Parallel-vector-modellen på GP-GPU via CUDA

Joakim Ahnfelt-Rønne Mikkel Kjær Jensen

17. april 2009

Resumé

Vi har undersøgt hvorvidt [Blelloch] Vektor-Scan for parallelisme er realiserbar på grafikkort via CUDA, NVIDIAs standard for GP-GPU-beregninger. Vi har implementeret et bibliotek der gør det muligt at bruge modellen til at programmere på grafikkort uden at skulle kende til CUDA. Denne er imidlertid for langsom til at være brugbar i praksis. Vi har undersøgt hvorvidt det ville gøre modellen brugbar hvis man erstattede vores scan-primitiv med NVIDIAs egen implementation, som er betydeligt hurtigere. Vi argumenterer for at modellen ikke er brugbar selv med denne implementation. Vi beskriver desuden hvad der gør NVIDIAs implementation hurtigere end vores.

Indhold

1 Introduktion **2**
1.1 Mål . . . 2

2 CUDA **3**
2.1 Introduktion . . . 3
2.2 Blokke og tråde . . . 3
2.3 Kernel . . . 3
2.4 Hukommelse . . . 4

3 Overvejelser **5**
3.1 Tidspunkt for valg af operator . . . 5
3.1.1 Run Time . . . 5
3.1.2 Compile time . . . 5

4 Vector Scan Modelen **7**
4.1 Paralleliserbare algoritmer . . . 9
4.2 Implementation af primitiver i forhold til modellen . . . 10
4.2.1 Scan . . . 10
4.2.2 Segmented Scan . . . 10
4.2.3 Copy . . . 10
4.2.4 Distribute . . . 10
4.2.5 Split . . . 10
4.3 Anvendelsesmuligheder . . . 11

5 Bibliotek **12**
5.1 Datastrukturer 12
5.2 Operatorer 12
5.3 Algoritmer 13
5.4 Eksempel 16
5.5 Manuel håndtering af hukommelse 17

6 Implementation **19**
6.1 Split 19
6.2 Radix sort 21
6.3 Forholdet mellem pseudokode og kode 21

7 Implementation af biblioteket **23**
7.1 Copy 23
7.2 Map 23
7.3 Permute 23
7.4 Scan 24
7.5 Segmented scan 25
7.6 Optimering 25

8 Tidsforbrug **26**
8.1 Elementvise primitiver 26
8.2 Scan 27
8.3 Split 27
8.4 Radix sort 28

9 Analyse **29**
9.1 Forbedringer 29
9.2 Arbejdsbyrde 29
9.3 Vector-scan-modellen på CUDA 30

10 Konklusion **31**
10.1 Hastighed 31
10.1.1 Fremtidig fokus på optimeringer 31
10.2 Mulighed for arbitrære operatorer 31

A Data for tidsforbrug **32**

B Kildekode **32**
B.1 main.cpp 32
B.2 test.cu 32
B.3 kernel.cu 39
B.4 copy.h 43
B.5 split.h 43
B.6 radix.h 44
B.7 arrayprint.h 45
B.8 paracuda.h 45
B.9 paracuda_struct.h 46
B.10 paracuda_map.h 53
B.11 paracuda_permute.h 55

B.12 paracuda_scan.h 56
B.13 paracuda_segmentedScan.h 59
B.14 nvidia_scan.h 63
B.15 nvidia_kernel.cu 64
B.16 nvidia_large_kernel.cu 69
B.17 cpu_map.h 74
B.18 cpu_scan.h 75
B.19 cpu_split.h 75
B.20 cpu_radix.h 75

1 Introduktion

I de sidste par år er man ved at have nået en hardwaremæssig grænse for hvor hurtigt sekventiele programmer kan afvikles, og man er derfor begyndt at fokusere på at eksekvere programmerne parallelt. Dette har allerede længe været tilfældet for GPU'er (Graphical Prosessing Units), som typisk kommer med mange kerner, og som samtidigt ligger i en overskuelig prisklasse. Det er derfor interessant at undersøge hvordan man bedst udnytter sådanne processorer til generelle algoritmer.

Der findes algoritmer der er åbentlyst paralleliserbare (map), men også mindre åbentlyse algoritmer, som at summere værdierne i en vektor, kan gøres effektivt parallelt. Scan er generalisering af dette, der anvender en associativ operator på elementerne i vektoren, og efterlader præfix-summen op til hvert element på elementets position. En måde at implementere disse på er vha. vektor-scan modellen, som tillader skabelsen af mange primitiver der kan bruges til at udvikle generelle algoritmer, samtidigt med at de er designet til at kunne eksekveres i et paralelt miljø - som f.eks. NVIDIAs CUDA platform.

1.1 Mål

Vi vil i dette projekt undersøge om det kan betale sig at implementere vector-scan modellen på CUDA, med det formål at demonstrere en implementation af Scan på NVIDIAs CUDA platform. Vi vil i dette projekt desuden gøre det til vores mål at lære at programmere på CUDA platformen, samt at forstå scan-vector modellen, og videregive de erfaringer vi har gjort os - med særlig fokus på optimering.

For at give en kvantitativ indikator af hvor godt vores projekt har lykkedes, vil vi sammenligne vores egne implementationer med CUDA's implementation af scan, samt en sekventiel version af algoritmen.

2 CUDA

2.1 Introduktion

CUDA, som står for "Compute Unified Device Architecture", er en arkitektur udviklet af grafikkort producenten NVIDIA, der tillader programmører skrive generel purpose programmer der kan eksekveres på et NVIDIA grafikkort. Ideen er at udnytte grafikkortenes mange kerner til at udføre mange paralle udregninger. Alle programmer der skal eksekveres på CUDA skal skrives i "C for CUDA", som er en modificeret version af C, der indeholder de nødvendige datastrukturer og funktioner til at man kan skrive en funktion, en såkaldt kernel, der eksekveres parallelt på grafikkortet. Udvidelserne kan placeres i 3 primære kategorier:

Host: Funktioner der kan eksekveres på CPU'en og som kan tilgå GPU'en/GPU'erne.

Device: Funktioner der kun kan eksekveres på GPU'en, og som kun har relevans for kortet

Fælles: Indbyggede datastrukturer samt en delmængde af C's bibliotek, der både kan køre på CPU'en såvel som GPU'en.

