennom grafikk-API-et. Disse hadde ikke støtte for *scatter/gather*-operasjoner (lesing/skriving til spredte minneadresser), så sorteringsnettverk var stort sett de eneste algoritmene man kunne benytte.

GPUSort

GPUSort [Govindaraju et al., 2005] var en ny algoritme basert på bitonisk sortering, men bedre tilpasset maskinvaren den kjørte på. Ved å utnytte *texture mapping*- og *blending*-funksjonaliteten til GPU-er, klarte man å lage en svært effektiv implementasjon som gav signifikante forbedringer over tidligere algoritmer. Artikkelen viser at GPUSort er 6–25 ganger raskere enn quicksort på CPU. I flere år var denne algoritmen en av de beste for sortering på GPU.

GPU-ABiSort

GPU-ABiSort [Greß og Zachmann, 2006] var den første sorteringsalgoritmen for GPU som var asymptotisk optimal:

> "For sorting n values utilizing p stream processor units, this approach achieves the optimal time complexity $O((n \log n)/p)$."
>
> [Greß og Zachmann, 2006, side 1]

For å oppnå denne tidskompleksiteten, var algoritmen basert på adaptiv bitonisk sortering i motsetning til vanlig bitonisk sortering. Dessverre var konstantene skjult bak denne tidskompleksiteten så store at det var få tilfeller hvor GPU-ABiSort faktisk var den raskeste algoritmen.

[Greß og Zachmann, 2006] demonstrerer at GPU-ABiSort er 1,9–2,6 ganger raskere enn *STL sort()* for $n \geq 2^{17}$.

Global-Radix Sort

I [Sengupta et al., 2007] presenteres en effektiv implementasjon av *Parallell Prefiks Sum*-operasjon (også kjent som *scan*) for GPU. De argumenterer for at dette er en nyttig operasjon i svært mange parallelle algoritmer og gir flere eksempler på dette. Blant annet lager de en implementasjon av radix-sortering som er ca. 3,5x raskere enn *STL sort()* på CPU. Dette var blant de raskeste algoritmene for sortering på GPU så langt.

GPU-Quicksort

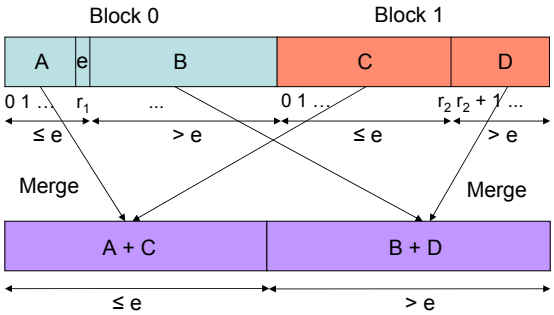
[Cederman og Tsigas, 2008] presenterte i 2008 en effektiv implementasjon av quicksort på GPU. Tidligere var denne algoritmen regnet som uegnet for GPU, men med bedre tilgang til maskinvaren gjennom CUDA viste Cederman at også denne algoritmen effektivt kunne

implementeres på GPU. Tester viser at algoritmen er ca. 10 ganger raskere enn *STl sort()* på CPU. Den er også raskere enn både GPUSort og Global-Radix Sort.

Forbedret radix- og flettesortering

I [Satish et al., 2008] presenteres to nye effektive sorteringsalgoritmer. Den første av disse er en videreutvikling av Global-Radix Sort (beskrevet i avsnitt 2.4.4 på forrige side). Algoritmen fokuserer på å minimere antall *scatter/gather*-operasjoner, som kan ha en dramatisk effekt på ytelsen (se avsnitt 5.1.4 på side 49, *Koalesert minneaksess*). Dette oppnår de ved først å delsortere dataene i delt minne (på multiprosessoren) før de skrives tilbake til globalt minne (på skjermkortet). Artikkelen demonstrerer at denne algoritmen var opp til 4 ganger raskere enn GPUSort.

Den andre algoritmen er en variant av flettesortering. Tallene som skal sorteres deles først opp i mindre blokker som effektivt sorteres i delt minne ved å bruke et bitonisk sorteringsnettverk. Deretter settes blokkene sammen med flettesortering. For å utnytte parallelliteten på skjermkortet deles blokkene som skal flettes opp i mindre biter som kan flettes uavhengig av hverandre. Figur 2.6 viser prinsippet for denne oppdelingen. Ved å sørge for at delblokkene har maksimalt 256 elementer hver kan man effektivt flette blokkene i delt minne før de skrives til resultat array-et. Denne implementasjonen var opp til 2 ganger raskere enn GPUSort. Med andre ord var den ikke like rask som radix-algoritmen, men raskere enn alle tidligere kjente algoritmer basert på sammenligning.



Figur 2.6: Deling av array-er for parallell fletting [Satish et al., 2008]

Bøttesortering

Den mest effektive algoritmen så langt er en variant av bøttesortering presentert i [Chen et al., 2009]. Hovedideen bak algoritmen er å først dele array-et som skal sorteres i mange små *stykker* (slices). Hvert element i stykke i er mindre enn elementene i stykke $i+1$. Deretter settes stykkene sammen i bøtter, slik at hver bøtte har så nær som mulig (men ikke mer enn) M elementer. Deretter sorteres hver bøtte med en bitonisk sorteringsalgoritme. Størrelsen M er valgt slik at hver bøtte får plass i delt minne, og kan sorteres av en trådblokk. Algoritmen er også adaptiv i forhold til distribusjon på tallene som skal sorteres.

Tester viser at algoritmen er 10 til 20 ganger raskere enn *STL sort()*, og yter bedre enn både GPU Quicksort og GPU Radix Sort [Chen et al., 2009, side 289].

2.4.5 Fellesnevnere

I algoritmene beskrevet over ser vi at man har skiftet fokuset fra sorteringsnettverk til en rekke nye algoritmer. Dette skyldes først og fremst fremveksten av nyere og bedre rammeverk for GPU-programmering. Disse gjør det enklere å programmere for GPU, og gir bedre utnyttelse av maskinvaren.

Blant de nyeste og mest effektive algoritmene ser vi samme trend som CPU algoritmene. Man kombinerer to algoritmer; en optimert for å sortere små blokker, og en som setter sammen disse blokkene til én sortert liste.

Vi ser også at de mest effektive GPU-algoritmene er organisert slik at hver tråd håndterer ett eller noen få elementer hver.

2.5 Veien videre

I denne oppgaven har vi valgt å benytte Java som plattform for å implementere algoritmene. Siden Java ikke har SIMD-instruksjoner, kommer vi kun til å fokusere på trådparallellitet.

Vi ønsker å forsøke å implementere våre egne algoritmer helt fra bunnen av, og se hvilke erfaringer vi gjør underveis. På denne måten håper vi å kunne få førstehåndserfaring med plattformene, og se hvilke problemer som kan oppstå.

Vi håper andre vil kunne bruke denne oppgaven for å få en innføring i utvikling av parallelle algoritmer.

Kapittel 3

Metode

I forrige kapittel gikk vi gjennom hvilken forskning som er gjort på parallell sortering tidligere. I dette kapittelet vil vi snevre oss inn mot arbeidet vi kommer til å gjøre i denne oppgaven.

Først vil vi presentere algoritmene vi har valgt. For hver av algoritmene vil vi kort repetere hvordan de er bygget opp. Deretter vil vi gå igjennom den sekvensielle implementasjonen, som senere vil bli brukt som referanse for de parallelle algoritmene.

Videre vil vi se nærmere på teorien bak målingene vi kommer til å gjøre. Så gir vi en kort presentasjon av klassen *BenchMark*, som vi vil benytte for å måle ytelsen til algoritmene.

Til slutt vil vi presentere testmiljøet vi kommer til å benytte og gi en kort oversikt over de viktigste delene av maskinvaren.

3.1 Sorteringsalgoritmene

En av de første avgjørelsene vi måtte ta var hvilke algoritmer vi skulle forsøke å parallellisere. Som vi skrev i introduksjonen, bestemte vi oss tidlig for å ta utgangspunkt i sortering. Dette gjorde vi av flere grunner:

1. Sortering er parallelliserbart. Vi mener at sortering er et passe komplisert problem å parallellisere. På den ene siden er det lett å løse deler av problemet i parallell. Men samtidig er det vanskelig å sette sammen disse løsningene på en effektiv måte.

2. Sortering er spennende. Både veileder og kandidat har interesse for emnet. Dermed var mange av forkunnskapene på plass, og vi kunne fokusere på selve parallelliseringen.

3. Sortering er kjent. Alle som jobber med programmering har på et tidspunkt vært borti sortering. Derfor vil det være enklere for lesere av oppgaven å følge med på problemstillingen enn om vi for eksempel hadde valgt partielle differensiallikninger.

3.1.1 Valg av algoritmer

Innenfor sortering finnes det et hav av algoritmer å velge mellom. Tabell 3.1 på neste side gir en kort oversikt over noen av algoritmene vi vurderte å benytte.

Algoritme	Type	Rekursiv
Flettesortering	Sammenlikning	Ja
Vanlig radix-sortering	Distribusjon	Nei
Venstre-radix-sortering	Distribusjon	Ja
Heapsort	Sammenlikning	Nei
Quicksort	Sammenlikning	Ja

Tabell 3.1: Noen mulige sorteringsalgoritmer

Vi ønsket å forsøke to forskjellige typer algoritmer, og bestemte oss derfor for å velge en distribusjonsbasert og en sammenlikningsbasert algoritme. Videre var vi interessert i å finne ut av om rekursive algoritmer (splitt-og-hersk-algoritmer) generelt er god egnet for parallellisering. Vår teori er at slike algoritmer kan parallelliseres ved å fordele de uavhengige delproblemene ut over flere kjerner, og dermed oppnå en forbedring i hastighet.

Vi ekskluderte *vanlig radix-sortering* fordi den ikke var rekursiv. Denne algoritmen har også vært grundig forsket på, og det er kjent at den har effektive parallelle implementasjoner både på CPU og GPU (se kapittel 2).

Også *heapsort* ble ekskludert fordi den ikke er rekursiv. Dessuten ser vi ingen opplagt måte å parallellisere denne algoritmen.

Til slutt ekskluderte vi også *quicksort* fordi [Cederman og Tsigas, 2008] allerede har vist at denne har en effektiv GPU-implementasjon.

Dermed står vi igjen med *flettesortering* og *venstre-radix-sortering*. Flettesortering er allerede en godt studert algoritme, men vi mener den passer godt med typen algoritme vi ønsker å teste. Venstre-radix-sortering er en spennende algoritme vi ikke har sett mye til i forskningen vi har studert.

3.1.2 Flettesortering

Beskrivelse

Flettesortering er en av de mest grunnleggende sorteringsalgoritmene. Fordi den er så enkelt formulert, blir den ofte brukt i lærebøker som første eksempel på splitt-og-hersk-algoritmer. Den baserer seg på en grunnleggende ide: Det er enkelt å slå sammen to sorterte lister til én større sortert liste. Selve algoritmen beskrives ofte som følgende:

> *Basissteg:* Hvis listen er av lengde 0 eller 1 er den sortert.
>
> *Rekursivt steg:* Del listen i to (ca.) like store biter. Sorter hver bit rekursivt. Flett de to sorterte listene til én sortert liste.

Siden algoritmen i hvert rekursivt steg deler listen i to, får vi maksimalt $\log n$ nivåer i rekursjonstreet. De to sorterte listene flettes i lineær tid ved å plukke det minste elementet fra starten av hver liste inntil begge listene er tomme. Dermed får algoritmen en kjøretid på $n \log n$. Siden dette er en algoritme basert på sammenlikninger, vet vi dermed at kjøretiden er asymptotisk optimal.

Implementasjon (F0: Referanse)

Vår implementasjon av flettesortering er basert på [Jodd Team, 2010]. Dette er en helt vanlig implementasjon av flettesortering som benytter ett ekstra hjelpe-array til å flette elementene. Vi har tilpasset vår koden noe siden vi kun fokuserer på sortering av 32-bits positive heltall. Et utdrag av Java-koden er vist i algoritme 3.1.

Algoritme 3.1 Referanseimplementasjon flettesortering

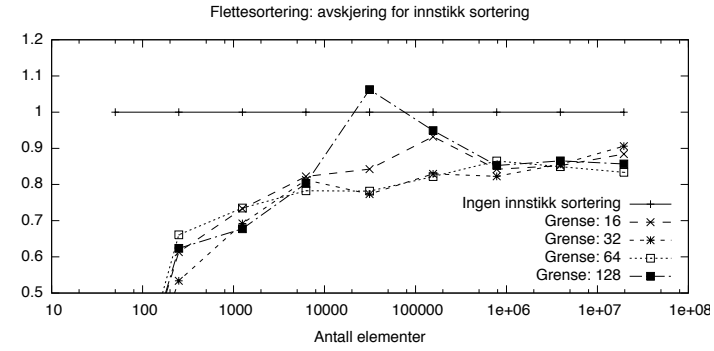
```
public class MergeSort extends SortAlgorithm {
  ⋮
  public void sort(int[] array) {
    mergesort(array, array.clone(), 0, array.length);
  }

  private void mergesort(int[] dst, int[] src, int start, int stop) {
    if(stop-start < insertLimit) {
      «Gjør innstikk sortering»
    } else {
      int mid = (start + stop) >> 1;
      mergesort(src, dst, start, mid);
      mergesort(src, dst, mid, stop);

      int p = start;
      int q = mid;

      for(int i=start; i<stop; i++) {
        if(q >= stop || (p < mid && src[p] < src[q])) {
          dst[i] = src[p++];
        } else {
          dst[i] = src[q++];
        }
      }
    }
  }
}
```

Hovedideen bak implementasjonen er at man kopierer så lite data som mulig ved å gjøre flettingen frem og tilbake mellom to array-er. Siden algoritmen er rekursiv ovenfra og ned, er vi sikret at resultatet vil havne i det opprinnelige array-et, og ikke kopien.

Når delen av array-et som skal sorteres kommer under en viss grense (*insertLimit*), benyttes innstikksortering for å sortere elementene. For å finne den beste verdien for denne grensen, har vi testet flere forskjellige verdier og plottet dem i en graf. Resultatene er vist i figur 3.1 på neste side. Her ser vi at det ikke er noen verdi som er tydelig bedre enn de andre, men 64 ser ut til å være den jevnt over beste verdien. For å holde oss innenfor tidsrammene til oppgaven, vil vi i alle senere flettealgoritmer i dette kapittelet anta at 64 er den beste grensen for innstikksortering.

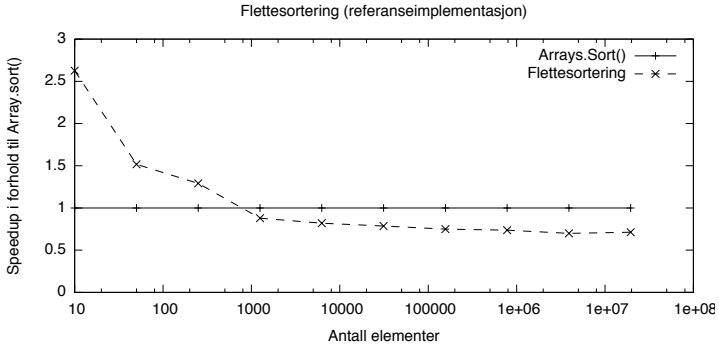
Ytelse

I figur 3.2 på neste side ser vi en sammenlikning av ytelsen mellom denne implementasjonen og Javas innebygde sortering. Her ser vi at ytelsen varierer en god del. Implementasjon vår er en del raskere for array-er opp til ca. 1000 elementer, men 20 til 40



Figur 3.1: Grenseverdi for innstikksortering

prosent tregere for større array-er. Dette spiller ikke så stor rolle, da vi senere i oppgaven kommer til å fokusere på de relative forbedringene. Vi er mer opptatt av effekten av parallellisering enn den totale ytelsen til slutt.



Figur 3.2: Sammenlikning med Javas innebygde sortering

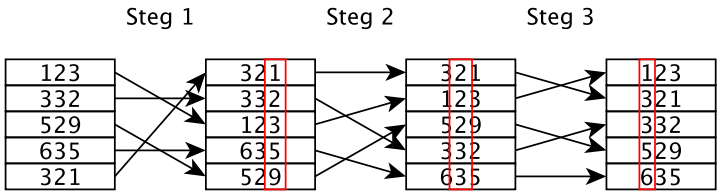
3.1.3 Venstre-radix-sortering

Beskrivelse

Denne algoritmen er en variant av vanlig radix-sortering. Derfor er det enklest å forklare vanlig radix først, og deretter vise hvordan venstre-radix skiller seg fra denne.

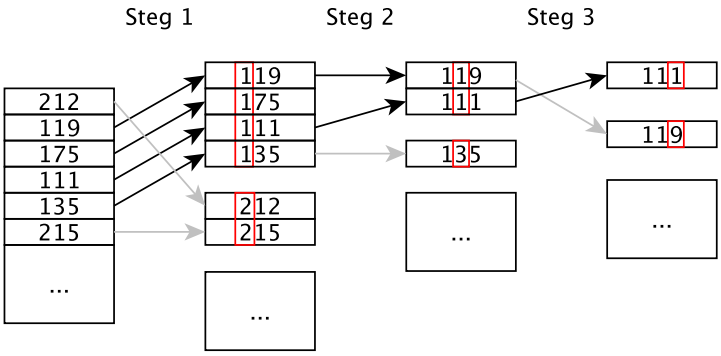
Vanlig radix stammer fra før datamaskinens tid og baserer seg på å sortere tallene på ett og ett siffer. Først sorteres de etter siste siffer, deretter nest siste og så videre. Når man har sortert på alle sifrene er tallene sortert. Hvis man for eksempel sorterer tall i vanlig 10-tall system, er det vanlig å fordele tallene i ti bunker nummerert 0 til 9. Tall som ender på 0 havner i 0-bunken, og så videre. På datamaskiner representeres tallene binært, og man jobber derfor med bit som sifre. Det er ofte lite effektivt å bare fordele på ett bit av gangen, så derfor bruker man ofte 6–10 bit av gangen (og sorterer dermed i 64–1024 "bunker"). Figur 3.3 på neste side viser et eksempel på vanlig radix sortering.

Venstre-radix-sortering skiller seg fra den vanlige algoritmen ved at man sorterer tallene fra venstre mot høyre. Altså starter man med å sortere på første (mest signifikante) siffer. Derfor kalles også ofte venstre-radix-sortering for MSB-radix-sortering (Most Significant



Figur 3.3: Vanlig radix-sortering

Bit). For at tallene nå skal bli sorterte, kan man ikke bare slå sammen bunkene og fordele på neste siffer. Istedenfor må man sortere hver bunke for seg på neste siffer, og igjen hver av de nye bunkene på sifferet etter det. Om man sorterer for hånd er dette en tungvinn metode, men på datamaskiner så løses det elegant ved å bruke rekursjon. Figur 3.4 viser et eksempel på venstre-radix-sortering.



Figur 3.4: Venstre-radix-sortering

Med tanke på parallellisering, er det en vesentlig forskjell på vanlig radix-sortering og venstre-radix-sortering. I vanlig radix flyttes tallene til vilkårlige steder i array-et ved hvert steg i algoritmen. I venstre-radix-sortering flyttes derimot tallene kun innenfor sin egen bøtte. Med andre ord kan man i venstre-radix etter første steg sortere hver enkelt bøtte helt uavhengig av de andre. Dette er en egenskap vi tror vil gjøre det lettere å parallellisere algoritmen.