2.2 Blokke og tråde

Et vigtigt begreb i CUDA er blokke og tråde. For at udnytte grafikkortets parallelle potentiale optimalt, bliver alle udregningerne på GPU'en udført i tråde. Videre er disse tråde inddelt i blokke for nem håndtering. Hver blok kan i CUDA 2.1 indeholde op til 512 tråde og kan inden for blokken dele delt hukommelse (se 2.4), samt synkronisere deres eksekvering. Den maksimale størrelse på en blok udgør derfor en naturlig størrelse at splitte større parallelle problemer op i, hvis udregningerne afhænger af tidligere resultater - se også 2.4. Blokke der er lige store bliver i så høj grad som muligt eksekveret parallelt, hvilket betyder at det, alt efter situationen, kan betale sig at allokere tråde i multipla af 512 (eller hvor meget der vælges at bruge pr. blok), og så sørge for at de ekstra tråde ikke foretager sig noget.

Modsat på CPU'en, vil tråde på GPU'en eksekvere hele kernelen. Hvis en blok støder på conditional statements (`if`, `if-else`, `switch`, etc.) vil alle veje igennem kernelen eksekveres - de tråde som ikke opfylder betingelsen bliver blot deaktiveret i den del af koden [Guide]. Det er derfor tilrådeligt at have så få conditional statements i koden som muligt, da alle branches skal gennemgås.

2.3 Kernel

Når en kernel køres, vil alle dens tråde arbejde på samme inddata, så den eneste måde at differentiere tråde på er vha. deres threadId og id'et på den blok de tilhører.

2.4 Hukommelse

CUDA skelner mellem hukommelse der er tilgængelig fra CPU'en ("host memory") og hukommelse der er tilgængelig fra GPU'en ("device memory"). Hvis man tilgår en forkert type hukommelse vil det enten resultere i segmentation faults eller udefineret data. Omkostningen ved at kopiere fra den ene til den anden er ikke ubetydeligt, og det kan derfor anbefales, at man prøver at undgå konstant at kopiere data frem og tilbage.

Når man skriver programmer på CUDA, er hukommelse noget af det vigtigste at optimere.

På GPU'en er de to vigtigste hukommelser den lille, men hurtige, delte hukommelse og den store, men langsomme globale hukommelse. For begge hukommelser tager det fire maskine cykler at skedulere at man vil gemme eller hente data - men for den globale hukommelse er der endvidere et overhead på 400-600 cykler hver gang det enten skal læses eller skrives. For at kunne udnytte CUDA ordentligt, er det derfor imperativt kun at tilgå den globale hukommelse så lidt som muligt, og derimod udnytte den delte hukommelse. Da denne hukommelse er lille og isoleret i hver blok, vil det, for algoritmer der skal arbejde på på store datamængder, være nødvendigt at finde en metode til at splitte algoritmen op i flere faser, og eventuelt til sidst lave en korrigering af data for at få et korrekt resultat.

Et problem der kan opstå når man bruger sekventiel hukommelse er bank conflicts. For at gøre hukommelses tilgangen i den delte hukommelse hurtigere, er den delte splittet op i flere lige store dele, kaldet banks, så hver bank kan tilgås af en tråd samtidigt. Hvis flere tråde prøver at tilgå den samme bank i hukommelsen, vil hardwaren være nød til at tilgå bank'en sekventielt. Et godt eksempel på at undgå dette bliver vist i implementeringen af Scan primitiven[1] i [Shubhabrata].

For videre information om CUDA henvises der til [Guide].

[1] Se 4, s. 7

3 Overvejelser

3.1 Tidspunkt for valg af operator

I vores projekt valgte vi at starte med at implementere Scan, da den både var overkommelig at implementere, samtidigt med at den dannede fundament for mange andre algoritmer - som Split og Quicksort.

Som nævnt tidligere i rapporten så virker Scan med alle associative binære operatorer, der sammen med et neutralt element danner en monoid. Derfor ville det være interessant at give brugeren af biblioteket muligheden at vælge, eller ligefrem implementere, den ønskede associative operator.

Et vigtigt spørgsmål, vi blev nød til at tage stilling til, var derfor på hvilket tidspunkt den nødvendige operator skulle vælges. Dette efterlader os med to valgmuligheder, enten på køretidspunktet eller overættelsestidspunktet:

- Muligheder på køretidspunktet:
  - At generere "PTX"assembler kode og lade driveren oversætte denne til NVIDIA maskinkode. Dette kræver dog at vi bruger driver-api'et som ikke er tilgængeligt i emulatoren.
  - At skrive en fortolker der kører i trådene og generere kode til denne.
- Muligheder på oversættelsestidpunktet:
  - At bruge makroer
  - At bruge klasser
  - At bruge templates

3.1.1 Run Time

Vi mente at PTX løsningen var urealistisk, da den ville krævede adgang til et grafikkort under hele udviklingsforløbet, samt at vi skulle sætte os ind i assembler-sproget, hvilket ville tage alt for lang tid. Desuden ville det at skrive en oversætter være et større projekt i sig selv, og ændre fokus for projektet.

Ligeledes mente vi at det ville være både for perifert til vores opgave, samt for tidskrævende at designe vores eget domæne specifikke sprog og skrive en fortolker til denne.

Da CUDA ikke understøtter funktionspointere i kernels antog vi at virtual tables og dermed virtuelle metoder var udelukket.

3.1.2 Compile time

Vi overvejede at bruge templates til at parameterisere algoritmerne med operatorer og datastrukturer, men vi kunne ikke finde dokumentation for hvor stor en delmængde af templates CUDA understøtter for kernels.

Vi valgte derfor at definere Scan (og senere Segmented Scan) vha. makroer, så brugeren kun blev nød til selv at definere navnet på scan

operationen, typen for ind- og output, og selve operatoren - der så vil blive sat sammen med et skelet for algoritmen. En klar fordel ved dette design er at der ikke på noget tidspunkt skal rodes med CUDA kode generering, da koden der kommer ind er ren CUDA kode, samtidigt med at vores API giver programmøren en stor frihed til at implementere sin egen operatorer og datastrukturer. Ulempen er så at udviklingstiden for os blev større og eliminering af fejl sværere, da makroer ikke har gode fejlbeskeder og giver et højt overhead i udviklingen.

4 Vector Scan Modelen

Vector Scan Modelen er en model som beskriver en række algoritmer og datastrukturer der kan bruges til parallel udregning. Modellen arbejder især med de såkaldte "Scan Primitives", som er en mængde lavniveaualgoritmer med et konceptuelt enkelt resultat, som tilsammen kan bruges til at skabe komplicerede algoritmer.

Vi har valgt at implementere følgende algoritmer fra [Blelloch] i kapitel 3:

Scan:

`Input:`

Værdi vektor: $[a_0, a_1, \ldots, a_{n-2}, a_{n-1}]$

Associativ binær operator: $\oplus$

Neutralt element: I - I findes ud fra både typen af værdierne i vektoren og operatoren. Hvis værdi vektoren indeholder heltal, og $\oplus$ var gange, så ville I være 1, men hvis $\oplus$ var addition, så ville I være 0. Sammen med den associative binære operation skal den danne en monoid.

`Beskrivelse:`
Givet værdivektoreren ovenfor vil Scan returnere en n-elements vektor med følgende indhold: [I, a_0, $a_0 \oplus a_1$, ..., $a_0 \oplus a_1 \oplus \ldots \oplus a_{n-3} \oplus a_{n-2}$]. I [Blelloch] sættes $\oplus$ kun til or, and, max, min og plus-scan, da disse er de eneste der anvendes i bogen, men også operatorer som gange ville kunne bruges. Beskrevet på s. 6 i [Blelloch].
`Eksempel:`
Givet vektoren [1, 2, 3, 4, 5] med det neutrale element 0, og $\oplus$ som addition giver [0, 1, 3, 6, 10]

Segmented Scan:

`Input:`

Værdi vektor: $[a_0, a_1, \ldots, a_{n-2}, a_{n-1}]$

Segmentation Vektor: $[b_0, b_1, \ldots, b_{n-2}, b_{n-1}]$, hvor $b_i \in \{T, F\}$ for alle $i \in \{0, 1, \ldots, n-2, n-1\}$.

Associativ binær operator: $\oplus$

Neutralt element: I

`Beskrivelse:`
Segmented Scan virker som Scan, med den forskel at den kan simulere flere vektorer i en enkelt vektor. Dette lader sig gøre vha. Segmentation vektoren, som indikerer hvornår en ny vektor begynder. Segmented Scan bruges når et enkelt fysisk vektor skal simulere flere logiske vektor, f.eks. hvis vektoren bliver splittet op undervejs i processen. Scan kan simuleres i segmented scan, ved at vidregive scan's værdivektor, og lade segmentation vektoren have et T i det 0'te element, og F i alle andre elementer. Algoritmen bliver beskrevet på s. 45 i [Blelloch].
`Eksempel:`
Givet Værdi vektoren [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] og segmention vektoren [T, F, F, F, T, F, F, F, F, T] vil outputtet af en plus-segmented-scan

(med det neutrale element 0) være [0, 1, 3, 6, 0, 4, 9, 15, 22, 0]

Permute:

`Input:`

Værdi vektor: $[a_0, a_1, \ldots, a_{n-2}, a_{n-1}]$

Adresse vektor: $[b_0, b_1, \ldots, b_{n-2}, b_{n-1}]$, hvor $b_i \in \{0, 1, 2, \ldots n-1\}$, for alle $i \in \{0, 1, \ldots, n-2, n-1\}$

`Beskrivelse:`
Hvis de 2 ovenstående vektorer, vil Permute returnere vektoren $[a_{b_1}, a_{b_2}, \ldots, a_{b_{n-2}}, a_{b_{n-1}}]$. Algoritmen bliver beskrevet på s. 40 i [Blelloch]
`Eksempel:`
Givet værdi vektoren [8, 6, 4, 1, 0] og adresse vektoren [2, 4, 0, 1, 3] returnerer Permute [4, 1, 8, 0, 6]

Enumerate:

`Input:`

Værdi vektor: $[a_0, a_1, \ldots, a_{n-2}, a_{n-1}]$, hvor $a_i \; in\{0, 1\}$ for $i \in \{0, 1, \ldots, n-2, n-1\}$

`Beskrivelse:`
Enumerate er et specialtilfældet af Scan, hvor alle værdierne i vektoren er enten 0 eller 1. Enumerate er værd at nævne separat, da den ofte bruges til at finde offsets, f.eks. i Split. Algoritmen bliver beskrevet på s. 42 i [Blelloch].
`Eksempel:`
Givet værdi vektoren [0, 1, 1, 0, 0, 0, 1, 1, 0] returnerer Enumerate [0, 0, 1, 2, 2, 2, 2, 3, 4]

Copy:

`Input:`

Vektor der skal skrives til: $[a_0, a_1, \ldots, a_{n-2}, a_{n-1}]$

Værdi der skal skrives til vektoren: b

`Beskrivelse:`
Skriver et givet værdi til alle pladser i den medfølgende vektor. Algoritmen bliver beskrevet på s. 42 i [Blelloch].
`Eksempel:`
Givet en vektor [5, 9, -7, 15] og tallet 6 returnerer den vektoren [6, 6, 6, 6]

Distribute:

`Input:`

Værdi vektor: $[a_0, a_1, \ldots, a_{n-2}, a_{n-1}]$

`Beskrivelse:`
Givet en værdi vektor vil den lave et scan over den[2], og kopierer derefter det sidste element ind på alle pladserne. Algoritmen bliver beskrevet

[2]Da vi kun er interesserede i det sidste element, i vektoren, kunne man godt nøjes med at lave en reduce, hvilket svarer til upsweep fasen beskrevet i [Harris]

på s. 42 i [Blelloch].
`Eksempel:`
For en additions distribute vil værdi vektoren [1, 8, 7, 2, 3] returnere [18, 18, 18, 18, 18]

Split:
`Input:`

Værdi vektor: $[a_0, a_1, \ldots, a_{n-2}, a_{n-1}]$

Binær vektor $[b_0, b_1, \ldots, b_{n-2}, b_{n-1}]$, hvor $b_i \in \{T, F\}$, for alle $i \in \{0, 1, \ldots, n-2, n-1\}$

`Beskrivelse:`
Givet en værdi vektor og en binær flag vektor, bliver en værdi vektor returneret, hvor a_i hvor b_i er F bliver placeret til venstre i vektoren, og alle de a_k hvor b_k er T bliver placeret i den højre del af vektoren. Værdierne bliver splittet stable, så hvis $b_i = b_k$, hvor $i < k$ så vil a_i fortsat være placeret før i vektoren end a_k. Se s. i [Blelloch].
`Eksempel:`
Hvis man har værdi vektoren [0, 1, 2, 3, 4, 5, 6, 7] og den binære vektor [T, F, T, F, T, F, T, F] så vil Split returnere [1, 3, 5, 7, 0, 2, 4, 6].

For at gøre vores arbejde lettere, har vi desuden tilføjet algoritmen Map, som udfører en bestemt operation på alle elementerne, selvom den ikke eksplicit bliver nævnt.

4.1 Paralleliserbare algoritmer

Alle de ovenstående primitiver er paralleliserbare. Dette lader sig gøre idet både Scan[3], Segmented Scan[4] og Map, er paralleliserbare, og fordi de ovenstående algoritmer kan bygges op af Scan, Segmented Scan og Map operationer.