Implementasjon (R0: Referanse)

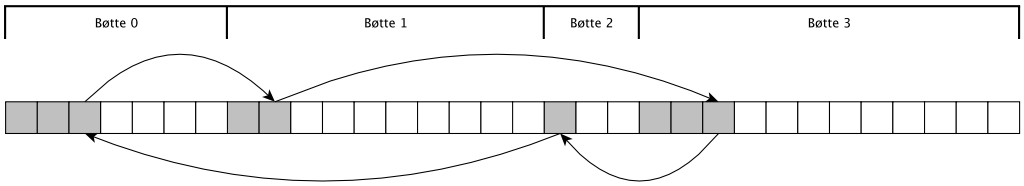
For å spare tid har vi benyttet en ferdig implementert venstre-radix klasse. Algoritmen vi benytter kalles Adaptiv venstre-radix-sortering og er hentet fra [Maus, 2002]. Implementasjonen består av fire hovedsteg:

1. Beregn maskestørrelse Maskestørrelsen er det samme som antall bit vi sorterer på i et gitt nivå. Dersom masken er på b bit, vil vi fordele tallene i 2^b bøtter. Det at vi beregner maskestørrelsen dynamisk underveis, gjør at algoritmen blir *adaptiv*. Vi forsøker å velge en størrelse b slik at tallene fordeles i bøtter jevnest mulig. Hvis vi i dette nivået skal sortere n tall, forsøker vi å finne største verdi av b slik at $2^b <= n$. Samtidig har vi en øvre grense $b <= bMaks$ for at indeks array-et skal få plass i L1-cache-minnet, og en nedre grense $b >= bMin$ for å sikre en viss progresjon. I vår implementasjon har vi

brukt $bMaks = 8$ og $bMin = 2$. Disse verdiene kan finjusteres etter hvilken CPU man kjører på.

2. Tell elementer Når maskestørrelsen er bestemt, teller vi antall elementer i hver bøtte. Dette er nødvendig å vite når vi senere skal stokke om elementene slik at hvert element havner i riktig bøtte.

3. Fordel i bøtter Når vi har talt opp antall elementer, fordeler vi dem i riktige bøtter ved å bruke permutasjonssykler. Hver permutasjonssykel består av å plukke opp det første usorterte elementet, og beregne elementets nye posisjon. Elementet plasseres i denne posisjonen samtidig som det gamle elementet i denne posisjonen plukkes opp. Deretter beregner man ny posisjon for *dette* elementet. Slik fortsetter man til ett av elementene havner i den opprinnelige posisjonen. Figur 3.5 viser et eksempel på en permutasjonssykel. Vi fortsetter med nye permutasjonssykler inntil alle elementene er sortert i riktig bøtte.

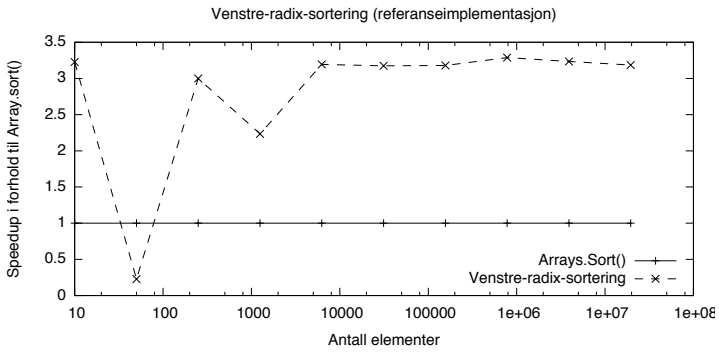


Figur 3.5: Permutasjonssykel (grå elementer er nå plassert i riktig bøtte)

4. Sorter rekursivt Etter at tallene er fordelt i bøtter, er det slik at alle tall i bøtte i er mindre enn alle tall i bøtte $i + 1$. Hver av disse bøttene sorterer vi rekursivt. Når alle bøttene er sortert, så er hele array-et sortert.

Ytelse

Vi har også sammenlignet denne algoritmen med Javas innebygde sortering, og resultatet er vist i figur 3.6. Vi ser at venstre-radix-algoritmen er stort sett 3 ganger raskere enn Java sin algoritme.



Figur 3.6: Sammenlikning med Javas innebygde sortering

3.2 Målemetoder

Senere i oppgaven kommer vi til å utvikle en rekke nye parallelle algoritmer. Da er det viktig å ha en pålitelig måte å måle ytelsen til hver av dem, slik at vi får resultater vi kan stole på. Resultatene vil vi bruke til å sammenligne ytelsen mellom de forskjellige algoritmene.

Når vi skal måle ytelsen til en algoritme, er det mange faktorer man kan basere seg på. Noen av de viktigste er:

- Kjøretid
- Tidskompleksitet
- Minneforbruk
- Antall skriveoperasjoner til minnet
- Antall treff/bom i cache-minnet

Veldig ofte ser man kun på tidskompleksitet når man velger algoritme. Denne gir et godt bilde av hvordan kjøretiden skalerer når antall elementer som skal sorteres øker. Allikevel har modellen en stor svakhet. Den tar ikke hensyn til effekten av cache-minnet på aksesstiden til hukommelsen. Derfor kan det være svært stor forskjell på to algoritmer med samme tidskompleksitet.

En enklere måte å teste ytelsen til algoritmen, er å måle kjøretiden ved praktiske forsøk. Dette er en rask og enkel måte å sammenligne algoritmer på. I motsetning til beregning av tidskompleksitet får man her vite hvor lang tid algoritmen tar på faktisk maskinvare. Men også denne metoden har noen begrensninger. For å kunne sammenligne to algoritmer må begge algoritmene testes på samme maskin. Dessuten er det vanskelig å ta hensyn til hvordan algoritmene vil oppføre seg med andre distribusjoner av data enn akkurat de som er testet.

De andre nevnte faktorene er også viktige å ta hensyn til om man skal lage gode algoritmer. Men siden vi i denne oppgaven skal lage mange algoritmer og har begrenset tid til rådighet, har vi valgt å holde oss kun til praktisk måling av kjøretid på én testmaskin.

3.2.1 Måle kjøretid i Java

I Java finnes det flere alternative metoder for å måle kjøretid. Her er en kort beskrivelse av de mest relevante metodene, og hvordan de skiller seg fra hverandre:

System.currentTimeMillis()

Denne funksjonen har vært tilgjengelig i Java siden begynnelsen (JDK 1.0), og har tradisjonelt vært den vanligste metoden å bruke for å måle tid i Java. Metodekallet returnerer antall millisekunder fra 1. januar 1970. Dessverre har denne metoden få garantier for nivå på presisjon, og det er opp til operativsystemet hvor ofte telleren oppdateres. På noen operativsystemer er oppløsningen så dårlig at telleren oppdateres med flere titalls millisekunder mellom hvert tikk.

System.nanoTime()

Dette er en nyere funksjon som ble lagt til i Java versjon 1.5. Et kall på denne funksjonen gir antall nanosekunder fra et fast, men vilkårlig valgt, tidspunkt (som også kan være i fremtiden). Derfor kan denne tiden ikke oversettes til et klokkeslett slik *System.currentTimeMillis()* kan. Derimot egner den seg svært bra til å måle differanser i tid, og er garantert å benytte den mest presise telleren operativsystemet kan tilby.

***RDTSC* - Antall CPU-sykluser**

TSC er et x86 register som teller antall CPU-sykluser siden maskinen ble startet. Gitt at man kjente klokkefrekvensen, ble *TSC* registeret tidligere ansett som en svært god høyoppløselig teller. Men nye CPU-er med flere kjerner gjør at denne telleren ikke lenger kan stoles på. Den egner seg også dårlig til å måle parallelle programmer.

Av de nevnte metodene har vi valgt å benytte *System.nanoTime()* for å måle kjøretiden med. Denne har vi valgt fordi den gir oss best presisjon, og den virker å være best egnet for måling av ytelse.

3.2.2 Valg av testdata

For å få et realistisk bilde av ytelsen til en algoritme er det viktig å ha et fornuftig sett med testdata. Siden vi måler ytelsen ved praktiske forsøk, ser vi kun hvordan algoritmen oppfører seg på de typene data vi tester den med. Derfor bør disse dataene være representative for hvordan algoritmen kommer til å bli brukt. Spesielt er det to faktorer vi må ta hensyn til:

Antall elementer

For å gi et godt bilde av hvordan algoritmen yter, må vi teste algoritmen med datasett av forskjellige størrelser. Hvilke størrelser vi skal velge, avhenger av hva vi ønsker å teste. Tradisjonelt pleier man å velge eksponentielt stigende verdier, slik at man får best mulig bilde av hvordan algoritmen skalerer.

Det er fristende å for eksempel velge funksjonen 2^x hvor $x \in [0..20]$. Men siden noen algoritmer er raskere ved potenser av 2, kan denne funksjonen gi et feilaktig bilde på ytelsen. Derfor har vi valgt å benytte følgende funksjon:

$$10 \cdot 5^x, x \in [0..9]$$

Vi tester med andre ord algoritmene med array-er bestående av 10 til 19 531 250 elementer.

Type fordeling

Hva slags fordeling av tall som skal sorteres har ofte stor påvirkning på ytelsen til algoritmen. For eksempel er algoritmen i [Chen et al., 2009] svært rask på tilfeldige data, men treg ved sortering av nesten sorterte data. Tabell 3.2 på neste side viser en oversikt over de vanligste typene fordelinger.

Av disse er helt klart *uniform* fordeling mest vanlig. For å begrense omfanget av oppgaven har vi derfor valgt å kun holde oss til denne fordelingen.

Uniform	`for(int i=0; i<n; i++) a[i] = random() % n;`
Sortert	`for(int i=0; i<n; i++) a[i] = i;`
Nesten sortert	`for(int i=0; i<n; i++) a[i] = i; a[0] = n;`
Omvendt sortert	`for(int i=0; i<n; i++) a[i] = n-i-1;`

Tabell 3.2: Vanlige fordelinger

3.2.3 BenchMark-klassen

For å teste ytelsen til algoritmene vi kommer til å utvikle, har vi laget en enkel klasse kalt *BenchMark*. Denne klassen tar seg av all måling av kjøretid, slik at vi får testet alle algoritmene på en konsekvent måte. Her vil vi kort beskrive hvordan denne virker.

Hvordan vi gjør målingene

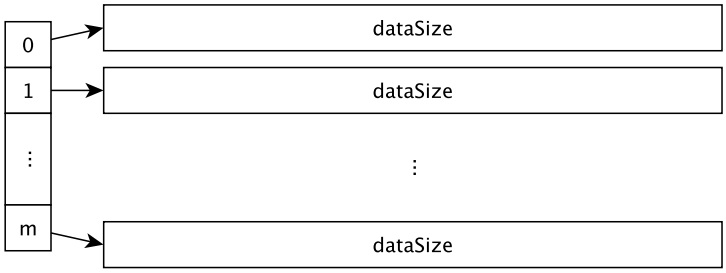
Når vi skal måle kjøretiden for en bestemt testmengde, er det fornuftig å gjøre målingen flere ganger for så å ta et gjennomsnitt av målingene. Spesielt for små array-er er dette viktig, da disse målingene fort kan påvirkes av andre prosesser som kjører på maskinen.

Hvis vi lar størrelsen på datasettet være n og antall målinger være m, så har vi i vår implementasjon latt $n \cdot m = 10 \cdot 5^9$. Samtidig har vi satt en øvre grense for m på 1000, da flere målinger ikke vil bidra nevneverdig til presisjonen. Tabell 3.3 viser hvilke array-størrelser vi tester, og hvor mange målinger vi gjør for hver av dem.

Antall elementer	10	50	250	1250	6250	31250	156250	781250	3906250	19531250
Antall målinger	1000	1000	1000	1000	1000	625	125	25	5	1

Tabell 3.3: Array-lengder som testes, og tilhørende antall målinger

For å unngå å bruke tid på å lage nye datasett mens vi måler ytelsen på algoritmene, genererer vi samtlige datasett før målingene utføres. Figur 3.7 viser hvordan vi lagrer disse datasettene.



Figur 3.7: Samling av datasett som skal testes

Et generelt grensesnitt

For å kunne akseptere en sorteringsalgoritme som parameter, er vi nødt til å definere et grensesnitt som sier at en klasse er en sorteringsalgoritme. Dette har vi valgt å gjøre ved å implementere en abstrakt baseklasse, som alle senere sorteringsalgoritmer vil arve fra.

Denne baseklassen er vist i program 3.1. Vi kunne selvfølgelig også brukt et *grensesnitt* (interface), men på grunn av JUnit 4 var det lettere å bruke en baseklasse.

Program 3.1 Baseklasse for alle sorteringsalgoritmer

```
public abstract class SortAlgorithm
{
  public abstract void sort(int[] array);
}
```

Implementasjon

Den viktigste metoden i BenchMark-klassen er *measure()*. Denne metoden tar en algoritme som parameter, og måler kjøretiden for array-er av størrelsene vist i tabell 3.3 på forrige side. Et skall av klassen er vist i program 3.2.

Program 3.2 Generell oppbygging av BenchMark-klassen

```
class BenchMark {
  ⋮
  public ResultSet measure(SortAlgorithm algorithm) {
    // Measure performance of algorithm
  }
}
```

Selve målingene gjøres på en likefram måte. For hver array-størrelse som skal måles gjør vi følgende fire steg:

1. Lag tilfeldige datasett. Først genererer vi tilfeldige datasett som algoritmen skal sortere. Hvor mange datasett som genereres bestemmes av tabell 3.3 på forrige side. Program 3.3 viser hvordan vi allokerer og genererer datasettene.

Program 3.3 Generering av datasett

```
int[][] data = new int[dataSets][dataSize];

for(int i=0; i<dataSets; i++) {
  fillWithRandomElements(data[i]);
}
```

2. Sorter datasett med referansealgoritme. Deretter lager vi en kopi av datasettene som vi sorterer med en algoritme vi vet er korrekt (her bruker vi Javas innebygde quicksort). Senere vil vi kontrollere at algoritmen som testes har sortert tallene riktig ved å sammenligne med disse kopiene. Program 3.4 på neste side viser hvordan vi lager denne referansekopien.

3. Mål kjøretiden på algoritmen. Videre måler vi tiden algoritmen bruker på å sortere datasettene. Implementasjonen vi benytter er vist i program 3.5 på neste side.

Program 3.4 Lage sorterte kopier av datasett

```
int[][] referenceSets = new int [dataSets][dataSize];

for(int j=0; j<dataSets; j++) {
  System.arraycopy(data[j], 0, referenceSets[j], 0, dataSize);
  Arrays.sort(referenceSets[j]);
}
```

Program 3.5 Mål kjøretiden til algoritmen

```
for(int i=0; i<dataSets; i++) {
  start = System.nanoTime();
  algorithm.sort(data[ji);
  stop = System.nanoTime();

  totalTime += stop - start;
}
```

4. Sjekk at resultatet er korrekt. Til slutt går vi gjennom datasettene sortert av algoritmen som skal testes og kontrollerer at disse er sortert riktig. Hvis resultatet ikke er riktig, avbryter vi kjøringen og skriver ut en feilmelding.

3.3 Testmiljø

For å sammenligne algoritmene vi kommer til å utvikle, har vi valgt å benytte en testmaskin med en 4-kjerners *Intel*-prosessor og et *NVIDIA*-skjermkort. Vi kommer til å måle kjøretiden til alle algoritmene på denne maskinen, og se på resultatene opp mot hverandre.

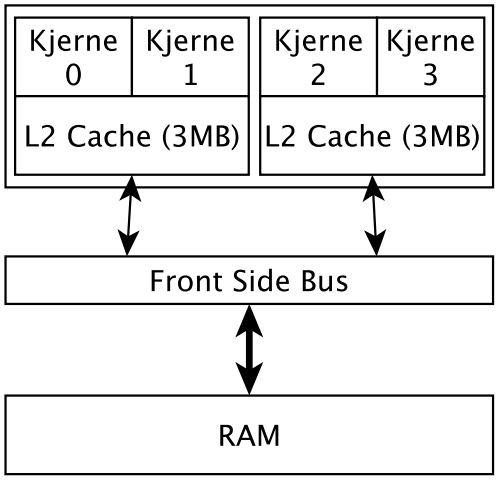
At vi har valgt å kun benytte én testmaskin har flere årsaker. Først og fremst prøver vi å begrense arbeidsmengden da vi kun har noen uker til å utvikle og teste hver algoritme. Vi er også mest interessert i å studere prosessen med å utvikle en algoritme for en bestemt plattform. Vi er derfor klar over at andre plattformer kan gi forskjellige resultater.

3.3.1 Testmaskinen

Forskningsgruppen vår hadde alt tilgjengelig en egnet testmaskin. En kort oversikt over maskinvaren i denne er vist i tabell 3.4. I de neste to avsnittene vil vi se nærmere på prosessoren og skjermkortet i denne maskinen.

Prosessor	Intel® Core™2 Quad CPU Q9400 @ 2,66GHz
Skjermkort	NVIDIA GeForce GTX 285
Minne	4GB
Operativsystem	Windows Server 2007 Service Pack 2 64bit

Tabell 3.4: Oversikt over testmaskin



Figur 3.8: Oversikt over CPU kjerner og cache-minne

3.3.2 Prosessor

Prosessoren i testmaskinen er en Intel Core^{TM}2 Quad CPU. Den har fire kjerner, hver med 64KB L1-cache-minne fordelt på 32KB data-cache og 32KB instruksjons-cache. Den har også 6MB L2-cache-minne, fordelt på to og to kjerner (se figur 3.8). Denne inndelingen skyldes at prosessoren egentlig består av to Core^{TM}2 Duo CPU-er integrert på en brikke. Hvis kjerne 1 og 3 skal kommunisere med hverandre må de derfor gå gjennom FSB-en. Det er også verdt å merke seg at CPU-en ikke har *Hyper-Threading*, så den kan maksimalt kjøre fire tråder samtidig.

3.3.3 Skjermkort

Vi har valgt å benytte et NVIDIA GTX 285-kort som vi tester alle GPU-algoritmene våre på. Dette er et av de beste NVIDIA-kortene tilgjengelig, og har 240 prosesseringselementer.

NVIDIA har i det siste gjort mye for å tilpasse kortene sine for GPGPU-programmering. De har også laget svært gode verktøy for å utvikle programmer på GPU-ene deres. Dette kombinert med at det fins mye kompetanse på denne plattformen innen forskningsgruppen, gjorde at vi valgte å bruke denne typen kort.

Tabell 3.9 på neste side gir en generell sammenligning mellom GPU-en og CPU-en i testmaskinen vår. For nærmere beskrivelse av GPU-ens oppbygging, se kapittel 5.1 på side 43.

	Intel Core 2 Q9400	NVIDIA GTX 285
Transistorer	456 millioner	1,4 milliarder
Klokkefrekvens	2,66GHz	1476 MHz
Kjerner	4	240
Cache / delt minne	6MB	16 KB x 30
Maksimalt antall aktive tråder	4	30720
Maksimal FLOPS	42,56 GFLOPS	1062,72 GFLOPS
Minne-båndbredde	ca. 15 GB/s	159 GB/s
Minnekontroller	separat	8 x 64bit

Figur 3.9: Sammenlikning av CPU og GPU

Kapittel 4

Flerkjerneprogrammering på CPU

I dette kapittelet vil vi først gi en kort introduksjon til parallell programmering i Java. Her ser vi på hvordan man starter nye tråder, det nye *Executor*-rammeverket, trådsamlinger og synkronisering.

Deretter vil vi presentere algoritmene vi har utviklet. Vi vil fokusere på prosessen med å utvikle nye algoritmer. Derfor starter hver algoritme med en idé. Deretter beskriver vi hvordan vi har implementert denne. Til slutt sammenligner vi ytelsen til algoritmen med tidligere algoritmer, slik at vi kan se effekten av forbedringen. Hver algoritme bygger videre på det vi lærte i den forrige.