Algoritmer som direkte bygger på Scan, Segmented Scan og Map:

- Copy[5]
- Enumerate
- Split
- Permute

Algoritmer som bygger på andre algoritmer:

Distribute: Bygger oven på Copy

For en mere udførlig liste af paralleliserbare algoritmer - se [Blelloch] s. 36, som også fremhæver mulige anvendelser af de forskellige primitiver.

[3] Se [Harris] for algoritme
[4] Se [Shubhabrata]
[5] I vores bibliotek er den implementeret anderledes, se afsnit 7

4.2 Implementation af primitiver i forhold til modellen

4.2.1 Scan

Udover en direkte implementation af Scan som beskrevet i [Blelloch] er det også praktisk at scan også returnerer "summen", altså $a_0 \oplus a_1 \oplus a_2 \ldots \oplus a_{n-2} \oplus a_{n-1}$, da denne ofte er nødvendig i de algoritmer der bygger oven på scan.

4.2.2 Segmented Scan

Ligesom med Scan har vi fundet det praktisk at gemme slutresultatet af hvert scan af en logisk vektor, for at sikre at segmenterede implementationer, f.eks. af Split, kan give de rigtige offset. Med Segmented Scan bliver det dog nødvendigt at gemme en vektor i stedet for blot en sum, da der er tale om flere logiske vektorer, som skal have resultatet af scannet.

4.2.3 Copy

[Blelloch] beskriver copy som en funktion der fylder alle pladser i en vektor med det neutrale element (for en plus-scan), og indsætter det kopierede element på den første plads. Derefter udføres et plus-scan på arrayet, og det kopierede element indsættes på den første plads igen (da den er blevet overskrevet af det neutrale element). Selvom denne metode er lige til, så er den ikke særlig effektiv. I vores implementation har vi derfor indført en copy kernel, som kopierer det ønskede element ind på hver position i vektoren. Grunden til at dette ikke blot blev gjort med en map operation, består i at vores model kræver at man statisk har fastlagt operatoren for map, og man kan derfor ikke få operatoren til at returnere en værdi der først kendes under afviklingen af programmet.

4.2.4 Distribute

Den eneste forskel på den version af distribute som er beskrevet i [Blelloch] og vores implementation, er at i implementationen bliver værdien taget direkte fra vektoren (alt efter om det skal være forwards eller backwards distribute) og givet videre til vores Copy (beskrevet ovenfor), hvor man i [Blelloch] bruger forskellige Copy funktioner alt efter om man skal lave forwards eller backwards distribute.

4.2.5 Split

Der er ikke den store forskel imellem modelen fundet i [Blelloch] og vores implementation, men der er visse detaljer som kan ses i 6.

4.3 Anvendelsesmuligheder

Udover de direkte anvendelsesmuligheder af primitiverne, er det muligt at implementere mere avancerede algoritmer som f.eks. Radix[6] og Quicksort[7], samt algoritmer som Line-of-Sight eller Line Drawing[8]. Vi har implementeret en split-radix-sort som udnytter vores implementerede primitiver - se s. 21 i 6.2. Videre kan man forstille sig at de fleste algoritmer som har brug for at summere over vektorer, eller lave en binær sortering vil kunne drage nytte af primitiverne.

[6]Se s. 43 i [Blelloch]
[7]Se s. 43 og s. 46 i [Blelloch]
[8]Se s. 40 og s. 50 i [Blelloch]

5 Bibliotek

Biblioteket implementerer en række funktioner bygget på vector-scan-modellen. Meningen med biblioteket er at man kan bruge det fra C uden at lære CUDA-bibliotekerne at kende. Dog er der visse begrænsninger på operatorerne som er CUDA-specifikke, og det er ligeledes nødvendigt at kunne oversætte med CUDA-oversætteren.

Typisk brug af biblioteket består i at definere en datastruktur, en operator og en algoritme der bruger disse. Til hver definition bruges en makro fra biblioteket. Hver algoritme-makro definerer en normal funktion, der kan kaldes som enhver anden C-funktion.

Hver algoritme svarer til en primitiv funktion i [Blelloch], dog med vilkårlige brugerdefinerede operatorer og vektorelementtyper.

5.1 Datastrukturer

Alle algoritmerne kræver at vi kan kopiere elementer ind og ud af vektorer. Vi repræsenterer en vektor af datastrukturer som en struct af arrays, da dette giver os mulighed for at behandle hvert felt i datastrukturen som en individuel vektor uden at skulle løbe hele vektoren igennem og kopiere det enkelte felt over. For at kunne hive et element ud fra en position genererer vi via makroen `PARACUDA_STRUCT_N` kode til at hive et enkelt element ud og samle det til en struct, og til at splitte et enkelt element op og sætte det ind igen.

Hvis det ønskes kan man også repræsentere vektorer som en array af structs ved at definere en normal struct og generere kode med `PARACUDA_SINGLE`.

Internt laver vi et alias til vektor-typen med `typedef PARACUDA_name_struct` hvor `name` er det brugerdefinerede navn for datastrukturen. Dermed kan vi slå vektor-typen op hvis vi har navnet for den ikke-vektoriserede datastruktur, så brugeren kun behøver at specificere denne.

Funktionerne der bliver genereret er navngivet på samme måde og er dækket i afsnit 5.5. Implementationen kalder CUDA-specifikke hukommelsesfunktioner for hvert felt i datastrukturen.

`PARACUDA_STRUCT_2(NAME, VECTOR, T0, X0, T1, X1)` genererer en `struct NAME { T0 X0; T1 X1; };` samt en `struct VECTOR { T0* X0; T1* X1; };`. Den førstenævnte bruges til at håndtere et enkelt element i vektoren, mens den anden bruges til at håndtere hele vektoren, hvor hvert felt har sin egen array. Der dannes desuden `typedef`s så de kan bruges i C uden at skulle skrive `struct` foran.

`PARACUDA_SINGLE(TYPE)` genererer funktioner for det angivne typenavn. Hvis typen består af mere end en token skal der bruges en `typedef`, således at parameteren er en enkelt token. Det samme navn skal bruges når den endelige algoritme defineres, så makroen kan finde de tilknyttede funktioner.

5.2 Operatorer

Det vi kalder en operator i vores bibliotek er typisk en funktion der læser fra et eller flere elementer og skriver til et resultatelement. Det ene-

ste generelle krav er dog at den er defineret med operator-makroen og overholder CUDAs specifikationer for kode der skal køre på grafikkortet (altså ingen rekursion, funktions-pointere eller kald til normale funktioner). Operator-makroen genererer to versioner af den funktion man specificerer - en der kan køre på værtssystemet og en der kan køre på grafikkortet. Sidstenævnte funktion navngives `PARACUDA_name_operator` hvor `name` er navnet på operatoren.