4.1 Parallell programmering i Java

Det er mange situasjoner hvor man kan dra nytte av å gjøre flere oppgaver samtidig. Ofte kan tråder gjøre det lettere å modellere hvordan vi mennesker arbeider ved å ta en asynkron arbeidsflyt og bryte den opp i flere mindre sekvensielle biter [Goetz et al., 2006, side 3]. Det er også slik at en tråd kun kan kjøre på én kjerne av gangen. For å få fullt utbytte av alle kjernene i en flerkjerne-CPU må man derfor opprette minst en tråd per kjerne.

4.1.1 Hvordan opprette tråder?

Den mest grunnleggende måten å lage en ny tråd i Java er å opprette et nytt *Thread*-objekt. Hvert slikt objekt representerer én tråd, og gir programmereren kontroll over denne tråden. Ved å kalle *start()*-metoden vil kjøringen av tråden starte. De andre metodene gir tilgang til blant annet å se trådens tilstand, stoppe tråden eller vente til den er ferdig.

Når man starter en tråd, vil *Thread*-objektets *run()*-metode kalles. Deretter vil tråden avslutte. For at vår egen kode skal kjøre i tråden, må vi enten lage en subklasse av *Thread* eller implementere *Runnable*-grensesnittet. Et eksempel på den første metoden er vist i program 4.1 på neste side.

Hvis man istedenfor velger å implementere *Runnable*-grensesnittet, står man fritt til å arve fra en annen klasse om man vil. Dette kan i mange tilfeller være nyttig. Når man

Program 4.1 Arv fra *Thread*-klassen

```
public class ThreadSample1 {
  public static void main(String[] args) {
    new MyThread().start();
  }
}

class MyThread extends Thread {
  public void run() {
    // Do stuff
  }
}
```

gir et objekt som implementerer *Runnable* til konstruktøren i *Thread*-klassen, vil dette objektets *run()*-metode kjøres når tråden startes. Program 4.2 illustrerer hvordan en ny tråd opprettes på denne måten.

Program 4.2 Implementer *Runnable* grensesnitt

```
public class ThreadSample2 {
  public static void main(String[] args) {
    new Thread(new MyThread()).start();
  }
}

class MyThread implements Runnable {
  public void run() {
    // Do stuff
  }
}
```

I denne oppgaven spiller det ikke så stor rolle hvilken metode vi velger. Vi har allikevel valgt å holde oss til den sistnevnte (implementere *Runnable*) fordi denne virker noe mer fleksibel.

4.1.2 Executor-rammeverket

Oppgaver er logiske arbeidsenheter, mens *tråder* er en mekanisme for å kjøre disse asynkront [Goetz et al., 2006]. Både det å opprette og veksle mellom tråder har en viss kostnad. Om vi oppretter flere tråder enn vi trenger, går mange CPU-sykluser med til trådhåndtering. Det vi egentlig ønsker er å bruke trådene til å utføre oppgaver på en mest mulig effektiv måte. Derfor introduserte Java 1.5 et nytt rammeverk for håndtering av oppgaver.

I dette rammeverket er *Executor*-objekter den primære abstraksjonen for å håndtere oppgaver (se program 4.3 på neste side). Dette gir en økt fleksibilitet i utføringen av parallelle oppgaver som vi vil se i neste avsnitt.

Program 4.3 Executor grensesnittet

```
public interface Executor {
  void execute(Runnable command);
}
```

4.1.3 Trådsamlinger

Sammen med *Executor*-rammeverket kom også en effektiv implementasjon av trådsamlinger (thread pools). Dette er en samling arbeidstråder som utfører oppgaver hentet fra en felles kø. På denne måten trenger brukere av trådsamlinger kun å tenke på hvilke oppgaver som skal utføres, ikke håndteringen av trådene.

Java tilbyr flere forskjellige typer trådsamlinger. Her vil vi bare se kort på en av dem; *FixedThreadPool*. Denne oppretter et fast antall tråder, som hver prosesserer oppgaver fra en felles kø så lenge nye oppgaver er tilgjengelig. Program 4.4 viser et eksempel som demonstrerer hvordan man oppretter en trådsamling, kjører oppgaver og til slutt avslutter trådene.

Program 4.4 Eksempel på bruk av trådsamlinger

```
void processInParallel(List<Runnable> tasks) {
  int numCores = Runtime.getRuntime().availableProcessors();
  ExecutorService exec = Executors.newFixedThreadPool(numCores);

  for(Runnable task : tasks)
    exec.execute(task);

  try {
    executor.shutdown();
    executor.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);
  } catch (InterruptedException e) {
    // Handle exception
  }
}
```

4.1.4 Synkronisering

Alle tråder har tilgang til det samme minneområdet. Kommunikasjon mellom trådene skjer derfor gjennom *felles objekter*. Siden rekkefølgen trådene kjøres i er uforutsigbar, kan utfallet ofte være uventet. Derfor må man *alltid* bruke synkronisering når man aksesserer objekter som andre tråder har tilgang til.

En måte å sørge for synkronisert tilgang er å benytte *låser*. Disse kan tenkes på som stafettpinner som kun én tråd kan holde av gangen. Dersom en tråd holder en lås, vil alle andre tråder som forsøker å ta den måtte vente til den blir frigjort igjen.

Et annet eksempel på synkronisering er *AtomicInteger*. Denne klassen ble innført i Java 1.5 sammen med en rekke andre metoder for synkronisering. AtomicInteger-klassen gir en heltall-verdi med atomiske operasjoner. En tråd kan med andre ord hente den nåværende verdien og oppdatere den i én operasjon. Flere tråder kan dermed trygt bruke verdien uten å benytte låser. Bruk av *AtomicInteger* kan gjøre koden både mer lettlest og mer effektiv.

Algoritmene vi jobber med i denne oppgaven er veldig enkle og har begrenset bruk for synkronisering. Derfor går vi ikke inn i nærmere detalj på dette emnet. For en grundig innføring i parallell programmering i Java, se [Goetz et al., 2006].

4.2 Flettesortering

4.2.1 Første algoritme (F1: Naiv)

Ide: Start alle rekursive kall på *mergesort()* i en ny tråd.

I vårt første naive forsøk på å lage en parallell algoritme har vi forsøkt å tenke så enkelt som mulig. Den eneste endringen vi har gjort på referansealgoritmen (se kapittel 3.1.2 på side 17) er å erstatte alle rekursive kall med å starte en ny tråd.

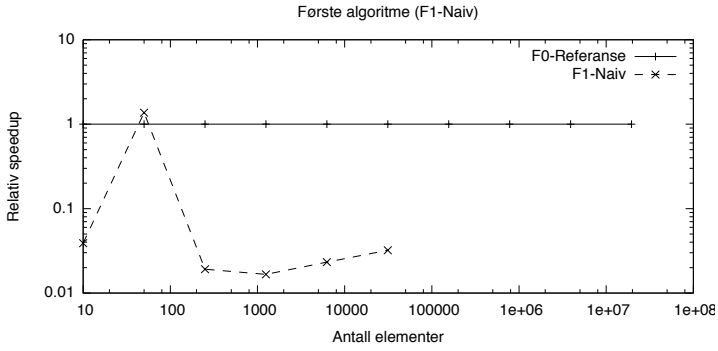
Når vi skal implementere denne endringen, må vi lage en ny klasse med koden som skal kjøre i nye tråder. Vi har valgt å lage en privat indre klasse, og flyttet *mergesort()*-metoden inn i denne. Utdraget i algoritme 4.1 viser hvordan vi har implementert denne.

Algoritme 4.1 Parallell flettesortering: Første algoritme (F1: Naiv)

```
private void mergesort(int[] src, int[] dst, int start, int stop) {
  if(stop-start < insertLimit) {
    ...
  } else {
    ⋮
    Thread t1 = new Thread(new MergesortThread(dst, src, start, mid));
    Thread t2 = new Thread(new MergesortThread(dst, src, mid, stop));

    t1.start();
    t2.start();

    try {
      t1.join();
      t2.join();
    } catch (InterruptedException e) {...}
    ⋮
  }
}
```

Resultat: Figur 4.1 på neste side viser ytelsen sammenlignet med referansealgoritmen. Vi ser at denne implementasjonen er ca. 30 til 60 ganger tregere enn den sekvensielle algoritmen (bortsett fra området under 64 elementer, som håndteres av innstikksortering). Når antall elementer som skal sorteres kommer opp i ca. 130 000, kræsjer algoritmen fordi maksimalt antall tråder er nådd. Derfor er det ingen høyere målinger i figuren.

Årsaken til at denne algoritmen yter så dårlig ligger først og fremst i kostnadene ved å starte en ny tråd. Alle nye tråder som opprettes krever både både minne til stakk og ekstra CPU-sykluser til veksling mellom tråder. Vi har derfor lært at for å oppnå god ytelse, må vi begrense antall tråder som kjører.



Figur 4.1: Resultat: Første algoritme (F1: Naiv)

4.2.2 Andre algoritme (F2: Maks dybde)

Ide: Slutt å lage nye tråder etter en viss rekursjonsdybde.

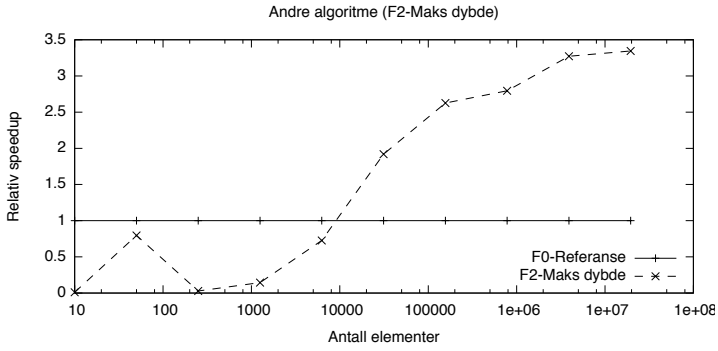
I forrige forsøk lærte vi at vi må begrense antall tråder som startes. Vår neste ide er derfor å slutte å lage nye tråder når vi kommer til en viss dybde i rekursjonstreet. Når vi passerer denne dybden fortsetter vi med vanlig sekvensiell sortering.

For å finne antall nivåer, tar vi utgangspunkt i antall kjerner som er tilgjengelig på maskinen. I Java finner man dette ved å kalle *Runtime.getRuntime().availableProcessors()*. I nivå d vil det startes 2^d nye tråder. Hvis k er antall kjerner, vil vi derfor lage nye tråder i $\log k$ nivåer. Algoritme 4.2 viser hvordan vi har implementert dette.

Algoritme 4.2 Parallell flettesortering: Andre algoritme (F2: Maks dybde)

```
private void mergesort(int[] src, int[] dst, int start, int stop)
{
  if(stop-start < insertLimit) {
    ⋮
  } else {
    ⋮
    if(d < d_lim)
    {
      Thread t1 = new Thread(new MergesortThread(dst, src, start, mid,
          d+1, d_lim));
      Thread t2 = new Thread(new MergesortThread(dst, src, mid, stop, d
          +1, d_lim));
      ⋮
    }
    else
    {
      mergesort(dst, src, start, mid);
      mergesort(dst, src, mid, stop);
    }
    ⋮
  }
}
```

Resultat: I figur 4.2 på neste side ser vi resultatet av denne endringen. Vi ser at algo-



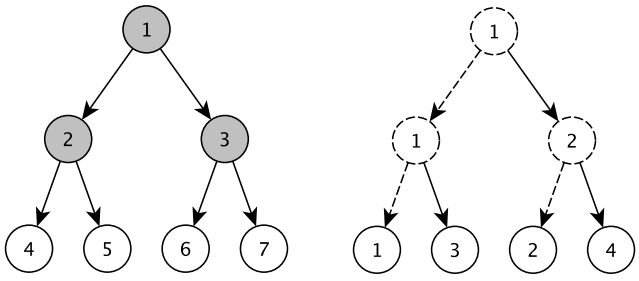
Figur 4.2: Resultat: Andre algoritme (F2: Maks dybde)

ritmen er opp til 101 ganger tregere enn den sekvensielle for array-er med mindre enn 10 000 elementer. Etter dette ser vi en jevn forbedring i ytelse når antall elementer stiger. Ved 20 millioner elementer er algoritmen ca. 3,3 ganger raskere enn referanseimplementasjonen. Dette er en tydelig forbedring fra vår første algoritme, og ikke så langt unna vår teoretiske speedup på 4 (antall kjerner i testmaskinen).

4.2.3 Tredje algoritme (F3: Gjenbruk tråd)

Ide: La det ene rekursive kallet fortsette i samme tråd.

I forrige algoritme oppnådde vi en tydelig forbedring over referansealgoritmen. Allikevel ser vi rom for forbedring. Figur 4.3a viser trådene i den rekkefølgen de opprettes i forrige algoritme. De grå trådene står kun og venter på at trådene under skal bli ferdig.



(a) Før endring (b) Etter endring

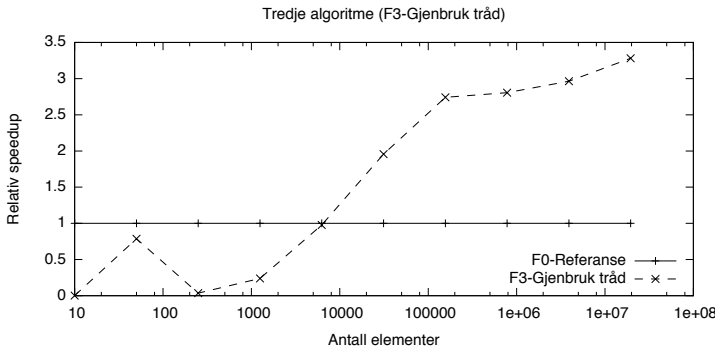
Figur 4.3: Tråder som startes før og etter denne endringen

Vår nye ide er å flette den ene halvdelen av array-et i tråden som kjører, istedenfor å lage en ny. Dermed vil vi bare starte halvparten så mange tråder. Figur 4.3b illustrerer rekkefølgen trådene nå vil startes i. Stiplede linjer illustrerer at tråden har blitt gjenbrukt. Implementasjonen av denne ideen er vist i algoritme 4.3 på neste side.

Resultat: Figur 4.4 på neste side viser resultatet av denne forbedringen. Vi ser at endringen ikke førte til noen høyere speedup, men ytelsen for mindre array-er er noe forbedret. Nå er algoritmen raskere enn referansealgoritmen allerede ved ca. 6000 elementer.

Algoritme 4.3 Parallell flettesortering: Tredje forsøk (F3: Gjenbruk tråd)

```
  ⋮
Thread t1 = new Thread(new MergesortThread(dst, src, start, mid, d+1,
    d_lim));
t1.start();

mergesort(dst, src, mid, stop, d+1);
try {
  t1.join();
} catch (InterruptedException e) {
  e.printStackTrace();
  System.exit(1);
}
  ⋮
```



Figur 4.4: Resultat: Tredje algoritme (F3: Gjenbruk tråd)

4.2.4 Fjerde algoritme (F4: Parallell fletting)

Ide: Utføre flettingen i parallell

Frem til nå har alle algoritmene vi har sett på hatt én stor begrensning. Selve flettingen av de to halvdelene skjer sekvensielt. Dette betyr at uansett hvor mange kjerner som er tilgjengelig, vil algoritmen i siste steg alltid kjøre én tråd som fletter de to største halvdelene. Dette er relativt tidkrevende, og det burde være en del tid å hente på å utføre flettingen i parallell.

Som vi så i kapittel 2, demonstrerer [Satish et al., 2008] hvordan man kan dele opp array-ene som skal flettes i flere uavhengige biter. Kort fortalt velger man et element e i det første array-et. Deretter finner man tilsvarende grense i det andre array-et, ved hjelp av binærsøk. Siden vi vet størrelsen på alle bitene, kan nå første og andre halvdel av array-et flettes uavhengig av hverandre. Figur 2.6 på side 12 illustrerer dette prinsippet.

I implementasjonen vår har vi laget en generalisert versjon av metoden beskrevet over. Istedenfor å dele array-ene i to biter, deler vi dem i p biter (hvor p er antall kjerner delt på antall tråder som kjører). Algoritme 4.4 på side 38 viser hvordan vi har implementert dette. Vår implementasjon av binærsøk (*lowerBound()*) har vi lånt fra STL-biblioteket for å spare tid.

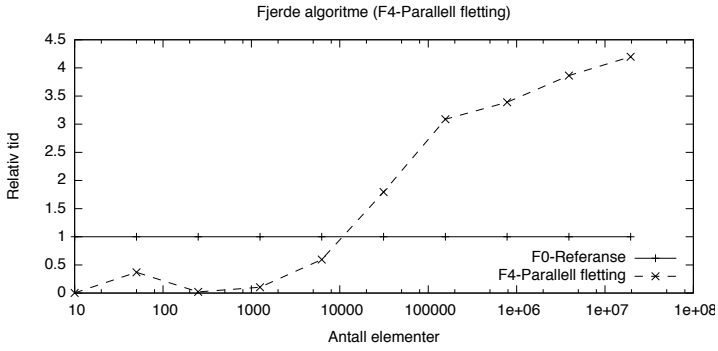
Siden vi nå fletter flere deler i parallell, må vi flytte denne koden ut i en egen tråd. Dette har vi gjort ved å lage en ny klasse som implementerer *Runnable* grensesnittet. Koden for å flette er også endret, siden vi nå kun fletter deler av arrayet i hver tråd. Implementasjonen vår av denne klassen er vist i program 4.5.

Program 4.5 Tråd for å flette to *stykker* av array-et

```
private class MergeThread implements Runnable {
  ⋮
  public void run() {
    int p = start1;
    int q = start2;

    int len = (stop1-start1) + (stop2 - start2);

    for(int i=0; i<len; i++) {
      if(q >= stop2 || (p < stop1 && src[p] < src[q]))
        dst[dstOffset+i] = src[p++];
      else
        dst[dstOffset+i] = src[q++];
    }
  }
}
```

Legg merke til at vår implementasjon av parallell fletting er noe forenklet. Istedenfor å gjenbruke tråder som allerede kjører, venter vi på at trådene som sorterer parallelt avsluttes før vi starter nye tråder for å flette parallelt. Dette har vi valgt fordi vi først og fremst ville teste effekten av å sortere parallelt. Vi har alt sett i forrige algoritme at det å halvere antall tråder ikke gir noen synlig effekt på sluttresultatet, men at det vil påvirke ytelsen for array-er med ca. 5 000–500 000 elementer. Vi registrerer derfor at her fins et lite potensiale for forbedring, dersom implementasjonen ellers har god ytelse.

Resultat: Figur 4.5 på neste side viser resultatet vi fikk. Ved å parallellisere flettingen,



Figur 4.5: Resultat: Fjerde algoritme (F4: Parallell fletting)

ser vi at den maksimale speedupen har økt fra ca. 3,3 til 4,2. Dette resultatet er vi godt fornøyd med. Men vi legger også merke til at denne endringen var et mye større inngrep i den opprinnelige algoritmen enn de tidligere endringene. Som algoritme 4.4 på neste side viser, var implementasjonen av funksjonaliteten ganske komplisert. Om vi også skulle redusere antall tråder som startes, ville den blitt enda mer omfattende.

Algoritme 4.4 Parallell flettesortering: Fjerde forsøk (F4: Parallell fletting)

```
  :
  // Calculate number of processors available to merge list
  int parts = Runtime.getRuntime().availableProcessors() >> d;

  if(parts > 1) {
    // Calculate start position for all slices in left half
    int[] mStart1 = new int[parts+1];
    for(int i=0; i<parts; i++)
      mStart1[i] = start + (mid - start) * i / parts;
    mStart1[parts] = mid;

    // Calculate corresponding start positions for slices in right
    // half using binary search
    int[] mStart2 = new int[parts+1];
    mStart2[0] = mid;
    for(int i=1; i<parts; i++)
      mStart2[i] = lowerBound(src, mid, stop, src[mStart1[i]]);
    mStart2[parts] = stop;

    // Create threads for merging slices into target array
    int offset = 0;
    MergeThread[] threads = new MergeThread[parts];
    for(int i=0; i<parts; i++) {
      int start1 = mStart1[i], stop1 = mStart1[i+1];
      int start2 = mStart2[i], stop2 = mStart2[i+1];

      threads[i] = new MergeThread(dst, src, start1, stop1, start2, stop2,
          start+ offset);
      offset += (stop1 - start1) + (stop2 - start2);
    }

    // Start merging-threads
    Thread[] tt = new Thread[parts];
    for(int i=1; i<parts; i++) {
      tt[i] = new Thread(threads[i]);
      tt[i].start();
    }
    threads[0].run();

    // Wait for all threads to finish
    for(int i=1; i<parts; i++)
      try {
        tt[i].join();
      } catch (InterruptedException e) {
        // Handle exception
      }
  }
  :
```

4.3 Venstre-radix-sortering

4.3.1 Første algoritme (R1: Naiv)

Ide: Start alle rekursive kall i en ny tråd.

I vårt første forsøk på å parallellisere venstre-radix-algoritmen, starter vi med samme ide som flettesorteringen. Vi erstatter alle rekursive kall med kode som starter en ny tråd. Vi har alt lært fra flettesorteringen at dette sannsynligvis ikke er en lur strategi, men vi er interessert i å se hvordan en annen algoritme vil oppføre seg med samme strategi.

I likhet med første algoritme for parallell flettesortering (beskrevet på side 32), startet vi også her med å flytte den rekursive metoden inn i en privat indre klasse. Deretter erstattet vi de rekursive kallene med å starte nye tråder. Implementasjonen vår er vist i algoritme 4.5.

Algoritme 4.5 Parallell venstre-radix-sortering: Første forsøk (R1: Naiv)

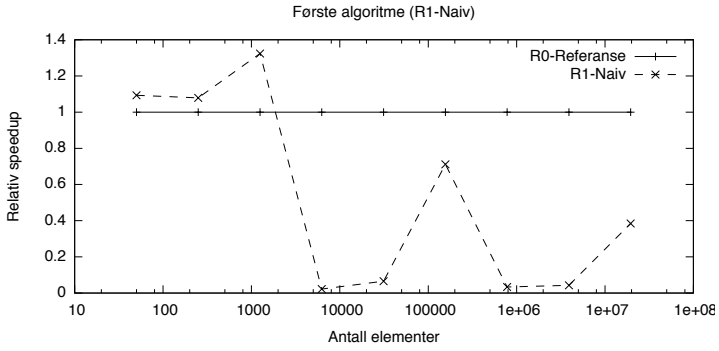
```
:
// call each cell if more than one number
if (newNum > 1) {
  if ( newNum <= INSERT_MAX ) {
    insertSort (a,t1,t2-1);
  } else {
    Thread t = new Thread(new ARLThread(a, t1, t2-1, leftBitNo));
    t.start();
    threads.add(t);
  }
} // if newNum > 1
:
for(Thread t: threads) {
  try {
    t.join();
  } catch (InterruptedException e) {
    // Handle exception
  }
}
:
```

Resultat: Figur 4.6 på neste side viser ytelsen til den parallelle algoritmen. Resultatene viser at algoritmen er opp til 45 ganger tregere enn referansealgoritmen, men jevnt over presterer den mye bedre enn den første flettealgoritmen.

Vi ser at denne algoritmen klarer å kjøre alle datasettene, helt opp til 20 millioner elementer. Trolig skyldes dette at rekursjonstreet er mye bredere og grunnere. På hvert nivå starter vi opp til 256 tråder, men vi har maksimalt fire nivåer i treet. Dermed vil trådene i nederste nivå avslutte mye tidligere, og vi får langt færre tråder som kjører samtidig.

4.3.2 Andre algoritme (R2: Ett nivå)

Ide: Kun starte tråder på første nivå i rekursjonstreet.

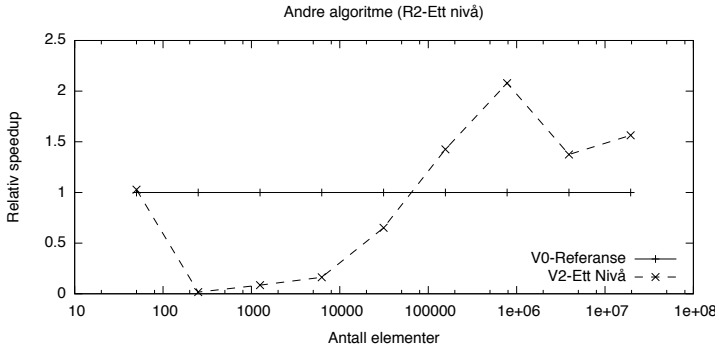


Figur 4.6: Første parallelle algoritme

I andre flettesorteringsalgoritme (beskrevet på side 33) så vi at det var effektivt å begrense start av nye tråder til en viss dybde i rekursjonstreet. I denne algoritmen er rekursjonstreet mye bredere og grunnere. Vi forsøker derfor å kun starte nye tråder i første nivå av treet.

Implementasjonen vår er vist i algoritme 4.6 på neste side. Her ser vi at vi har trukket ut en egen metode som gjør første fordeling i bøtter, og starter en tråd per bøtte. For å starte et passelig antall tråder, justerer vi maskestørrelsen som benyttes dynamisk i forhold til antall kjerner. Siden størrelsen på bøttene er ukjent, er det vanskeligere å fordele arbeidsmengden jevnt over kjernene. Derfor sørger vi for at antall bøtter er noe større enn antall kjerner, slik at vi får flere og mindre tråder. Vi ender opp med å beregne maskestørrelsen etter formelen $m = 4 \cdot \log k$, hvor k er antall kjerner.

Resultat: Figur 4.7 viser resultatet av denne endringen. Vi ser at algoritmen er raskere enn referanseimplementasjonen ved ca. 80 000 elementer. På sitt beste er den ca. dobbelt så rask, men deretter synker ytelsen noe.



Figur 4.7: Andre parallelle algoritme

4.3.3 Tredje algoritme (R3: Trådsamling)

Ide: Starte færre tråder ved å bruke en trådsamling.

Når vi ser nærmere på forrige algoritme, legger vi merke til at denne kan egne seg godt til å benytte en trådsamling. Alle trådene startes i samme nivå, og alle trådene er helt

Algoritme 4.6 Parallell venstre-radix-sortering: Andre forsøk (R2: Ett nivå)

```
public class ParallelLeftRadixSort2 extends SortAlgorithm {
  public void sort(int[] array) {
    :
    sortARLwithBorders(array, 0, array.length - 1, leftBitNo);
  }

  private void sortARLwithBorders(int a[], int start, int end, int
      leftBitNo) {
    :
    ArrayList<Thread> threads = new ArrayList<Thread>();
    :
    if (newNum > 1) {
      if (newNum <= INSERT_MAX) {
        insertSort(a, t1, t2 - 1);
      } else {
        Thread t = new Thread(new ARLThread(a, t1, t2 - 1, leftBitNo));
        t.start();
        threads.add(t);
      }
    } // if newNum > 1
    :
    for (Thread t : threads)
      try {
        t.join();
      } catch (InterruptedException e) {
        // Handle exception
      }
  }// end sortARLwithBorders

  private class ARLThread implements Runnable {
    // Sequential Adaptive Radix Left implementation
  }
}
```

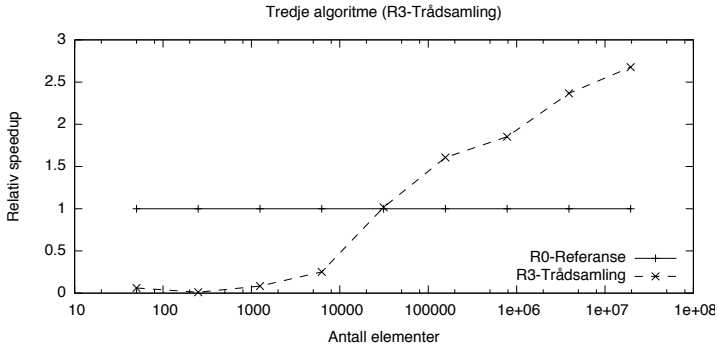
uavhengige. Ved å benytte en trådsamling kan vi begrense antall tråder som startes. Dette vil også føre til at færre tråder kjører samtidig, og forhåpentligvis skape mindre konkurranse om ressursene.

Algoritme 4.7 viser hvordan vi har implementert denne endringen. Først lager vi en trådsamling med like mange tråder som vi har tilgjengelige kjerner. Deretter lager vi en *oppgave* for hver bøtte, og kjører disse i trådsamlingen. Og til slutt venter vi på at alle oppgavene skal bli ferdig.

Algoritme 4.7 Parallell venstre-radix-sortering: Tredje forsøk (R3: Trådsamling)

```
...
ExecutorService executor = Executors.newFixedThreadPool(Runtime.
    getRuntime().availableProcessors());
...
// call each cell if more than one number
if (newNum > 1) {
  if (newNum <= INSERT_MAX)
    insertSort(a, t1, t2 - 1);
  else
    executor.execute(new ARLThread(a, t1, t2 - 1, leftBitNo));
} // if newNum > 1
...
executor.shutdown();
try {
  executor.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);
} catch (InterruptedException e) {
  // Handle exception
}
...
```

Resultat: Figur 4.8 viser resultatet av denne endringen. Hvis vi sammenligner denne grafen med resultatet for forrige algoritme (figur 4.7 på side 40), ser vi en interessant effekt. Vi ser i området hvor forrige algoritme sank i ytelse, fortsetter denne å stige. Totalt oppnår vi en speedup på ca. 2,7X.



Figur 4.8: Tredje parallelle algoritme

Kapittel 5

Flerkjerneprogrammering på GPU

Når man skal designe algoritmer for GPU er det viktig å ha en god forståelse av hvordan moderne skjermkort er strukturert. Siden skjermkort i utgangspunktet er optimalisert for å tegne grafikk, er de bygget opp svært annerledes enn en vanlig CPU og man må derfor bruke andre teknikker for å oppnå god ytelse.

I kapittel 5.1 vil vi se nærmere på NVIDIA sine skjermkort. Vi ser hvordan de er strukturert, og hva man trenger å vite for å kunne utnytte dem effektivt. Deretter vil vi gi en kort introduksjon til programmering i CUDA (NVIDIA sitt rammeverk for GPU-programmering).

Så vil vi i kapittel 5.2 og 5.3 se på våre egne algoritmer vi har laget for GPU og resultatene vi fikk fra disse. Nærmere diskusjon av disse resultatene vil vi gå igjennom i kapittel 6.

5.1 Parallell programmering med CUDA

5.1.1 Terminologi

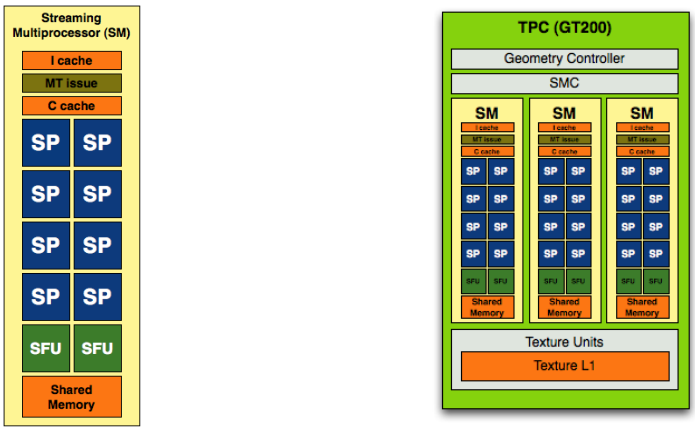
Når man programmerer for GPU, er det viktig å skille mellom *vert* og *enhet*. Enhet refererer til skjermkortet, mens vert refererer til resten av datamaskinen. Koden som kjører på CPU kaller vi for *vertskode* mens kode som kjører på skjermkortet kalles *enhetskode*. Enhetskode samles i *kjerner*. Hver kjerne må lastes over til skjermkortet før den kan aktiveres av vertskoden. Tråder som kjører på GPU-en blir gruppert i grupper på 32. En slik gruppe kalles en *varp*.

5.1.2 Arkitektur

Den største forskjellen mellom GPU-programmering og vanlig CPU-programmering, er at at GPU-en er laget for å gjøre massivt parallelle beregninger. Derfor er GPU-en organisert i en Single Instruction Multiple Threads-arkitektur (SIMT-arkitektur). Kort fortalt betyr dette at alle tråder innenfor en *varp* vil til enhver tid kjøre samme instruksjon.

Ved å bruker tråder som abstraksjon, blir SIMT-arkitekturen mye enklere å programmere for enn vanlig SIMD. Utvikleren skriver tråder som arbeider med ett og ett dataelement, og maskinvaren vil samle disse i grupper på 32 som kjøres samtidig. Dersom trådene i en

varp kommer til en forgrening (f.eks. if-test, for-løkke el.l.) vil maskinvaren automatisk håndtere dette. Hvis alle trådene velger samme gren vil kun denne kjøres. Skulle derimot noen av trådene velge den andre grenen, må disse vente til første grenen er utført. Deretter må alle trådene som valgte første gren vente, mens andre gren utføres. For å oppnå god ytelse er det derfor viktig å passe på at trådene innenfor en varp i størst mulig grad unngår å velge forskjellige grener.



(a) Streaming Multiprocessor

(b) Texture/Processor Cluster

Figur 5.1: Byggesteiner: NVIDIA GT200-arkitektur [Shimpi og Wilson, 2008]

Figur 5.1a viser en oversikt over en *multiprosessor* (SM). Dette er "byggestenen" i alle moderne NVIDIA-kort. De kan ha alt fra 1 SM (lavbudsjett) til 30 SM-er (dyre modeller). Hver SM har 8 *Skalar Prosessorer* (SP-er) som kjører trådene. Ofte refererer man til en SP som en *CUDA-kjerne*. Hver av disse kjernene opererer på dobbel frekvens av SM-en, og kjører derfor 16 tråder (en *halv-varp*) per klokkesyklus. All håndtering av tråder skjer i maskinvaren med tilnærmet null ekstra kostnad. Derfor kan GPU-en håndtere millioner av tråder samtidig, og oppnå svært god ytelse.

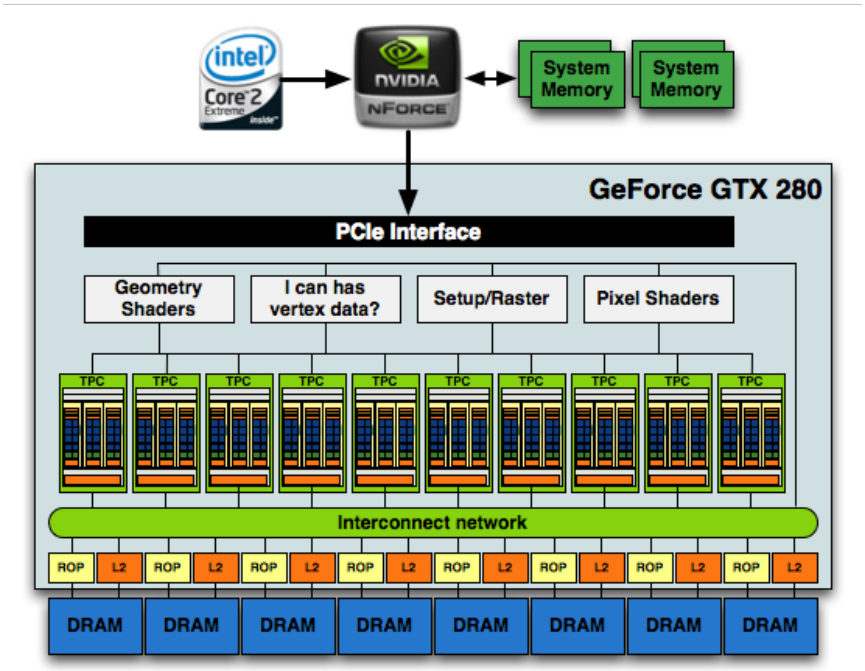
Tar vi ett steg utover, ser vi i figur 5.1b at tre og tre SM-er er gruppert i en *Texture/Processor Cluster* (TPC). Disse inkluderer noe ekstra logikk for å håndtere teksturer. Figur 5.2 på neste side viser en oversikt over hele GT200 arkitekturen. Vi ser at skjermkortet har 10 TPC-blokker, som gir en total av 30 SM-er og 240 CUDA-kjerner. Vi ser også at kortet har åtte separate minnekontrollere.

5.1.3 Minne

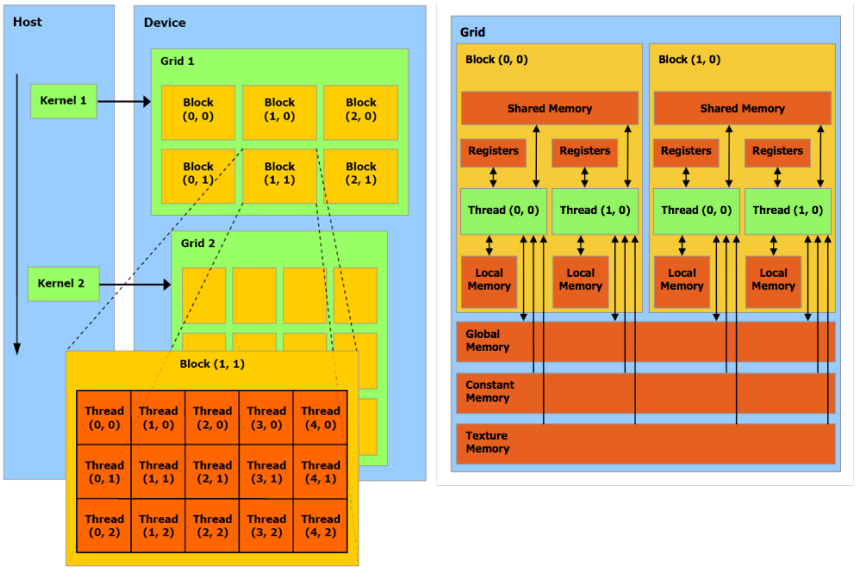
Skjermkortet er delt opp i flere typer minne. For å oppnå god ytelse er det viktig å forstå hver type minne, og hva de brukes til. Figur 5.3b på neste side gir en generell oversikt over hvilke typer som finnes, og hvilken synlighet de har for trådene. Under vil vi kort gå igjennom hver type:

Globalt minne

Dette er hovedminnet på skjermkortet. Kjerner som kjører på GPU-en kan kun se dette minnet, og har ikke direkte tilgang til vertsminnet. Programmer som ønsker å utføre



Figur 5.2: Oversikt over NVIDIA GT200-arkitektur [Shimpi og Wilson, 2008]



(a) Tråd-gruppering (b) Minne-modell

Figur 5.3: Organisering av tråder og blokker [NVIDIA, 2007, side 9 og 11]

beregninger på GPU må derfor alltid kopiere dataene til skjermkortet før beregningen kan starte. Når beregningen er ferdig, må dataene kopieres tilbake.

I nyere skjermkort har man ett unntak til denne regelen. Ved å bruke *page-locked memory* kan man låse deler av minnet i RAM på verten. Med CUDA 2.2 kan man da mappe dette minnet direkte inn i adresserommet på GPU-en. Dermed slipper man å kopiere data til/fra GPU-en, men blir begrenset til båndbredden tilgjengelig over PCIe (som er betydelig lavere enn båndbredden til skjermkortminnet).