`PARACUDA_OPERATOR(NAME, RETURN, ARGUMENTS, BODY)` definerer en normal funktion og en funktion der kan køre på grafikkortet. `NAME` er funktionens navn, `RETURN` er dens returtype (typisk `void`), `ARGUMENTS` er argumenterne (i parentes) og `BODY` er funktionskroppen omkranset af paranteser yderst og tuborgklammer inderst (se eksemplet i afsnit 5.4).

Operatorer kan hverken bruge rekursion eller funktionspointere, idet disse ikke er understøttet af CUDA. Desuden kan normale funktioner heller ikke kaldes (kun dem der kan kaldes fra en CUDA `__device__` funktion, se eventuelt CUDAs dokumentation).

5.3 Algoritmer

De understøttede algoritmer er herunder angivet med navn, beskrivelse og parametre.

`PARACUDA_COPY (NAME, TYPE)`

Definerer en funktion der fylder en vektor af længde l med en værdi.

$$\text{copy}(v, l) = [v, \ldots, v]$$

`NAME`	bliver navnet på den genererede funktion.
`TYPE`	er typen på værdien.

```
TYPE-VECTOR* NAME(
        TYPE-VECTOR* out,
        TYPE in,
        size_t length,
        size_t thread_count);
```

Hvor `TYPE-VECTOR` svarer til vektor-typen for `TYPE`, og `NAME` er det angivne navn. Hvis funktionen kaldes med `out=NULL` allokeres hukommelsen for `out` automatisk inde i funktionen, og ellers antages det at der allerede er allokeret den nødvendige hukommelse. Automatisk allokeret hukommelse vil ikke automatisk blive deallokeret. Antallet af elementer i inddata angives som `length`. Returværdien er en pointer til vektoren med resultatet (`out` med mindre `out=NULL`). Den sidste parameter er antallet af tråde, og er normalt `PARACUDA_MAX_THREADS`. Det er tilladt at `out=in` hvis de har samme type.

PARACUDA_MAP (NAME, OPERATOR, OUT, IN)

Definerer en funktion der implementerer map-algoritmen, som anvender en operator på hvert element.

$$\text{map}(f, [a_0, a_1, \ldots, a_{n-1}]) = [f(a_0), f(a_1), \ldots, f(a_{n-1})]$$

`NAME`	bliver navnet på den genererede funktion.
`OPERATOR`	er den operator som skal anvendes på hvert element. Den skal tage argumenterne `(OUT* out, IN* in)`.
`OUT`	er typen på uddata.
`IN`	er typen på inddata.

Den genererede funktion kan kaldes som enhver anden funktion og har signaturen

```
OUT-VECTOR* NAME(
        OUT-VECTOR* out,
        IN-VECTOR* in,
        size_t length,
        size_t thread_count);
```

Hvor `OUT-VECTOR` og `IN-VECTOR` svarer til vektor-typerne for henholdsvis `OUT` og `IN`, og `NAME` er det angivne navn. Hvis funktionen kaldes med `out=NULL` allokeres hukommelsen for `out` automatisk inde i funktionen, og ellers antages det at der allerede er allokeret den nødvendige hukommelse. Automatisk allokeret hukommelse vil ikke automatisk blive deallokeret. Antallet af elementer i inddata angives som `length`. Returværdien er en pointer til vektoren med resultatet (`out` med mindre `out=NULL`). Den sidste parameter er antallet af tråde, og er normalt `PARACUDA_MAX_THREADS`. Det er tilladt at `out=in` hvis de har samme type.

PARACUDA_SCAN (NAME, OPERATOR, NEUTRAL, TYPE)

Definerer en funktion der implementerer scan-algoritmen (exclusive prefix sum), som er i familie med fold.

$$\text{scan}(\oplus, z, [a_0, a_1, \ldots, a_{n-1}]) = [z, a_0, a_0 \oplus a_1, \ldots, a_0 \oplus a_1 \oplus \cdots \oplus a_{n-2}]$$

`NAME`	bliver navnet på den genererede funktion.
`OPERATOR`	er den operator der skal anvendes på elementpar. Det er et krav at operatoren er associativ, så det er ligegyldigt hvor man sætter paranteserne: $(a \oplus b) \oplus c = a \oplus (b \oplus c)$, og at den sammen med det neutrale element (og typen) danner en monoid. Den skal tage argumenterne `(TYPE* result, TYPE* left, TYPE* right)`.
`NEUTRAL`	er en operator der returnerer det neutrale element. Den skal tage argumenterne `(TYPE* result)`.
`TYPE`	er typen på ud- og inddata.

Den genererede funktion kan kaldes som enhver anden funktion og har signaturen

```
TYPE-VECTOR* NAME(
        TYPE* sum,
        TYPE-VECTOR* out,
        TYPE-VECTOR* in,
        size_t length,
        size_t thread_count);
```

Hvor `TYPE-VECTOR` svarer til vektor-typen for `TYPE`, og `NAME` er det angivne navn. Hvis funktionen kaldes med `out=NULL` allokeres hukommelsen for `out` automatisk inde i funktionen, og ellers antages det at der allerede er allokeret den nødvendige hukommelse. Automatisk allokeret hukommelse vil ikke automatisk blive deallokeret. Antallet af elementer i inddata angives som `length`. Hvis `sum` er forskellig fra `NULL` lagres summen af alle elementerne i denne: $a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1}$. Returværdien er en pointer til vektoren med resultatet (`out` med mindre `out=NULL`). Den sidste parameter er antallet af tråde, og er normalt `PARACUDA_MAX_THREADS`. Det er tilladt at `out=in`.

PARACUDA_SEGMENTEDSCAN (NAME, OPERATOR, NEUTRAL, TYPE)

Definerer en funktion der implementerer segmented scan. Hvert sat flag indikerer starten på en ny delvektor.

$$\text{scan}(\oplus, z, [a_0, a_1, a_3, a_4, a_5, a_6], [0, 0, 0, 1, 0, 0]) = [z, a_0, a_0 \oplus a_1, z, a_3, a_3 \oplus a_4]$$

Den genererede funktion kan kaldes som enhver anden funktion og har signaturen

```
TYPE-VECTOR* NAME(
        TYPE-VECTOR* out,
        TYPE-VECTOR* in,
        int* flags,
        size_t length,
        size_t thread_count);
```

Selvom der ikke er nogen entydig sum i Segmented Scan, så skal resultaterne fra de enkelte logiske arrays stadigvæk gemmes i en vektor til senere brug - f.eks. hvis Segmented Split skulle implementeres. Udover dette, flagene og opdelingen i delvektorer er den identisk med `PARACUDA_SCAN`.

PARACUDA_PERMUTE (NAME, TYPE)

Definerer en funktion der returnerer en given permutation af en vektor.

$$\text{permute}([a_0, a_1, \ldots, a_{n-1}], [i_0, i_1, \ldots, i_{n-1}]) = [a_{i_0}, a_{i_1}, \ldots, a_{i_{n-1}}]$$

`NAME` bliver navnet på den genererede funktion.
`TYPE` er typen på ud- og inddata.

Den genererede funktion kan kaldes som enhver anden funktion og har signaturen

```
TYPE-VECTOR* NAME(
        TYPE-VECTOR* out,
        TYPE-VECTOR* in,
        int* positions,
        size_t length,
        size_t thread_count);
```

Hvor `TYPE-VECTOR` svarer til vektor-typen for `TYPE`, og `NAME` er det angivne navn. Hvis funktionen kaldes med `out=NULL` allokeres hukommelsen for `out` automatisk inde i funktionen, og ellers antages det at der allerede er allokeret den nødvendige hukommelse. Automatisk allokeret hukommelse vil ikke automatisk blive deallokeret. Antallet af elementer i inddata angives som `length`. Permutationen angives som en vektor af positioner `positions`, og det antages at den samme position ikke optræder flere gange. Returværdien er en pointer til vektoren med resultatet (`out` med mindre `out=NULL`). Den sidste parameter er antallet af tråde, og er normalt `PARACUDA_MAX_THREADS`. Det er tilladt at `out=in`.

5.4 Eksempel

Lad os sige vi har brug for at definere følgende funktion:

plus_times_map $[(a_0, b_0), (a_1, b_1), \ldots] = [(a_0{+}b_0, a_0{\times}b_0), (a_1{+}b_1, a_1{\times}b_1), \ldots]$

Dette kræver en par-datastruktur, en $+\times$-operator og map-algoritmen.

```
#include "paracuda.h"

PARACUDA_STRUCT_2(pair_t, pair_vector_t,
    int, x,
    int, y
)

PARACUDA_OPERATOR(plus_times, void, (pair_t* r, pair_t* v), ({
    r->x = v->x + v->y;
    r->y = v->x * v->y;
}))

PARACUDA_MAP(plus_times_map, plus_times, pair_t, pair_t)
```

Ovenstående skal gemmes i en `.cu` (CUDA) fil og skal oversættes med cuda-oversætteren.

Den genererede funktion `plus_times_map` kan kaldes som en normal funktion og har følgende signatur:

```
pair_vector_t* plus_times_map(
        pair_vector_t* out,
        pair_vector_t* in,
        size_t length,
        size_t thread_count);
```

5.5 Manuel håndtering af hukommelse

Alle funktioner der genereres med algoritme-makroer kopierer først vektorerne ned i grafikkortets hukommelse, og derefter tilbage igen når udregningen er færdig. Man kan undgå dette ved at bruge følgende funktioner, hvor `T` er navnet på datastrukturen og `F` er navnet på funktionen.

Det er ikke tilladt at tilgå hukommelse der ligger på grafikkortet uden disse funktioner. Overholdes dette ikke er programmets opførsel ikke veldefineret, men en typisk fejlmeddelelse er `segmentation fault`.

`PARACUDA_T_allocate_host (length)`

Denne funktion allokerer og returnerer en vektor af typen `T` med længden `length` i værtssystemet.

`PARACUDA_T_allocate_device (length)`

Denne funktion allokerer og returnerer en vektor af typen `T` med længden `length` på grafikkortet.

`PARACUDA_T_copy_host_device (out, in, length)`

Denne funktion kopierer en vektor fra værtssystemet til grafikkortet.

`PARACUDA_T_copy_device_host (out, in, length)`

Denne funktion kopierer en vektor fra grafikkortet til værtssystemet.

`PARACUDA_T_copy_device_device (out, in, length)`

Denne funktion kopierer en vektor fra et sted på grafikkortet til et andet.

`PARACUDA_T_from_vector (out, in, index)`

Kopierer et element ud af en vektor og ind i `out`, der er af typen `T`.

`PARACUDA_T_to_vector (out, in, index)`

Kopierer et element ind i en vektor fra `in`, der er af typen `T`.

`PARACUDA_T_shallow_allocate_host (out, in, index)`

Allokerer en vektor på værtssystemet uden at allokere hukommelse til indholdet.

`PARACUDA_T_shallow_allocate_device (out, in, index)`

Allokerer en vektor på grafikkortet uden at allokere hukommelse til indholdet.

`PARACUDA_T_shallow_copy_host_device (out, in, index)`

Kopierer en vektor til grafikkortet under den antagelse at dens pointere allerede peger på data på grafikkortet. Indholdet bliver ikke kopieret.

`PARACUDA_T_shallow_copy_device_host (out, in, index)`

Kopierer en vektor til værtsystemet uden at kopiere indholdet, så dens pointere stadig peger på data på grafikkortet.

`PARACUDA_F_run (...)`

Denne funktion kalder funktionen `F` under den antagelse at alle vektorer allerede ligger på grafikkortet. Resultatet skal manuelt kopieres tilbage til værtssystemet. Hukommelse til uddata allokeres ikke automatisk. Der er små variationer i funktionssignaturen i forhold til `F`. Disse ses bedst i kildekoden for den individuelle `run`-funktion og dens tilknyttede kommentar.