Hver gang en tråd leser eller skriver til det globale minnet, tar det ca. 400–600 klokkesyklus å fullføre operasjonen [NVIDIA, 2007, side 49]. For å skjule denne forsinkelsen, har NVIDIA laget en svært effektiv trådveksler. Når trådene i en varp aksesserer det globale minnet, sendes en forespørsel etter de ønskede adressene til minnekontrolleren. Samtidig legges varpen i en kø av tråder som venter på data fra minnet. Deretter fortsetter multiprosessoren å kjøre andre tråder. Når dataene er tilgjengelige fra minnet, oppdateres registrene i varpen som ba om dem og varpen flyttes til en kø av tråder som er klar for å kjøre igjen. Dersom antall tråder på SM-en er stort nok, klarer den tilnærmet fullstendig å skjule forsinkelsen til det globale minnet.

Delt minne

På GPU blir ikke tilgang til det globale minnet cachet. Istedenfor har hver SM 16KB delt minne hvor trådene selv kan cache data dersom det er behov for det. Som vi ser i figur 5.3b på forrige side, har alle trådene innenfor samme blokk tilgang til det samme delte minnet. Hvor mye delt minne hver blokk bruker, setter derfor en begrensning på hvor mange blokker SM-en kan håndtere samtidig.

Tilgang til delt minne er like raskt som tilgang til registre, forutsatt at ikke man har *bankkonflikter*. Siden SM-en kjører 16 tråder (en halv-varp) samtidig, er det delte minnet delt opp i 16 like store moduler kalt *banker*. Dersom hver tråd skriver til hver sin bank kan alle verdiene skrives samtidig. Skulle derimot n tråder skrive til forskjellige adresser i samme bank, må disse skrives etter hverandre. Det vil derfor ta n ganger så lang tid.

Konstant minne

Konstant minne er en del av det globale minnet reservert for verdier som ikke endrer seg. Som vi ser i figur 5.3b på forrige side har alle tråder tilgang til dette minnet. Siden verdiene i dette minnet ikke endrer seg, har hver SM et lite cache-minne for å akselerere tilgangen til disse verdiene (se figur 5.1a på side 44).

Lokalt minne

Lokalt minne er også en del av det globale minnet, og brukes for å lagre lokale variable i trådene som er for store til å få plass i registrene. Dette kan for eksempel være arrayer eller strukt-er. Tilgang til det lokale minnet er like tregt som tilgang til det globale minnet. Fordelen med lokalt minne er at dataene organiseres slik at når alle trådene aksesserer samme variabel, vil operasjonen være fullstendig *koalesert*. Hva dette betyr vil vi beskrive nærmere i avsnitt 5.1.4 på side 49.

Tekstur minne

Tekstur minnet er en del av det globale minnet som er optimalisert for lesing. Dataene som leses fra tekstur minnet blir cachet i TPC-en (se figur 5.1b på side 44). Dette cache-minnet er dog ikke garantert å oppdateres når dataene endrer seg. Det er derfor kun i spesifikke situasjoner det egner seg å benytte teksturminnet.

Registere

Hver SM har til sammen 16 384 registre. Disse registrene deles mellom trådblokkene som kjører på SM-en, og brukes til å holde de lokale variablene i hver enkelt tråd. Hvor mange registre man bruker per tråd påvirker derfor hvor mange trådblokker SM-en kan kjøre samtidig. Hver tråd kan derfor maksimalt bruke 16 registre, dersom man ønsker å utnytte hver SM fullstendig.

5.1.4 Programmering

Når man skal programmere på et NVIDIA-kort, skjer dette gjennom et rammeverk som heter CUDA. Dette er en full pakke med kompilator, debugger (foreløpig bare for Linux) og andre verktøy for å utvikle programmer som benytter seg av GPU-en. I dette rammeverket finnes det to tilnærminger for å programmere mot skjermkortet:

1. *C for CUDA*: Dette er en tilpasning av programmeringsspråket C med ekstra syntaks for å lage og kjøre kjerner på skjermkortet. For å kompilere programmer skrevet i C for CUDA må man benytte NVIDIA sin egen kompilator, nvcc, som er en del av CUDA-Toolkit-pakken.
2. *CUDA driver API*: Dette er et programmeringsgrensesnitt til CUDA-driveren som gir full tilgang til skjermkortet. Dette grensesnittet kan brukes fra en hvilken som helst C kompilator, og tilbyr mer fleksibilitet enn C for CUDA. Derimot er driver-API-et mer komplisert å sette seg inn i og bruke, og mer vanskelig å vedlikeholde.

I denne oppgaven har vi valgt å benytte oss av den første metoden, C for CUDA. Vi ser ikke at vi har behov for den finkornede kontrollen som driver-API-et gir, og velger derfor å bruke metoden som er enklest og raskest å komme i gang med.

En enkel kjerne

For å illustrere hvordan man skriver kjerner som kjører på GPU vil vi starte med et enkelt eksempel på en slik. Figur 5.1 viser en kjerne som adderer to vektorer og lagrer resultatet i en tredje.

Algoritme 5.1 Enkel CUDA-kjerne

```
__global__ void VecAdd(int A[], int B[], int C[]) {
  int n = blockDim.x * blockIdx.x + threadIdx.x;

  C[n] = A[n] + B[n];
}
```

Vi ser at dette ligner en vanlig funksjon i C, men med noen unntak. For det første bruker man nøkkelordet `__global__` for å angi at dette er en kjerne som skal kjøres på GPU. Vi ser kjernen tar tre parametere. Disse angir vi som vanlig når vi kaller kjernen fra C-koden, og kompilatoren vil sørge for at disse blir overført til skjermkortet.

Det viktigste å legge merke til i eksempelet er SIMT-tankegangen. Vi tenker oss at hver tråd kun leser *ett* element fra hver vektor, adderer disse elementene og skriver resultatet tilbake til den tredje. Vi starter med andre ord én tråd per element. For å vite hvilke tråd man er i, bruker man variablene *blockIdx*,*blockDim* og *threadIdx*. De to første angir henholdsvis hvilke trådblokk man er i, og hvor stor hver blokk er. Den siste variabelen angir hvilken tråd innenfor blokken man er i. Disse variablene er en del av C for CUDA og fylles inn av kompilatoren, selv om de ikke er definert noe sted.

Trådene i en blokk organiseres i en, to eller tre dimensjoner. Hva man velger å bruke avhenger av dataene man skal jobbe med. Størrelsen på blokkene kan være opp til 512x512x64 i henholdsvis x-, y- og z-dimensjonen. Samtidig kan hver blokk ha maksimalt 512 tråder ($x \cdot y \cdot z \leq 512$). Blokkene organiseres igjen i grids, som også kan ha opp til tre dimensjoner. Hver grid kan bestå av opp til 65 536x65 536x1 blokker. Figur 5.3a på side 45 illustrerer denne sammenhengen mellom tråder, blokker og grids.

Program 5.2 viser hvordan kjernen over kan kalles. Her er det spesielt to ting å legge merke til. Først ser vi at før kjernen kan kalles, må alle dataene som kjernen skal jobbe med allokeres og kopieres til skjermkortet. Deretter kommer kallet på selve kjernen. Vi ser at i *C for CUDA* kan en kjerne kalles akkurat som en vanlig funksjon, bortsett fra en liten ekstra syntaks på formen `<<<x,y>>>`. Her er x antall blokker vi skal starte, og y antall tråder per blokk. Disse kan også angis 2- eller 3-dimensjonalt ved å bruke `dim3`-struct-er. Til slutt må vi kopiere resultatet tilbake til verten, før vi frigjør det allokerte minnet på skjermkortet.

Algoritme 5.2 Typisk kall på CUDA-kjerne

```
int main() {
  int* A = arrayWithRandomData(N);
  int* B = arrayWithRandomData(N);
  int* C = (int*)malloc(N*sizeof(int));

  int *d_A, *d_B, *d_C;
  cudaMalloc((void**)&d_A, N*sizeof(int));
  cudaMalloc((void**)&d_B, N*sizeof(int));
  cudaMalloc((void**)&d_C, N*sizeof(int));
  cudaMemcpy(d_A, A, N*sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, A, N*sizeof(int), cudaMemcpyHostToDevice);

  int threadsPerBlock = 512;
  int blocksPerGrid = N / threadsPerBlock;
  VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

  cudaMemcpy(C, d_C, N*sizeof(int), cudaMemcpyDeviceToHost);

  // Use result now stored in C

  cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
  free(A); free(B); free(C);
}
```

Koalesert minneaksess

For å oppnå størst mulig båndbredde mot det globale minnet, forsøker GPU-en å sette sammen forespørsler fra flest mulig tråder innenfor en halv-varp. Dersom alle trådene aksesserer sammenhengende 32-bit verdier, kan dette gjøres i én operasjon. Dette kalles en *koalesert* aksess. Er derimot dataene spredt, må de hentes i separate forespørsler. I verste fall kan derfor ikke-koaleserte aksesser føre til at hver minneoperasjon tar 16 ganger så lang tid.

Rekursjon

En viktig ting å være oppmerksom på når man programmerer med CUDA, er at kjerner som kjører på GPU støtter *ikke* rekursjon. Faktisk blir alle funksjoner man benytter i en kjerne fullstendig *inlinet* i koden. Om man skal løse et problem som er rekursivt definert, slik vi skal i denne oppgaven, må man derfor finne andre måter å formulere problemet før det kan implementeres på GPU. Hvordan vi har løst dette i forhold til våre algoritmer, går vi igjennom i kapittel 5.2 og 5.3.

Automatisk terminering av tråder

En annen viktig begrensning å være klar over, er at skjermkort driveren har en øvre grense for hvor lenge en kjerne kan kjøre på GPU-en. Dersom kjernen bruker lengre tid enn dette, antar driveren at den ikke virker som den skal og terminerer den. Denne grensen er som regel satt til 5 sekunder. Dersom vi skal løse et stort problem er det derfor viktig å dele dette problemet inn i flere mindre kjerner slik at hver kjerne kjører mindre enn 5 sekunder.

5.1.5 Java Native Interface

Siden CUDA er en utvidelse for C, kan vi ikke bruke denne koden i Java-programmene våre direkte. Vi må istedenfor "pakke inn" denne koden slik at den kan kalles fra Java. Til dette formålet finnes Java Native Interface (herifra referert til som JNI).

JNI er en del av Java-standarden, og er designet for at Java programmer skal kunne kalle plattform-avhengig kode. Med andre ord tilbyr JNI større frihet, men mindre fleksibilitet. Man må lage et eget JNI-bibliotek for hver plattform man planlegger å kjøre koden på.

Hvordan bruke JNI

Gjennom de neste avsnittene kommer vi til å gå gjennom alle stegene nødvendig for å kalle kompilert C-kode fra Java. Vi kommer til å lage et lite eksempel program som sorterer et array ved å bruke *STL sort()* i C++.

1. Spesifisere JNI-metoder Det første man må gjøre er å lage en Java klasse som spesifiserer hvilke metoder man ønsker å implementere i C. Program 5.1 på neste side viser hvordan denne Java klassen vil se ut i vårt eksempel program.

Program 5.1 Klasse med en JNI-metode.

```
public class JniSort {
  public native void sort(int[] array);

  static {
    System.loadLibrary("JniSort");
  }
}
```

Vi ser at metoder som skal implementeres i C blir definert på samme måte som abstrakte metoder. Eneste forskjellen er at JNI-metoder deklareres med nøkkelordet *native*.

Når en metode deklareres som *native*, må programmet under kjøring laste JNI-biblioteket som definerer metoden. Dette gjøres med *System.loadLibrary()* kommandoen. Som vi ser i koden, må dette gjøres samtidig som klassen lastes inn i JVM-en. Kallet er derfor plassert i en *static* blokk.

2. Generere header-filer Når Java klassen er ferdig, må man for hver klasse som benytter JNI generere en header-fil med C-definisjoner av metodene som skal implementeres. Denne header-filen lages av *javah* programmet, som er en del av Java Development Kit (JDK).

Før man kan kjøre *javah*, må man kompilere kildekoden til bytekode. Deretter kjører man *javah* med fullt klassenavn som parameter. Her er et eksempel på en slik kjøring:

```
> javac JniSort.java
> javah JniSort
```

3. Implementere metodene i C Når header-filen er generert, er det tid for å implementere *native* metodene i C. Program 5.2 viser en implementasjon av *sort()*-metoden vår. Under vil vi gå kjapt gjennom detaljene i implementasjonen.

Program 5.2 C-implementasjon av en JNI-metode

```
1  #include <jni.h>
2  #include <algorithm>
3  #include "JniSort.h"
4
5  JNIEXPORT void JNICALL
6  Java_JniSort_sort (JNIEnv *env, jobject obj, jintArray array) {
7    jsize len = env->GetArrayLength(array);
8    jint *body = env->GetIntArrayElements(array, 0);
9
10   std::sort(body, body+len);
11
12   env->ReleaseIntArrayElements(array, body, 0);
13   return;
14 }
```

I linje 1 og 3 ser vi at vi må inkludere både *jni.h* og vår egen genererte header-fil. Deretter ser vi en ganske komplisert metode-definisjon. Denne kan heldigvis bare kopieres rett ut ifra den genererte header-filen. Det eneste vi må gjøre er å legge til navn på parameterne.

Så kommer den interessante biten. Siden array-er i Java er første ordens objekter, må vi bruke JNI-metoder for å hente ut størrelse på array-et (linje 7) og peker til elementene (linje 8). Under normal kjøring av et Java program kan garbage collector-en flytte dataene i array-et til andre deler av minnet. Ved å bruke *GetIntArrayElements()* vil JNI enten markere array-et så det ikke kan flyttes, eller kopiere det til en egen del av minnet som ikke er flyttbart.

Etter vi har fått en peker til dataene, så kan vi sortere elementene i listen (linje 10). Til slutt må vi frigi array-et ved å kalle *ReleaseIntArrayElements()*, slik at JNI kan kopiere dataene tilbake eller fjerne markeringen som låser det fast i minnet.

4. Kompilere C bibliotek Til slutt må man kompilere C programmet til et bibliotek som Java kan laste. Nøyaktig hvordan man kompilerer avhenger av hvilken kompilator og plattform man bruker. Her er et eksempel som virker på Windows med Visual C++:

```
> cl -o JniSort.dll JniSort.cpp /I <path-til-JDK>\include ↩
     /I <path-til-jdk>\include\win32 /link /DLL
```

Felles for alle plattformer er at man må kompilere til en biblioteksfil (og ikke et vanlig program). Navnet biblioteksfilen må ha for at Java skal finne den avhenger av plattformen. Tabell 5.1 viser en kort oversikt over disse navnene.

Windows:	JniSort.dll
Linux:	libJniSort.so
Mac OS:	libJniSort.jnilib

Tabell 5.1: Navngivning av biblioteksfiler

5. Kjøre Java-koden Nå som JNI-biblioteket er kompilert, kan man kjøre Java-koden som bruker den. For at Java skal finne biblioteksfilen, er det viktig at vi angir hvilken katalog den ligger i. En måte å gjøre dette på, er å angi katalogen i en miljøvariabel (`%Path%` i Windows, eller `LD_LIBRARY_PATH` i Linux/Mac). Alternativt kan vi gi katalogen direkte til Java, som vist her:

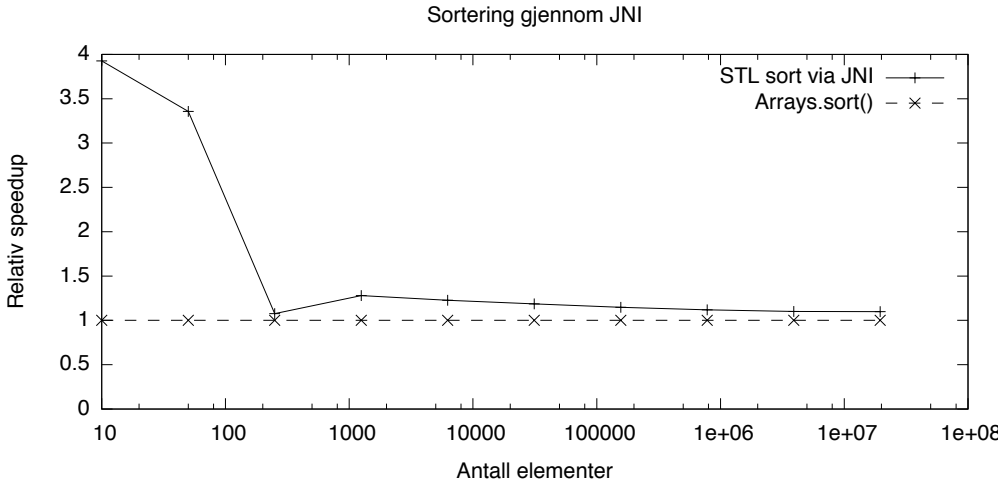
```
> java -Djava.library.path="." MyClass
```

Ytelse

For å demonstrere ytelsen til JNI-kall i forhold til vanlige Java kall, har vi sammenlignet kjøretiden til *JniSort* klassen i forrige avsnitt med Java sin innebygde *Arrays.sort()*. Resultatet av sammenligningen er vist i figur 5.4 på neste side.

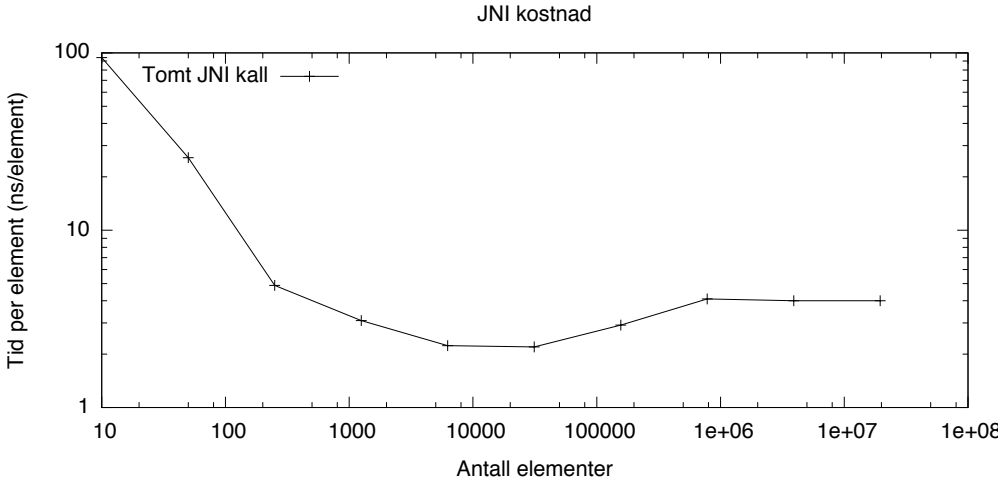
Vi ser at C++ sin standard sorteringsalgoritme demonstrerer ganske bra ytelse i forhold til Java. For array-er med mindre enn 100 elementer er den opp til 4 ganger så rask som Java sin innbygde sortering. Ved større array-er er den jevnt over 10 til 20 prosent raskere.

Det er noe overraskende at sortering via JNI er så mye raskere for små array-er siden det er en relativt stor kostnad å kalle *native*-metoder fremfor vanlige Java-metoder. For å undersøke dette nærmere har vi laget noen ekstra tester for å avdekke hvor mye stor denne kostnaden er.



Figur 5.4: Sammenligning: *STL sort()* mot *Arrays.sort()*

Kostnad ved å kalle JNI I vår første test vil vi undersøke hvor mye ekstra tid som går med ved å gjøre et kall til en JNI-metode. For å teste dette har vi laget et tomt JNI-kall ved å fjerne sorteringskallet i JniSort (se linje (10) i program 5.2 på side 50). Resultatet er vist i figur 5.5.