6 Implementation

6.1 Split

Vi har implementeret split oven på vores bibliotek (altså uden at bruge CUDA direkte). Vores implementation svarer til det følgende:

$$
\begin{aligned}
&\text{split}(f, v) = \\
&\texttt{let } n = \text{map}(\neg, f) \texttt{ in} \\
&\texttt{let } (l, sum) = \text{scan}(+, 0, n) \texttt{ in} \\
&\texttt{let } c = \text{copy}(sum, \#v) \texttt{ in} \\
&\texttt{let } (t, \_) = \text{scan}(+, 0, f) \texttt{ in} \\
&\texttt{let } r = \text{map}(+, \text{zip}(c, t)) \texttt{ in} \\
&\texttt{let } g = \lambda(a, b, c).\ \texttt{if } a \texttt{ then } c \texttt{ else } b \texttt{ in} \\
&\texttt{let } p = \text{map}(g, \text{zip}(f, l, r)) \texttt{ in} \\
&\text{permute}(p, v)
\end{aligned}
$$

Hvor zip foretages ved at pege pointerene i en tuppel hen på de relevante arrays. $\#v$ er antallet af elementer i v. Derudover foretages der allokering og deallokering af hukommelse. Koden for delalgoritmerne, datastrukturerne og operatorerne er:

```
PARACUDA_SINGLE(int)

PARACUDA_OPERATOR(plus, void, (int* r, int* a, int* b), ({
    *r = *a + *b;
}))

PARACUDA_OPERATOR(zero, void, (int* r), ({
    *r = 0;
}))

PARACUDA_SCAN(plus_scan, plus, zero, int)

PARACUDA_OPERATOR(negate, void, (int* r, int* a), ({
    *r = !*a;
}))

PARACUDA_MAP(negate_map, negate, int, int)

PARACUDA_STRUCT_3(split_t, split_vector_t,
    int, flags,
    int, left,
    int, right
)

PARACUDA_OPERATOR(split, void, (int* r, split_t* a), ({
    *r = (a->flags) ? a->right : a->left;
}))

PARACUDA_MAP(split_map, split, int, split_t)
```

```
PARACUDA_PERMUTE(int_permute, int)
```

Den endelige funktion bliver så (variabelnavnene kan være lidt kryptiske her eftersom vi genbruger vektorene):

```
void split(int* array, int* flags, int length)
{
  int* posDown      = PARACUDA_int_allocate_device(length);
  int* posUp        = PARACUDA_int_allocate_device(length);
  int* positions    = PARACUDA_int_allocate_device(length);
  pair_vector_t* pair = PARACUDA_pair_t_shallow_allocate_device();
  split_vector_t* input = PARACUDA_split_t_shallow_allocate_device();

  int before;
  int computed_sum;
  PARACUDA_negate_map_run(posDown, flags, length, PARACUDA_MAX_THREADS);

  PARACUDA_int_peek(&before, posDown, length - 1);
  PARACUDA_plus_scan_run(0, posDown, length, PARACUDA_MAX_THREADS);
  PARACUDA_int_peek(&computed_sum, posDown, length - 1);
  computed_sum += before;

  PARACUDA_int_copy_run(positions, computed_sum, length, PARACUDA_MAX_THREADS);

  PARACUDA_int_copy_device_device(posUp, flags, length);
  PARACUDA_plus_scan_run(0, posUp, length, PARACUDA_MAX_THREADS);

  pair_vector_t host_pair;
  host_pair.x = positions;
  host_pair.y = posUp;
  PARACUDA_pair_t_shallow_copy_host_device(pair, &host_pair);

  PARACUDA_map_add_run(posUp, pair, length, PARACUDA_MAX_THREADS);

  split_vector_t host_input;
  host_input.flags = flags;
  host_input.left = posDown;
  host_input.right = posUp;
  PARACUDA_split_t_shallow_copy_host_device(input, &host_input);

  PARACUDA_split_map_run(positions, input, length, PARACUDA_MAX_THREADS);

  PARACUDA_int_permute_run(posDown, array, positions, length, PARACUDA_MAX_THREADS);

  PARACUDA_int_copy_device_device(array, posDown, length);

  PARACUDA_pair_t_shallow_free_device(pair);
  PARACUDA_split_t_shallow_free_device(input);
  PARACUDA_int_free_device(posDown);
```

```
  PARACUDA_int_free_device(posUp);
  PARACUDA_int_free_device(positions);
}
```

Kildekoden kan iøvrigt findes i bilag B.5, og datastruktur-, algoritme- og operator-definitionerne kan findes i B.3.

6.2 Radix sort

Vi har implementeret radix som beskrevet i [Blelloch], afsnit 3.4.

$$
\begin{aligned}
&\text{step}(i, v) = \\
&\texttt{let } n = \text{copy}(2^i, \#v) \texttt{ in} \\
&\texttt{let } f = \text{map}((\lambda(x, y).\ x \,\&\, y), \text{zip}(n, v)) \texttt{ in} \\
&\text{split}(f, v)
\end{aligned}
$$

Hvor zip foretages på samme måde som i split, og & er bitvis eller. Vi gentager step 32 gange med i fra 0 til 31, ved hele tiden at anvende step på resultatet af det foregående skridt.

```
void radix(int* array, int length, int max_threads)
{
  int* t_numbers = PARACUDA_int_allocate_device(length);
  int* t_flags = PARACUDA_int_allocate_device(length);
  bitwise_vector_t* in = PARACUDA_bitwise_t_shallow_allocate_device();

  for(int i = 0; i < 32; ++i) {
    int num = (1 << i);
    PARACUDA_int_copy_run(t_numbers, num, length, max_threads);

    bitwise_vector_t input;
    input.number = t_numbers;
    input.integer = array;
    PARACUDA_bitwise_t_shallow_copy_host_device(in, &input);

    PARACUDA_int_bitwise_map_run(t_flags, in, length, max_threads);

    split(array, t_flags, length);
  }
  PARACUDA_bitwise_t_shallow_free_device(in);
  PARACUDA_int_free_device(t_numbers);
  PARACUDA_int_free_device(t_flags);
}
```

Kildekoden kan iøvrigt findes i bilag B.6, og datastruktur-, algoritme- og operator-definitionerne kan findes i B.3.

6.3 Forholdet mellem pseudokode og kode

Sammenligner man pseudo-koden med kildekoden svarer hver linje i pseudo koden til få linjer C kode - hvis man ser bort fra allokeringen og

den separate definitioner af datastrukturer, operatorer og algoritmer, som alligevel skal gøres lige meget hvilken implementation man bruger.

Det skulle derfor være ganske lige til at omskrive parallel-vektor-pseudokode til C-kode oven på vores bibliotek.

7 Implementation af biblioteket

Algoritmerne er implementerede som makroer der genererer funktioner. Disse funktioner har tre formål: at håndtere kopiering af hukommelse frem og tilbage fra grafikkortet, at håndtere de små sekventielle dele af hver algoritme og at implementere den parallele del af algoritmen.

Den paralelle del er defineret som en såkaldt kernel, der er en C-funktion med CUDA-specifikke tilføjelser og begrænsninger (hverken rekursion, kald til normale funktioner eller funktions-pointere er tilladt, se [Guide]). Når en kernel startes kører alle tråde den samme funktion parallelt og med samme inddata, bortset fra et koordinat der fortæller hvilken tråd man er i.

Genererede funktioner navngives `PARACUDA_NAME_subname` hvor `NAME` er navnet på algoritmefunktionen og `subname` beskriver funktionens ansvar.