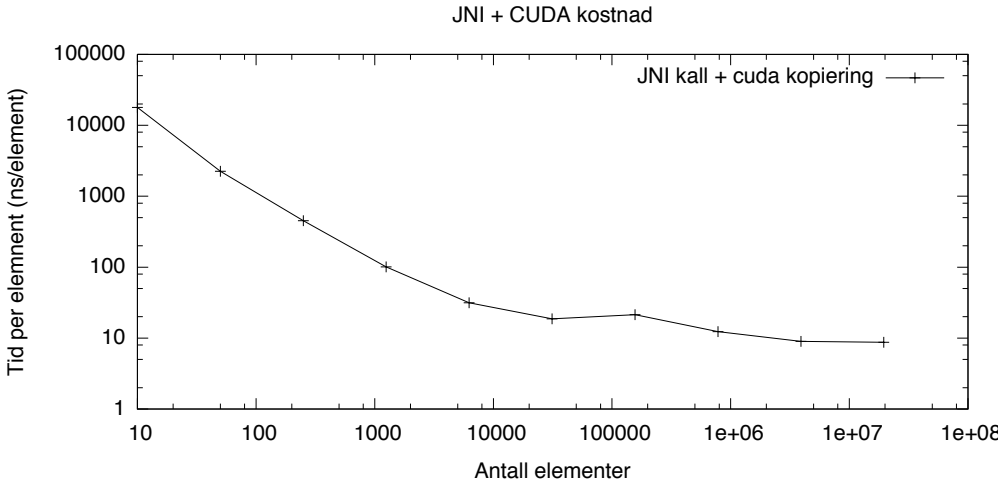


Figur 5.5: Kostnad ved å kalle JNI med array av forskjellige størrelser

Vi ser at for array-er mindre enn 1000 elementer er kostnaden relativt stor (opp til 50 nanosekunder per element). Men etterhvert som størrelsen på array-ene vokser, jevner den seg ut på ca. 2 til 4 nanosekunder per element.

Hvis vi ser på disse tallene i forhold til tidene i figur 5.4, ser vi at for 10 elementer går 80 prosent av tiden med til selve JNI-kallet, mens bare 20 prosent går med til sortering. Vi må derfor konkludere med at *STL sort()* er raskere, til tross for at mesteparten av tiden går med til selve JNI-kallet.

Kostnad ved å kalle JNI + CUDA Siden CUDA krever at man kopierer array-et som skal sorteres videre fra C til skjermkortet, er det også interessant for oss å vite hvor stor ekstra kostnad dette medfører. Vi har derfor utvidet testprogrammet fra 5.1.5 på forrige side til å kopiere data til skjermkortet og tilbake. Resultatene er vist i figur 5.6.



Figur 5.6: Kostnad ved å kopiere data til C og videre til skjermkort

Her ser vi i likhet med forrige test en ekstra høy kostnad ved array-er mindre enn 1000 elementer (opp til ca. 18 000 nanosekunder per element). Mens for større array-er jevner kostnaden seg ut på ca. 10 til 20 nanosekunder per element.

5.2 Flettesortering

Den første algoritmen vi implementerte på GPU var flettesortering. Siden vi ikke tidligere har programmert for GPU, var fokuset først og fremst å programmere for korrekthet. Når vi har en fungerende algoritme, ønsker vi å se nærmere på hvilke grep som kan gjøres for å tilpasse den til GPU-arkitekturen.

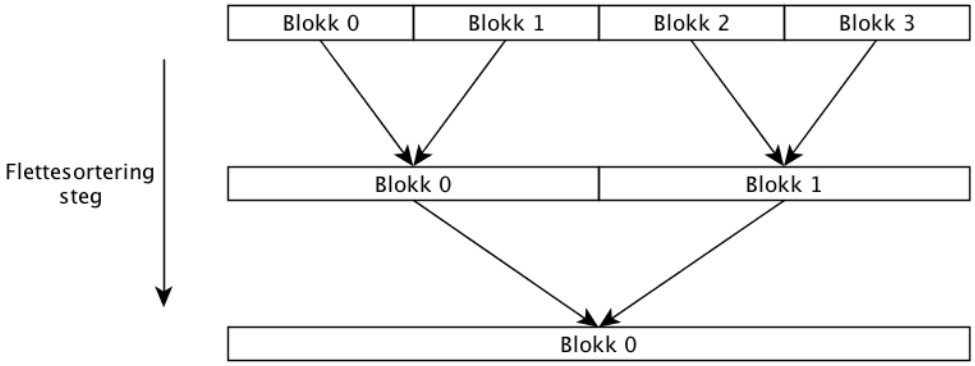
5.2.1 Første algoritme (F1: GPU)

Som vi har sett i kapittel 5.1.4 på side 49, støtter ikke CUDA-kjerner rekursjon. Den første utfordringen vår ble derfor å finne en alternativ måte å implementere flettesorteringen på.

Ide: Flette array-et parallelt med én tråd per 2 blokker.

I figur 5.7 på neste side ser vi stegene i vanlig flettesortering. Vi ser at hvert horisontalt nivå kan gjøres parallelt. Vi skisserer derfor en algoritme som starter med å se på array-et som en samling av n blokker med størrelse 1. Vi starter $n/2$ tråder som fletter to og to blokker, slik at vi får $n/2$ blokker med størrelse 2. I neste nivå starter vi $n/4$ tråder, og slik fortsetter vi til alle elementene er sortert.

I algoritme 5.3 på neste side ser en enkel CUDA-kjerne som implementerer denne algoritmen. Vi beregner først hvor mange tråder vi skal starte. Hver tråd fletter to blokker



Figur 5.7: Flettesortering: parallelle steg

av array-et sammen til en ny sortert blokk. Antall tråder regner vi ut av med formelen $lim = \lceil \frac{len}{rank} \rceil$, hvor *len* er lengden på array-et og *rank* er lengden på resultatblokken. Deretter beregner hver tråd start og stopp indeks for sin del av array-et. Til slutt fletter trådene sine blokker fra *src* til *dst*.

Algoritme 5.3 CUDA-flettekjerne

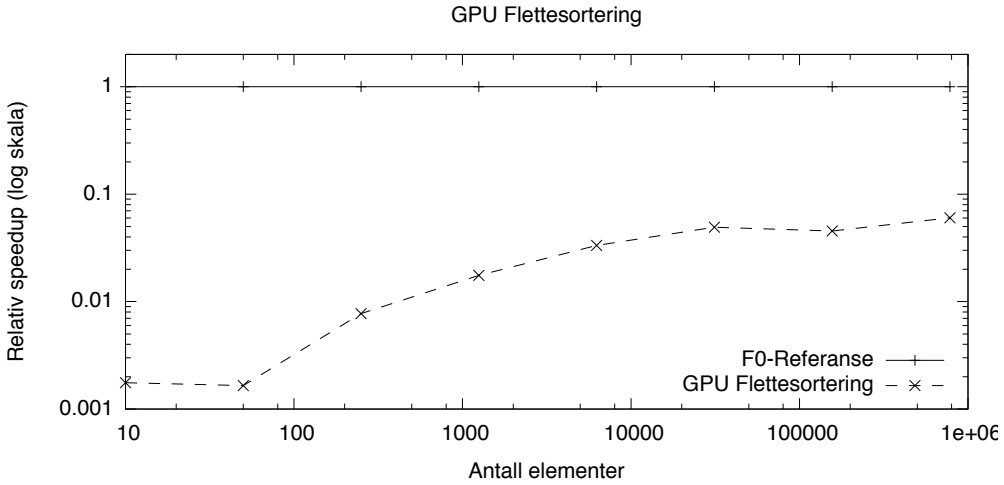
```
__global__ void merge(int* src, int *dst, int rank, int len) {
  int n = blockDim.x * blockIdx.x + threadIdx.x;

  int lim = (len + rank-1) / rank;

  if(n < lim) {
    int start = rank * n;
    int stop = MIN(start + rank, len);
    int mid = start + (rank >> 1);

    for(int i=start, p=start, q=mid; i<stop; ++i) {
      if(q >= stop || p < mid && src[p] < src[q])
        dst[i] = src[p++];
      else
        dst[i] = src[q++];
    }
  }
}
```

Resultat: Når vi skal måle ytelsen til algoritmen, lager vi et JNI-bibliotek slik vi har beskrevet i kapittel 5.1.5 på side 49. Deretter sammenligner vi kjøretiden mot referanse-implementasjonen for flettesortering. Resultatet er vist i figur 5.8 på neste side.

Som vi ser, presterer denne algoritmen svært dårlig. På sitt beste er den ca. 17 ganger tregere enn referanseimplementasjonen, og ved ca. 2 millioner elementer støter vi på 5-sekunders grensen som vi har beskrevet i kapittel 5.1.4 på side 49.

5.2.2 Nærmere analyse

På grunn av mangelfull tid rakk vi ikke å forsøke flere implementasjoner av flettesortering på GPU. Istedenfor har vi forsøkt å avsløre hvilke deler av algoritmen som sviktet, slik



Figur 5.8: Ytelse: flettesortering på GPU

at vi bedre kan forstå hvordan GPU-en virker.

Vi har alt en mistanke om at den synkende graden av parallellitet som kommer når blokkene som skal flettes blir større kan være hovedårsaken. Vi starter derfor med å se på hvor mange tråder som startes i hvert parallelle steg i algoritmen. Tabell 5.2 viser en oversikt over dette.

Steg:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Blokker:	3907	1954	977	489	245	123	62	31	16	8	4	2	1	1	1	1	1	1	1	1	1
Tråder per blokk:	256	256	256	256	256	256	256	256	256	256	256	256	245	123	62	31	16	8	4	2	1

Tabell 5.2: Tråd- og blokk-dimensjoner per steg for 1 million elementer

Her ser vi ekstra tydelig hvordan parallelliteten i algoritmen synker. Samtidig vet vi at når antall tråder synker, blir blokkene som skal flettes større. En mulig løsning på dette problemet kan være å gjøre de siste stegene av algoritmen på CPU istedenfor på GPU-en. Vi testet derfor ut samme algoritme på CPU, og tok tiden på hvert steg. Tabell 5.3 viser disse tidene sammenlignet med tiden på GPU når vi sorterer 1 million tall.

Steg:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
CPU:	7 ms	7 ms	8 ms	7 ms	7 ms	7 ms	6 ms	7 ms	7 ms	7 ms	7 ms	7 ms	6 ms	6 ms	6 ms	7 ms	6 ms	5 ms	5 ms	3 ms
GPU:	1 ms	1 ms	2 ms	3 ms	3 ms	3 ms	3 ms	3 ms	4 ms	6 ms	9 ms	15 ms	23 ms	38 ms	74 ms	126 ms	222 ms	364 ms	616 ms	1001 ms

Tabell 5.3: Tid per steg på CPU i forhold til GPU.

Tabellen viser akkurat det vi mistenkte. Så lenge GPU-en kjører mange tråder som bare håndterer noen få elementer hver er den opp til 7 ganger raskere enn CPU versjonen. Men når det blir færre tråder og større blokker presterer den mye dårligere. Samtidig ser vi at CPU-en blir raskere når blokkene vokser. Dette er fordi større blokker på CPU gir mer cache-vennlig minne-aksess. Vi ser at man ved å gjøre de siste stegene på CPU kun kunne spart noen få millisekunder.

Tabellene over viser tydelig at dersom vi ønsker bedre ytelse på GPU-en må vi finne en måte å parallellisere flettingen av større blokker. [Satish et al., 2008] presenterer én mulig måte å gjøre dette på. I kapittel 6.3.3 på side 64 vil vi gå nærmere inn på denne metoden, og andre ting vi kan gjøre for å forbedre ytelsen til algoritmen.

5.3 Venstre radix sortering

5.3.1 Første algoritme (R1: GPU)

Ide: Sortere 1024 bøtter i parallell.

Når vi skulle implementere venstre-radix-sortering på GPU måtte vi i likhet med flette-sorteringen finne en annen måte å håndtere rekursjon på. Siden denne algoritmen er mer komplisert, var det mer utfordrende å finne en alternativ implementasjon. Blant annet ser vi ingen god måte å parallellisere en permutasjonssykel på. Derfor endte vi opp med å simulere rekursjon ved å bruke én stakk per tråd.

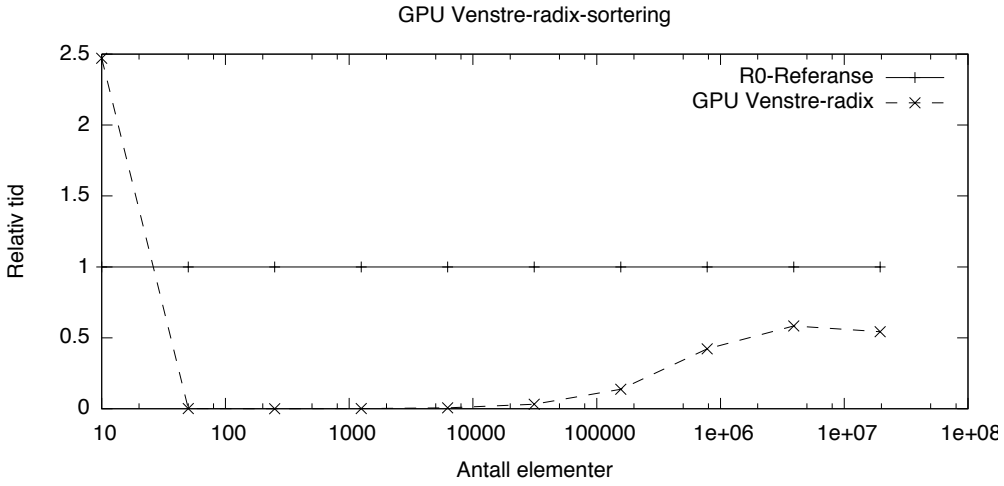
Siden vi alt har sett at GPU-en ikke egner seg til å kjøre sekvensiell kode, har vi valgt å gjøre første fordeling av tallene i bøtter på CPU. Deretter sorterer vi hver bøtte parallelt på GPU-en. For å oppnå en rimelig grad av parallellitet har vi valgt å dele array-et i 1024 bøtter. Algoritme 5.4 viser hvordan vi har implementert CUDA-kjernen.

Algoritme 5.4 Første algoritme (R1: GPU)

```
__global__ void radixKernel(int* a, int* point, int* border, int* startA,
    int* endA, int* leftBitNoA) {
  int threadid = blockDim.x * blockIdx.x + threadIdx.x;

  // Select stack by threadid
  int idxA    = 1;
  startA     += STACK_SIZE * threadid;
  endA       += STACK_SIZE * threadid;
  leftBitNoA += STACK_SIZE * threadid;
  ⋮
  while(idxA > 0) {
    idxA--; // Pop stack
    start     = startA[idxA];
    end       = endA[idxA];
    leftBitNo = leftBitNoA[idxA];
    ⋮
    «Sort numbers in buckets based on current bitmask»
    ⋮
    for(«each bucket») {
      if(bucketsize > 1) {
        // Push results on stack
        startA[idxA]     = t1;
        endA[idxA]       = t2-1;
        leftBitNoA[idxA] = leftBitNo;
        idxA++;
      }
    }
  }
}
```

Vi ser at selve stakken allokeres av vertskoden før kjernen kalles. Deretter sender vi en peker til den første stakken til kjernen. Hver tråd beregner så hvilken stakk de skal bruke, og henter ut første region av array-et som skal sorteres. Denne er allerede plassert på stakken i vertskoden. Så bruker vi samme algoritme som tidligere for å fordele tallene i bøtter. Og til slutt putter vi hver bøtte på stakken, slik at den kan sorteres "rekursivt".

Resultat: Figur 5.9 viser ytelsen til denne algoritmen, sammenlignet med referanseimplementasjonen. Vi ser at for array-er med mindre enn 100 000 elementer, er ytelsen svært dårlig (opp til 2500 ganger tregere). For større array-er er ytelsen noe bedre, og på det beste er den ca. 40 prosent tregere enn referanseimplementasjonen.



Figur 5.9: Ytelse: Venstre-radix-sortering på GPU

Dersom vi øker størrelsen på array-et ytterligere, støter vi på samme problem som vi hadde med flettesorteringen. Kjernen termineres automatisk fordi den bruker for lang tid (se kapittel 5.1.4 på side 49).

5.3.2 Andre algoritme (R2: GPU)

Ide: Dele kjernen i mindre biter

I forrige algoritme fikk vi problemer med å håndtere større datasett fordi den var implementer som én stor kjerne. Vi ønsker derfor å se hvordan vi kan splitte denne opp i flere mindre kjerner. I tillegg har vi fokusert på å øke båndbredden mot minnet, ved å utnytte koalesering der vi kan (se kapittel 5.1.4 på side 49).

Vi startet med å skrive om kjernen til å kun håndtere ett "kall" fra stakken hver gang den blir kjørt. Dette gjorde vi ved å fjerne den ytterste løkken i algoritme 5.4 på forrige side. Istedenfor må vi ha en test om stakken er tom i tilfelle ikke alle tråder blir ferdig samtidig. Vi måtte også skrive om C-koden til å sjekke stakken mellom hver gang kjernen kalles, og kjøre den om igjen dersom stakken ikke er tom. Deretter så vi på muligheten til å dele denne nye kjernen inn i mindre, spesialiserte kjerner. Vi endte opp med tre kjerner som vi vil beskrive under:

radix_clear: Hver gang vi går gjennom en ny bøtte må vi nullstille våre to hjelpe array-er (*point* og *border*). Vi ser at dette kan gjøres mer effektivt med en spesialisert kjerne. Algoritme 5.5 på neste side viser hvordan vi har implementert denne. Kjernen sørger for at alle skrivinger er koalesert, og utnytter trådparallelliteten bedre fordi den starter en tråd per element.

radix_count: En annen del av algoritmen som er lett å skille ut, er tellingen av elementer i hver bøtte. Algoritme 5.6 på neste side viser denne kjernen. Vi har modifisert kjernen

Algoritme 5.5 Andre algoritme (R2: GPU)

```
__global__ void radix_clear(int* point, int* border) {
  int threadid = blockDim.x * blockIdx.x + threadIdx.x;

  if(threadid < NUM_THREADS*POINT_SIZE)
    point[threadid] = 0;
}
```

slik at vi starter 32 tråder per bøtte. Trådene samarbeider med lesingen av array-et, slik at all lesing blir koalesert. Siden vi nå har flere tråder som oppdaterer samme verdier, bruker vi atomiske operasjoner for å øke tellerne.

Algoritme 5.6 Andre algoritme (R2: GPU)

```
#define IDX(idx) ((idx)*NUM_THREADS+threadid)

__global__ void radix_count(int* a, int* point, int* border, int* startA,
    int* endA, int* leftBitNoA, int* idxA) {
  int threadid = (blockDim.x * blockIdx.x + threadIdx.x) / 32;
  int offset = (blockDim.x * blockIdx.x + threadIdx.x) % 32;

  // Pop stack
  if(offset == 0)
    idxA[threadid] -= 1;

  __syncthreads();
  if(idxA[threadid] >= 0) {
    int stack_idx = IDX(idxA[threadid]);
    int start     = startA[stack_idx];
    int end       = endA[stack_idx];
    int leftBitNo = leftBitNoA[stack_idx];
    :
    «Calculate bitmask»
    :
    for(i = start; i<= end; i += 32) {
      if(i + offset <= end) {
        int value = a[i + offset];
        int index = IDX((value & mask) >> rBitNo);
        atomicInc((unsigned int*)&point[index], 0xFFFFFFFF);
      }
    }
  }
}
```

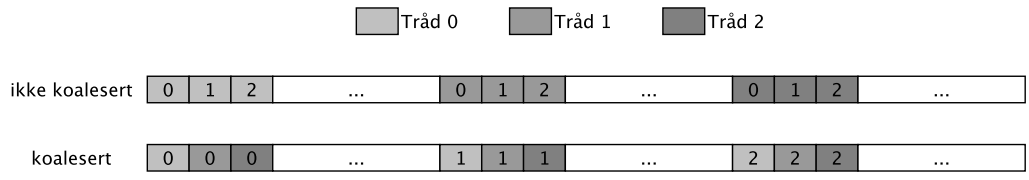
radix_shuffle: Den siste kjernen tar seg av fordelingen av tall i bøtter ved hjelp av permutasjonssykler. Implementasjonen er vist i algoritme 5.7 på neste side.

Til slutt har vi også endret på måten trådene aksesserer stakken på, slik at disse blir mer koalesert enn de var. I forrige algoritme lot vi hver tråd ha en egen del av array-et som de brukte til stakken. Som vi ser i figur 5.10 på neste side fører dette til veldig spredt minne-aksess. Vi ser at det er smartere å ha hvert element i stakkene samlet. Algoritme 5.6 og 5.7 viser hvordan vi gjør dette i koden.

Resultat: Figur 5.11 på side 60 viser ytelsen til denne algoritmen. Vi ser den yter noen

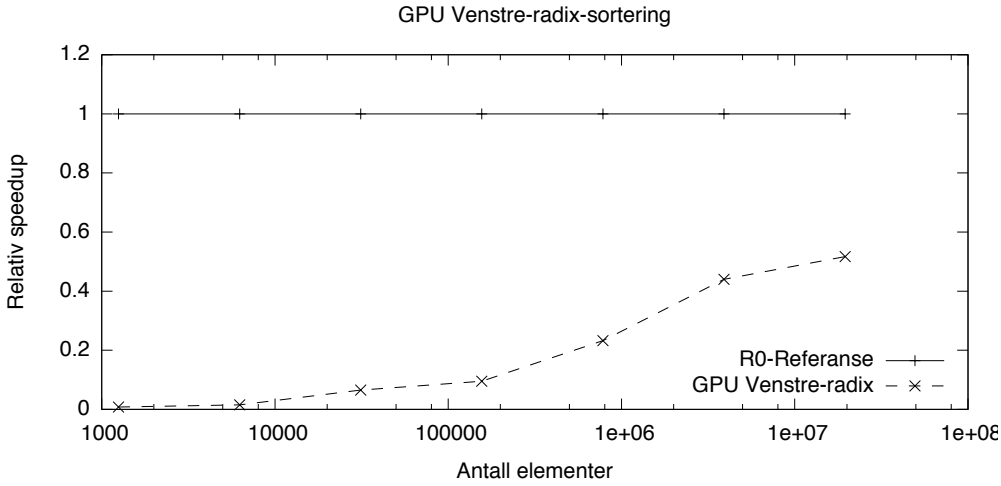
Algoritme 5.7 Andre algoritme (R2: GPU)

```
__global__ void radix_shuffle(int* a, int* point, int* border, int* startA,
    int* endA, int* leftBitNoA, int* idxA) {
  int threadid = blockDim.x * blockIdx.x + threadIdx.x;

  if(idxA[threadid] >= 0) {
    int start     = startA[IDX(idxA[threadid])];
    int end       = endA[IDX(idxA[threadid])];
    int leftBitNo = leftBitNoA[IDX(idxA[threadid])];
    :
    «Sort numbers in buckets based on current bitmask»
    :
    for(«each bucket») {
      if(bucketsize > 1) {
        // Push results on stack
        startA[idxA]     = t1;
        endA[idxA]       = t2-1;
        leftBitNoA[idxA] = leftBitNo;
        idxA++;
      }
    }
  }
}
```



Figur 5.10: Forskjell på koalesert og ikke koalesert minne-aksess

få prosent dårligere enn forrige algoritme. Dette skyldes trolig den ekstra kostnaden ved å splitte kjernen i mindre biter. Fordelen er selvsagt at denne implementasjonen er i stand til å sortere mye større array-er fordi den ikke får problemer med 5-sekunders grensen.



Figur 5.11: Ytelse: Venstre-radix-sortering på GPU

Legg merke til at vi ikke har noen målinger for array-er med mindre enn 1000 elementer. Dette skyldes at vi har fått en liten feil i algoritmen. Denne gjør at enkelte array-er utløser en evig løkke. Vi har ikke klart å finne denne feilen i tiden vi hadde til rådighet, men tror ikke den vil påvirke ytelsen til algoritmen. Siden vi er mest interessert i ytelsen for store array-er, har vi valgt å ikke gjøre noe med den.

Kapittel 6

Diskusjon

I dette kapittelet vil vi se tilbake på algoritmene vi har utviklet. Først vil vi presentere Amdahl's lov, som er svært nyttig til å analysere parallelle algoritmer. Deretter vil vi oppsummere resultatene vi oppnådde, før vi ser nærmere på hver enkelt algoritme. Til slutt vil vi se litt mer generelt på parallell sortering, og hva vi har lært i denne oppgaven.

6.1 Amdahl's lov

Amdahl's lov gir en øvre grense for hvor stor forbedring i hastighet man kan oppnå når man parallelliserer et problem [Goetz et al., 2006, side 225]. Loven tar utgangspunkt i at det er sjeldent alle deler av et problem kan løses parallelt. Som regel er det alltid noen steg som må gjøres i en bestemt rekkefølge, hvor ett steg er avhengig av at det forrige er fullført. For eksempel kan vi se på den andre flettesorteringsalgoritmen vi implementerte på CPU (se kapittel 4.2.2 på side 33). I første kall på *mergesort()* delte vi listen i to deler som hver kunne sorteres uavhengig. Men etterpå ble disse to delene sekvensielt flettet til en sortert liste. Hvis vi lar F være andelen av algoritmen som må gjøres sekvensielt, sier Amdahl's lov at maksimal speedup på en maskin med N kjerner vil være:

$$speedup \leq \frac{1}{F + \frac{(1-F)}{N}}$$

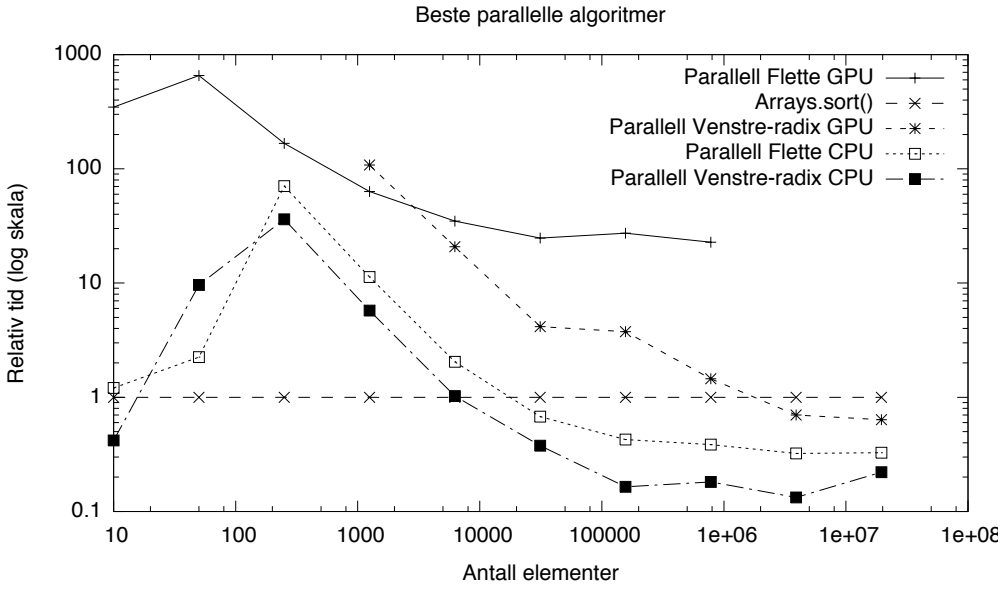
Når antall kjerner øker mot uendelig, ser vi at dette uttrykket grenser mot $1/F$. Dette betyr at dersom 10 prosent av et program må utføres sekvensielt, kan vi aldri få en speedup større enn 10. Hvis vi ønsker god skalerbarhet er det derfor svært viktig å sørge for at minst mulig av algoritmen må gjøres sekvensielt.

6.2 Resultatene

Generelt presterte algoritmene for CPU ganske bra, og algoritmene på GPU ikke fullt så bra. Her vil vi vise en oversikt over algoritmene sammenlignet med hverandre. Så vil vi i kapittel 6.3 gå igjennom hver enkelt algoritme.

6.2.1 Kjøretid

I figur 6.1 ser vi kjøretiden til alle fire algoritmene sammenlignet med Javas innebygde sortering.



Figur 6.1: Sammenlikning av de beste algoritmene (lavere tid er bedre)

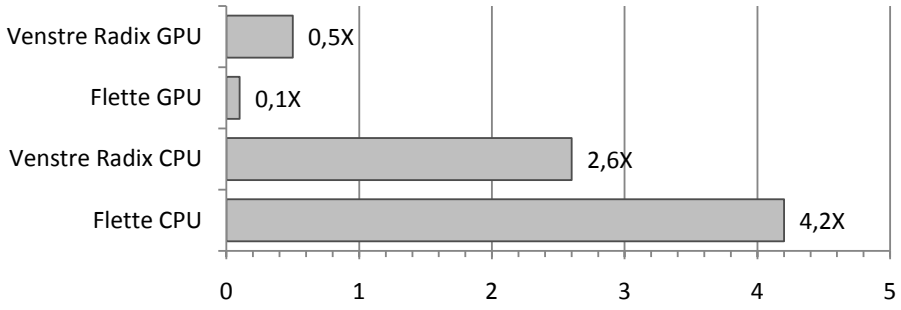
Vi ser en fellesnevner for alle algoritmene, er at de yter dårlig for små array-er. Vi må opp i minimum 5 000 elementer, før de blir raskere enn å bruke vanlig sortering i Java. Jeg anser allikevel dette for å være et lite problem. Det er først når datasettene blir store at vi har behov for effektive algoritmer. Derfor kan vi like godt skifte til en rask sekvensiell algoritme hvis antall elementer er under en gitt grense. For eksempel kunne vi bruke [Maus, 2002] som vi har sett er ca. 3 ganger så rask som *Arrays.sort()*.

6.2.2 Speedup

Speedup er en annen viktig faktor å ta hensyn til når vi evaluerer ytelsen til algoritmene. Figur 6.2 på neste side viser en oversikt over hvilken speedup vi oppnådde med algoritmene våre. Denne er målt i forhold til referanseimplementasjonen til hver algoritme.

6.2.3 Skalerbarhet

Den siste faktoren vi skal se på er *skalerbarhet*. Jeg har ikke hatt tid til å teste skalerbarheten på maskiner med flere kjerner. Istedenfor vil jeg senere benytte meg av Amdahl's lov for å estimere hvor godt jeg tror hver algoritme vil skalere når antall kjerner øker.



Figur 6.2: Beste speedup for hver av algoritmene

6.3 Algoritmene

6.3.1 Flettesortering på CPU

Dette var vår nest beste algoritme. I figur 6.1 på forrige side ser vi at den er opp til 3 ganger raskere enn Javas innebygde sortering.

Det mest interessante resultatet vi fikk fra denne algoritmen, var hastighets forbedringen i forhold til referansealgoritmen. Figur 6.2 viser at den parallelle algoritmen var opp til 4,2 ganger raskere. Vi ser altså en større økning i hastighet, enn antall kjerner vi benyttet. Dette kalle super-lineær-speedup [Wikipedia, 2010b].

Hvis vi bruker den sekvensielle algoritmen til å sortere 20 millioner tall, ser vi at ca. 4,7 prosent av tiden går med til å flette de siste to halvdelene til én sortert liste. På 4 kjerner får vi da etter Amdahl's lov maksimalt en speedup på 3,5. Dette stemmer ganske godt overens med vår tredje algoritme for flettesortering (se kapittel 4.2.3 på side 34), hvor vi så en speedup på 3,3.

I den endelige algoritmen vår gikk vi over til å gjøre flettingen i parallell. Dermed reduserte vi andelen arbeid som må gjøres sekvensielt ytterligere. Det er også verdt å legge merke til at vi legger til en ekstra kostnad, siden vi må benytte binærsøk til å finne indeksene til bitene som skal flettes. For større array-er blir allikevel denne kostnaden minimal i forholdt til tiden man sparer på flettingen.

Det at vi oppnådde en speedup større enn 4, mener jeg skyldes den økte mengden cache-minne man får ved å bruke flere kjerner. Om vi ser tilbake til figur 3.8 på side 26 ser vi at en kjerne maksimalt kan bruke 3MB cache-minne, mens vi ved å bruke 4 kjerner får tilgang til 6MB cache-minne.

Selv om vi oppnådde bra speedup, var ikke denne algoritmen like rask som venstre-radix-algoritmen. Dette mener jeg skyldes at flettesorteringen leser og skriver data flere ganger. Mens venstre-radix-algoritmen kun leser og skriver hvert element 3 til 4 ganger, vil flettesorteringen gjøre det $\log n$ ganger. For 1 million elementer vil altså dataene leses og skrives ca. 20 ganger.

I [Chhugani et al., 2008] presenteres det en interessant algoritme som forsøker å adressere dette problemet. Først benyttes en spesialisert algoritme for å sortere blokker små nok til å få plass i prosessorens cache-minne. På vår testmaskin tilsvarer dette ca. 1,5 millioner elementer. Deretter flettes disse blokkene ved hjelp av et tre med køer (se kapittel 2.3.1

på side 8 for nærmere beskrivelse). På denne måten flettes blokkene ved kun å lese/skrive hvert element én gang.

Dersom vi skulle arbeide videre med algoritmen, hadde det vært veldig spennende å forsøkt en tilsvarende tilnærming i Java og sett hvilken ytelse vi kunne oppnå med dette.

6.3.2 Venstre-radix-sortering på CPU

Dette var vår raskeste algoritme. Figur 6.1 på side 62 viser at den for array-er større enn 100 000 elementer er 4 til 7 ganger raskere enn Javas innebygde sortering.

I figur 6.2 på forrige side ser vi at denne algoritmen kun oppnår en speedup på 2,7. I forhold til flettesorteringen er ikke dette så veldig bra. Vi går derfor tilbake til referanseimplementasjonen, og ser på hvor stor andel av algoritmen som utføres sekvensielt. Dersom vi i første fordeling bruker en bitmaske med størrelse 4, ser vi at ca. 17 prosent av tiden går med til å fordele tallene i bøtter. Amdahl's lov gir oss da med 4 kjerner en maksimal speedup på 2,6. Dette stemmer svært bra med resultatet vi fikk.

Det er tydelig at dersom vi ønsker en høyere speedup, må vi redusere andelen av koden som kjører sekvensielt. For denne algoritmen betyr det at vi må finne en måte å parallellisere den første fordelingen av tall i bøtter. Umiddelbart virket dette vanskelig, fordi permutasjonssyklene aksesserer array-et relativt tilfeldig. Men etter nærmere ettertanke, har jeg kommet til at dette antagelig er ganske lett. Jeg foreslår derfor følgende algoritme:

1. Telle elementer i hver bøtte: Dette steget er opplagt parallelliserbart. Dersom vi har k kjerner tilgjengelig, deler vi array-et i k biter og lar hver kjerne telle elementer for sin del. Til slutt summerer vi resultatet. For å være sikker på å unngå problemer med synkronisering bør vi benytte Javas nye tellere med atomiske operasjoner (se kapittel 4.1.4 på side 31) i siste steg.

2. Fordele elementene i riktig bøtte: Når vi ser nærmere på hvordan en permutasjonssykel opererer, ser vi at det er faktisk mulig å utføre k permutasjonssykler samtidig. Den kritiske delen er å unngå at to tråder plasserer et element i samme posisjon. Dette kan man enkelt unngå ved å bruke atomiske tellere.

Jeg har ikke hatt tid til å teste denne nye algoritmen, men ser for meg at den kan gi betydelig forbedring i hastighet. Jeg anser derfor dette som et naturlig sted å fortsette om man ønsker å utforske denne algoritmen videre.

Valg av maskestørrelse er en annen viktig faktor man bør se nærmere på dersom man skal gå videre med algoritmen. For øyeblikket har vi kun valgt maskestørrelse i forhold til antall kjerner, med $4 \cdot k$ bøtter for å sikre jevn arbeidsfordeling. Vi innser nå at det også vil være viktig å ta hensyn til størrelsen på prosessorens cache-minne. Hvis M er antall elementer som får plass i det totale cache-minnet og k er antall kjerner, bør vi derfor minimum ha $\frac{n \cdot k}{M}$ bøtter.

6.3.3 Flettesortering på GPU

Dette var helt klart vår minst vellykkede algoritme. Figur 6.1 på side 62 viser at den er 22 til 655 ganger *tregere* enn Javas innebygde sortering. Her vil vi se på de viktigste årsakene til at vi ikke oppnådde bedre resultat.

[Satish et al., 2008] har allerede demonstrert at det er mulig å lage en effektiv implementasjon av flettesortering på GPU. Dermed er det tydelig at denne algoritmens dårlige resultater kun skyldes en dårlig implementasjon. Dette kommer nok først og fremst av min manglende erfaring med GPU-programmering. GPU-en er en komplisert plattform, og det har tatt tid å få innsikt i hvordan den kan utnyttes.

Det er allikevel verdt å merke seg hvor vanskelig det er å skrive effektive programmer på GPU. Selv enkle operasjoner som *Parallel Prefix Sum* [Sengupta et al., 2007] krever svært gjennomtenkt kode for å oppnå god ytelse. Selv om CUDA gjør det enkelt å komme i gang med GPU-programmering, ser vi at det kreves mye finjustering for å virkelig utnytte regnekraften.

Det største problemet med implementasjonen vår er mangelen av parallellitet. Vi benytter for få tråder og gjør for mye arbeid i hver tråd. I kapittel 2.4 på side 9 har vi sett at GPU-en best utnyttes når vi kun behandler noen få elementer i hver tråd. I likhet med flettesortering på CPU ser vi at det er nødvendig å gjøre flettingen i parallell for å oppnå høyere ytelse.

Den andre store feilen med implementasjonen vår er at vi ikke utnytter muligheten til å bruke delt minne. Ved å utnytte det delte minnet kan blokker med opp til 512 elementer sorteres internt på hver SM. Dermed blir bruken av globalt minne redusert, og det er lettere å oppnå høy ytelse. I kapittel 2.4 ser vi flere eksempler på at god bruk av delt minne er nøkkelen til gode resultater på GPU.

Dersom vi skulle gått videre med denne algoritmen ville jeg først fokusert på å parallellisere flettingen. Som vi har sett tidligere er denne delen kritisk for å oppnå god parallell ytelse med flettesortering. Spesielt er det viktig for å utnytte parallelliteten i GPU-en. Algoritmen presentert i [Satish et al., 2008] virker som et naturlig sted å begynne (se figur 2.6 på side 12). Deretter ville jeg utforsket mulighetene for å utnytte delt minne. For eksempel ved å splitte blokkene som skal flettes inn i så små biter at to og to biter kan flettes i delt minne.

6.3.4 Venstre-radix-sortering på GPU

Heller ikke denne algoritmen presterte særlig godt på GPU. Som vi ser i figur 6.1 på side 62, er den opp til 1,5 ganger raskere enn Javas innebygde sortering. Allikevel er den bare halvparten så rask som referanseimplentasjonen (se figur 6.2 på side 63).

Vårt største problem med denne algoritmen var å håndtere rekursjon. Vi endte opp med å dele listen i 1024 bøtter, og lot hver bøtte bli håndtert av én tråd. Denne tilnærmingen fungerte bedre enn algoritmen for flettesortering, men var fortsatt ikke god nok til å utnytte parallelliteten i GPU-en.

[Yang et al., 2009] presenterer tre alternative metoder for å implementere rekursjon på GPU. De foreslår å bruke en stakk per varp eller per blokk istedenfor per tråd. Ved å benytte flere tråder per bøtte ville parallelliteten i algoritmen vår økt, og det ville trolig være enklere for GPU-en å fordele arbeidsmengden jevnt over alle kjernene. Dette forutsetter selvsagt at stegene i venstre-radix-algoritmen lar seg parallellisere over en varp eller blokk.

Det steget i algoritmen som er vanskeligst å parallellisere er permutasjonssyklene. Vi har sett i kapittel 6.3.2 på forrige side et forslag til hvordan dette kan gjøres, men jeg er usikker

på om denne metoden vil egne seg for GPU. Et annet problem med permutasjonssykler er at de aksesserer minnet i en mer eller mindre vilkårlig rekkefølge. Dette fører igjen til lite koalesert lesing og skriving, som også reduserer effektiviteten av algoritmen.

Den siste svakheten med algoritmen jeg vil trekke frem er at første fordeling av tall i bøtter gjøres på CPU. Dette er et relativt tungt sekvensielt steg som derfor setter store begrensninger på hvor stor speedup man kan få.

Vi ser det er flere store problemer som må overkommes dersom vi skal få denne algoritmen til å kjøre effektivt på GPU-en. Dersom vi skal lage en effektiv implementasjon må omtrent alle delene av algoritmen skrives om for å utnytte GPU-en. Vi kan derfor ikke utelukke at en slik implementasjon finnes, men har sett at vil kreve svært omfattende endringen av algoritmen.

6.4 Generelt om parallell sortering

I denne oppgaven har vi forsøkt en håndfull forskjellige metoder for å parallellisere sortering. Vi startet med et ønske om å undersøke hvorvidt rekursive algoritmer egner seg godt for parallellisering (se kapittel 3.1.1 på side 15).

Vi har sett at det å kun starte rekursive kall i nye tråder ikke gav noen hastighetsforbedring i noen av algoritmene våre. Derfor begrenset vi antall tråder i forhold til antall kjerner. Dette gav oss bedre resultater. I algoritme *F2* på side 33 fikk vi en speedup opp til 3,3X og i algoritme *R2* på side 39 en speedup opp til 2X. Altså har vi sett at vi med svært enkle grep få en speedup på 2–3 ganger med 4 kjerner.

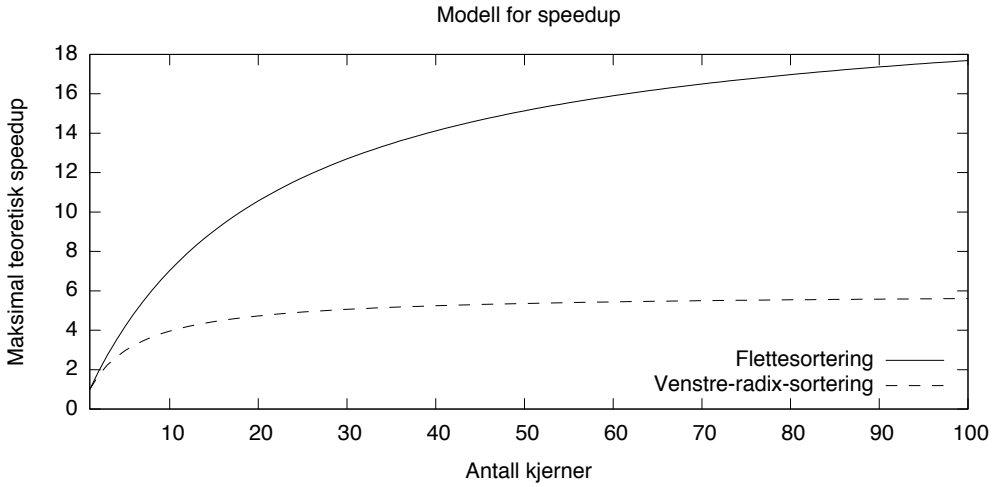
Vi har også sett at denne tilnærmingen har ett fundamentalt problem; det første kallet må alltid gjøres sekvensielt. I referansealgoritmene våre utgjør dette ca. 4,7 prosent for flettesorteringen og ca. 17 prosent for venstre-radix-sorteringen. Amdahl's lov gir oss da sterke begrensninger på hvor høy speedup man kan få. Figur 6.3 på neste side viser den maksimale speedup vi kan forvente når antall kjerner øker[1].

Selv med denne optimistiske modellen ser vi at skalerbarheten til algoritmene er ganske dårlig. Det er derfor tydelig at det ikke er nok å parallellisere kun de rekursive kallene. Dersom vi skal få en effektiv og skalerbar algoritme er det også nødvendig å parallellisere det første steget. Med flettesorteringen fikk vi dette til, og så en signifikant forbedring i hastighet (se kapittel 4.2.4 på side 36. Med venstre-radix algoritmen rakk vi ikke å teste dette, men har presentert et forslag til hvordan det kan gjøres i kapittel 6.3.2 på side 64.

Våre forsøk på GPU var mindre vellykket, og vi har sett at det er vanskelig å lage gode sorteringsalgoritmer på denne plattformen. GPU-en har en komplisert arkitektur og det er hovedsaklig utvikleren som har ansvar for å utnytte den. Vi har sett at det kreves mye finjustering av algoritmen for å oppnå god ytelse. Med venstre-radix algoritmen klarte vi å sortere raskere enn *Arrays.sort()*, men resultatet var langt fra så bra som vi hadde håpet på.

Vi opplevde også at det var vanskelig å feilsøke problemer på GPU-en. Mens vi fra CPU-programmering er vandt til gode debuggere hvor man kan stoppe kjøringen på angitte punkter, pause programmet under kjøring og følge programmet linje for linje,

[1] Dette er en optimistisk modell som ikke tar hensyn til om man fullt ut får utnyttet alle kjernene på den parallelle delen av algoritmen. For eksempel vil flettesorteringsalgoritmen i andre nivå av rekursjonen kun utnytte to kjerner, uansett hvor mange som er tilgjengelig.



Figur 6.3: Modell for speedup av de sekvensielle algoritmene

hadde vi ingen slike muligheter på GPU. Det finnes riktignok en kommandolinjebasert debugger, cuda-gdb, men denne er kun tilgjengelig på Linux. En plugin til Visual Studio på Windows er også under utvikling, men denne har vi ikke hatt tilgang til. Med bedre muligheter for feilsøking, ville vi trolig ha klart å rette feilen som oppstod i algoritme *R2: GPU* (se side 57).

Til slutt vil vi også stille spørsmål ved hvorvidt det er hensiktsmessig å benytte GPU for å sortere tall fra Java. For å benytte GPU-en fra Java, må vi gå gjennom JNI. Dette betyr at array-et som skal sorteres først må kopieres til C-koden, som igjen må kopiere det til GPU-en. Når så dataene er sortert må de kopieres tilbake. For 20 millioner tall tar dette ca. 170ms, mens *Arrays.sort()* sorterer 20 millioner tall på 2,9 sekunder. Vi ser altså at uansett hvor rask GPU-algoritmen er, kan den aldri bli mer enn 17 ganger så rask som *Arrays.sort()*. Dette skyldes at sortering har så effektiv løsning ($O(n \log n)$) at tiden det tar å kopiere data til/fra skjermkortet blir en vesentlig kostnad.

Kapittel 7

Konklusjon

Vi har i denne oppgaven designet og implementert fire parallelle algoritmer; to for flerkjerne-CPU og to for GPU. Hver av algoritmene har vi utviklet gjennom flere iterasjoner ved først å starte med en idé, deretter forsøke denne og til slutt lære av resultatene.

På flerkjerne-CPU erfarte vi at det var relativt enkelt å oppnå gode resultater. Vår implementasjon av parallell venstre-radix-sortering var med 4 kjerner opp til 7 ganger raskere enn Javas innebygde sortering, mens parallell flettesortering var opp til 3 ganger raskere (se figur 6.1 på side 62). Vi har også sett at begge disse algoritmene viser potensiale til å oppnå enda høyere ytelse.

Blant artiklene vi har sett på presenterer [Chhugani et al., 2008] den raskeste algoritmen. Denne algoritmen er implementert med en kombinasjon av C og assembler, og rapporteres å sortere 4 millioner tall på 26,9ms. Dermed er den 19,6 ganger raskere enn Arrays.sort() i Java[1]. Den raskeste GPU-algoreitmen presenteres i [Chen et al., 2009] og sorterer 20 millioner tall på ca. 384ms. Dermed blir den 7,6 ganger raskere enn Arrays.sort(). Om vi korrigerer for tiden det vil ta å kalle disse algoritmene gjennom JNI (se figur 5.5 på side 52), blir disse verdiene henholdsvis 12,4 ganger og 6,3 ganger raskere.

Vi ser at våre algoritmer presterer bra, men ikke fullt på høyde med disse. Men sett fra et Java perspektiv mener jeg allikevel at vår tilnærming er mer fornuftig. Ved å bruke JNI mister man plattform-uavhengigheten til Java, og kostnaden ved å kopiere data til/fra C-koden er så høy at vi kun får begrenset gevinst.

Algoritmene vi utviklet på GPU var mindre vellykket. Vi erfarte at CUDA gjør det lett å komme i gang med GPU-programmering, men at det er vanskelig å utnytte GPU-en effektivt. Vi har også sett at sortering på CPU går så raskt at tiden det tar å kopiere data til/fra skjermkortet setter en stor begrensning på hvilken forbedring i hastighet vi kan oppnå. Sortering er rett og slett for "lett" til å få noen særlig gevinst av å benytte GPU.

Vi ser allikevel at det fins situasjoner hvor det vil være gunstig å sortere på GPU. Det kan for eksempel hende at sorteringen kun er et steg i en større algoritme, og dataene som skal sorteres blir generert på skjermkortet. I dette tilfellet virker det mer hensiktsmessig å sortere dataene *på* GPU-en enn å kopiere dem til/fra systemhukommelsen for å sorteres av CPU-en.

[1] Legg merke til at dette tallet er litt for stort da våre målinger er gjort på en tregere CPU

I introduksjonen (se side 1) etterlyste vi generelle teknikker for å parallellisere sekvensielle algoritmer. Gjennom arbeidet med oppgaven har vi sett at parallellisering av rekursive kall i seg selv ikke er nok til å oppnå god ytelse. Vi må også parallellisere det første steget av algoritmen. Dette virker til å måtte gjøres individuelt for hver algoritme. Vi har derfor ikke funnet noen generell teknikk, men kan heller ikke utelukke at en slik finnes.

Om jeg skulle gjort noe annerledes under arbeidet med oppgaven, så skulle jeg brukt noe mer tid på artiklene før jeg startet arbeidet med algoritmene. Artiklene om parallell sortering på CPU fant jeg ikke før sent i oppgaven, og fikk derfor ikke mulighet til å prøve ut ideene som kom fra å lese disse. Først og fremst gjelder dette ideen om parallell fletting ved hjelp av køer (se figur 2.5 på side 8) og generelt at jeg kunne tatt bedre hensyn til størrelsen på CPU-ens cache-minne.

Den overordnede lærdommen vi kan trekke ut ifra denne oppgaven er at det tilsynelatende er langt enklere å parallellisere algoritmer på flerkjerne-CPU enn på GPU. Dette var i alle fall min erfaring med disse plattformene.

Tillegg A

Kildekode

All kildekode vi har utviklet i løpet av denne oppgaven ligger vedlagt i en zip-fil på CD. Tabell A.1 viser en oversikt over hver av algoritmene og hvor man finner dem på CD-en.

Algoritme	Beskrevet i	Sti på CD-en
F0: Referanse	kapittel 3.1.2 på side 17	`java/algorithms/MergeSort.java`
R0: Referanse	kapittel 3.1.3 på side 19	`java/algorithms/LeftRadixSort.java`
F1: Naiv	kapittel 4.2.1 på side 32	`java/algorithms/ParallelMergeSort1.java`
F2: Maks dybde	kapittel 4.2.2 på side 33	`java/algorithms/ParallelMergeSort2.java`
F3: Gjenbruk tråd	kapittel 4.2.3 på side 34	`java/algorithms/ParallelMergeSort3.java`
F4: Parallell fletting	kapittel 4.2.4 på side 36	`java/algorithms/ParallelMergeSort4.java`
R1: Naiv	kapittel 4.3.1 på side 39	`java/algorithms/ParallelLeftRadixSort1.java`
R2: Ett nivå	kapittel 4.3.2 på side 39	`java/algorithms/ParallelLeftRadixSort2.java`
R3: Trådsamling	kapittel 4.3.3 på side 40	`java/algorithms/ParallelLeftRadixSort3.java`
F1: GPU	kapittel 5.2.1 på side 53	`c/MergeSort1/merge_v1_jni.cu`
R1: GPU	kapittel 5.3.1 på side 56	`c/LeftRadixSort1/radix_v1-jni.cu`
R2: GPU	kapittel 5.3.2 på side 57	`c/LeftRadixSort2/radix_v2-jni.cu`

Tabell A.1: Oversikt over innhold på CD-en

Zip-filen har md5sum: `f121ff69fac1f4c1e7eeec6a5a292953`

Bibliografi

[Akl, 1985] Akl, S. G. (1985). *Parallel Sorting Algorithms*. Academic Press.

[AMD, 2010] AMD (2010). ATI Stream Technology. http://www.amd.com/stream [Testet: 19.05.2010].

[Batcher, 1968] Batcher, K. E. (1968). Sorting networks and their applications. I *Proc. AFIPS Spring Joint Computer Conference*, side 307–314. ACM.

[Cederman og Tsigas, 2008] Cederman, D. og Tsigas, P. (2008). A Practical Quicksort Algorithm for Graphics Processors. I Halperin, D. og Mehlhorn, K., red, *Algorithms - ESA 2008, 16th Annual European Symposium, Karlsruhe, Germany, September 15-17, 2008. Proceedings*, volume 5193 of *Lecture Notes in Computer Science*, side 246–258. Springer. ISBN 978-3-540-87743-1.

[Chen et al., 2009] Chen, S., Qin, J., Xie, Y.-M., Zhao, J. og Heng, P.-A. (2009). A Fast and Flexible Sorting Algorithm with CUDA. I Hua, A. og Chang, S.-L., red, *Algorithms and Architectures for Parallel Processing, 9th International Conference, ICA3PP 2009, Taipei, Taiwan, June 8-11, 2009. Proceedings*, volume 5574 of *Lecture Notes in Computer Science*, side 281–290. Springer. ISBN 978-3-642-03094-9.

[Chhugani et al., 2008] Chhugani, J., Nguyen, A. D., Lee, V. W., Macy, W., Hagog, M., Chen, Y.-K., Baransi, A., Kumar, S. og Dubey, P. (2008). Efficient implementation of sorting on multi-core SIMD CPU architecture. *Proceedings of the VLDB Endowmen*, 1(2):1313–1324.

[Coombe og Harris, 2005] Coombe, G. og Harris, M. (2005). *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional.

[Fernando, 2004] Fernando, R. (2004). *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education.

[Gedik et al., 2007] Gedik, B., Bordawekar, R. og Yu, P. S. (2007). CellSort: High Performance Sorting on the Cell Processor. I *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, side 1286–1207. ACM. ISBN 978-1-59593-649-3.

[Goetz et al., 2006] Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D. og Lea, D. (2006). *Java Concurrency in Practice*. Addison Wesley.

[Govindaraju et al., 2005] Govindaraju, N. K., Raghuvanshi, N., Henson, M., Tuft, D. og Manocha, D. (2005). A Cache-Efficient Sorting Algorithm for Database and Data Mining Computations using Graphics Processors. Rapport TR05-016, Department of Computer Science, University of North Carolina - Chapel Hill.

[Greß og Zachmann, 2006] Greß, A. og Zachmann, G. (2006). GPU-ABisort: optimal parallel sorting on stream architectures. I *International Parallel & Distributed Processing Symposium*. IEEE.

[Inoue et al., 2007] Inoue, H., Moriyama, T., Komatsu, H. og Nakatani, T. (2007). AA-sort: A new parallel sorting algorithm for multi-core SIMD processors. I *Parallel Architecture and Compilation Techniques*, side 189–198. IEEE Computer Society.

[Intel Corporation, 2010] Intel Corporation (2010). Sh: A High-level metaprogramming language for modern GPUs. http://libsh.org/ [Testet: 19.05.2010].

[JáJá, 1992] JáJá, J. (1992). *An Introduction to Parallel Algorithms*. Addison-Wesley.

[Jodd Team, 2010] Jodd Team (2010). Fast merge sort. http://www.java2s.com/Code/Java/Collections-Data-Structure/FastMergeSort.htm [Testet: 19.05.2010].

[Khronos Group, 2010] Khronos Group (2010). OpenCL. http://www.khronos.org/opencl/ [Testet: 19.05.2010].

[Knuth, 1973] Knuth, D. E. (1973). *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley.

[Lacey og Box, 1991] Lacey, S. og Box, R. (1991). A fast, easy sort: A novel enhancement makes a bubble sort into one of the fastest sorting routines. *Byte Magazine*, 16(4):315–316, 318, 320.

[Maus, 2002] Maus, A. (2002). ARL, a faster in-place, cache friendly sorting algorithm. I *NIK'2002, Norsk Informatikkonferanse*, Kongsberg, Norway. ISBN 82-91116-45-8.

[Microsoft Corporation, 2010] Microsoft Corporation (2010). DirectCompute. http://code.msdn.microsoft.com/directcomputehol [Testet: 19.05.2010].

[Miller og Boxer, 2000] Miller, R. og Boxer, L. (2000). *Algorithms sequential & parallel — A unified approach*. Prentice-Hall.

[NVIDIA, 2007] NVIDIA (2007). CUDA: Programming Guide v1.1. http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf [Testet: 19.05.2010].

[NVIDIA, 2010] NVIDIA (2010). Cuda zone. http://www.nvidia.com/cuda [Testet: 19.05.2010].

[Parikh, 2008] Parikh, R. (2008). Accelerating quicksort on the intel pentium 4 processor with hyper-threading technology. http://software.intel.com/en-us/articles/accelerating-quicksort-on-the-intel-pentiumr-4-processor-with-hyper-threading-technology/ [Testet: 19.05.2010].

[Purcell et al., 2003] Purcell, T. J., Donner, C., Cammarano, M., Jensen, H. W. og Hanrahan, P. (2003). Photon mapping on programmable graphics hardware. I *Proceedings of Graphics Hardware 2003*, side 41–50.

[Satish et al., 2008] Satish, N., Harris, M. og Garland, M. (2008). Designing efficient sorting algorithms for manycore GPUs. I *IEEE International Symposium on Parallel & Distributed Processing, 2009*, side 1–10.

[Sengupta et al., 2007] Sengupta, S., Harris, M., Zhang, Y. og Owens, J. D. (2007). Scan primitives for GPU computing. I Segal, M. og Aila, T., red, *Graphics Hardware*, side 97–106. Eurographics Association.

[Shimpi og Wilson, 2008] Shimpi, A. L. og Wilson, D. (2008). NVIDIA's 1.4 Billion Transistor GPU: GT200 Arrives as the GeForce GTX 280 & 260. http://www.anandtech.com/show/2549 [Testet: 19.05.2010].

[Stanford University Graphics Lab, 2010] Stanford University Graphics Lab (2010). BrookGPU. http://graphics.stanford.edu/projects/brookgpu/ [Testet: 19.05.2010].

[Wikipedia, 2010a] Wikipedia (2010a). Graphics processing unit — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Graphics_processing_unit&oldid=347403422 [Benyttet: 03.03.2010, Testet: 19.05.2010].

[Wikipedia, 2010b] Wikipedia (2010b). Speedup — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Speedup&oldid=356264568 [Benyttet: 06.05.2010, Testet: 19.05.2010].

[Yang et al., 2009] Yang, K., He, B., Luo, Q., Sander, P. V. og Shi, J. (2009). Stack-based parallel recursion on graphics processors. I *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (14th PPOPP'09)*, side 299–300, Raleigh, NC, USA. ACM.