7.1 Copy

Denne algoritme producerer en vektor hvor alle elementerne er sat til en given værdi.

Hver tråd regner ud hvilken position i vektoren den repræsenterer ved `blockIdx.x * blockDim.x + threadIdx.x`, altså bloknummeret gange blokstørrelsen plus trådnummeret. Den læser så værdien ind fra hukommelsen og skriver den ind på den beregnede position i vektoren.

Vi starter så nok blokke med nok tråde til at hvert element har sin egen tråd. Dette giver en begrænsning på antallet af elementer, da det således skal være deleligt med antallet af tråde i en blok og højest være antallet af blokke gange antallet af tråde per blok.

Vi omgår den første begrænsning ved at tjekke om positionen er mindre end længden før der skrives til vektoren.

Maksimummet kunne omgås ved at give hver tråd flere elementer at arbejde på, eller ved at dele kørslerne op så hver kørsel arbejdede på en del af vektoren svarende til det nuværende maksimum. Dette er dog ikke implementeret.

Kildekoden kan findes i bilag B.4.

7.2 Map

Denne algoritme anvender en funktion på hvert element i en vektor.

Den eneste forskel fra copy er at den læser et element ind fra en anden vektor og anvender en funktion på det, istedet for at alle tråde læser den samme værdi.

Kildekoden kan findes i bilag B.10.

7.3 Permute

Denne algoritme returnerer en permutation af en vektor.

Hver tråd tager det element ud af vektoren der svarer til trådens koordinat, slår op i positionsvektoren med samme index for at få målpositionen, hvor elementet puttes ind i resultatvektoren.

Det antages at ingen position forekommer to gange. Hvis den gør er det en race condition, hvor det afhænger af trådskeduleringen hvilket element og hvor meget af det der havner der. Desuden vil der så være tilsvarende positioner der ikke er dækket, hvilket betyder at visse elementer slet ikke har fået tildelt en værdi, hvorfor de kan være hvad som helst.

Den har samme begrænsninger som copy.

Kildekoden kan findes i bilag B.11.

7.4 Scan

Vi har delt denne algoritme op i to kernels, svarende til upsweep og downsweep fra [Shubhabrata]. Den rekursive definition kan se sådan ud:

$$\begin{aligned}&\text{upsweep}(v, \oplus) = \texttt{if } \#v = 1 \texttt{ then } v \texttt{ else}\\&\texttt{let } m = \lambda(a, b).\ [a_0, \ldots, a_{\#a-1}, b_0, \ldots, b_{\#b-2}, a_{\#a-1} \oplus b_{\#b-1}] \texttt{ in}\\&m(\text{upsweep}([v_0, \ldots, v_{\#v/2-1}]), \text{upsweep}([v_{\#v/2}, \ldots, v_{\#v-1}]))\end{aligned}$$

I vores implementation er denne del af algoritmen en separat kernel. Den består af en løkke som går fra skridtet efter det trivielle basistilfælde, hvor længden af vektoren er 2 og opefter indtil længden af vektoren er den samme som inddata. Den samler hele tiden vektorerne og overskriver det sidste element i den nye vektor med summen af det sidste element fra de to vektorer.

For hvert skridt er der brug for færre tråde, og vi beregner antallet af påkrævede tråde og tjekker at trådens nummer er mindre end dette. Hver tråd kan have flere opgaver, og vi bruger i dette tilfælde en indre løkke til at udføre alle opgaverne sekventielt inde i tråden.

$$\begin{aligned}&\text{downsweep}(v, \oplus) = \texttt{if } \#v = 1 \texttt{ then } v \texttt{ else}\\&\texttt{let } l = \text{downsweep}([v_0, \ldots, v_{\#v/2-2}, v_{\#v-1}]) \texttt{ in}\\&\texttt{let } r = \text{downsweep}([v_{\#v/2}, \ldots, v_{\#v-2}, v_{\#v/2-1} \oplus v_{\#v-1}]) \texttt{ in}\\&[l_0, \ldots, l_{\#l-1}, r_0, \ldots, r_{\#r-1}]\end{aligned}$$

Denne del af algoritmen er også en kernel. Den består af en løkke som går fra længden af inddata ned til skridtet lige før basistilfældet. Hver iteration samles splittes delvektorerne op, mens summen af det sidste element i hver delvektor skrives ind i det sidste element af den højre delvektor, det oprindelige sidste element i den højre delvektor skrives ind i det sidste element af den venstre delvektor.

Opgaverne og antallet af arbejdende tråde behandles som i upsweep.

$$\begin{aligned}&\text{scan}(v, \oplus, z) =\\&\texttt{let } w = \text{upsweep}(v, \oplus) \texttt{ in}\\&\text{downsweep}([w_0, \ldots, w_{\#w-2}, z], \oplus)\end{aligned}$$

Scan består så af et kald til opsweep efterfulgt af et kald til downsweep. Mellem de to kald overskrives det sidste element (som efter upsweep er summen af vektorerne) af det neutrale element.

Den maksimale længde den kan arbejde på er det maksimale antal tråde pr. blok gange det maksimale antal blokke, da hvert element får tildelt sin egen tråd. Desuden oplever vi en segmentation fault i denne algoritme for inddata af størrelse 2^24 og derover som vi ikke har kunnet finde kilden til.

Kildekoden kan findes i bilag B.12.

7.5 Segmented scan

Implementationen af segmented scan er meget lig implementationen af scan, bortset fra at den også arbejder på en vektor af flag, og med følgende forskelle:

I upsweep overskrives sidste element af højre delvektor kun af summen hvis flaget er sat. Flaget på denne position overskrives derimod altid af den logiske sum af de sidste flag i de to delvektorer.

I downsweep er der tre tilfælde: enten er det originale flag på første plads i anden delvektor sat, og det sidste element i den høre delvektor bliver overskrevet med det neutrale element; eller det opdaterede flag på sidste plads i første delvektor er sat, og det sidste element i den anden delvektor overskrives med sidste element i første delvektor; og ellers overskrives det sidste element i den anden delvektor med summen som i scan. I alle tilfældene sættes flaget på sidste position i første delvektor til nul.

Implementationen af denne algoritme har de samme begrænsninger som scan.

Kildekoden kan findes i bilag B.13.

7.6 Optimering

Vi udnytter i vores implementation at alle funktioner der tager pointere som argumenter bliver inlinede [Guide] når de kaldes fra en kernel. Vi antager at de pointer-argumenter som operatorerne har bliver elimineret af NVIDIAs compiler, så værdierne om nødvendigt kan lægges i registre, og der derfor ikke er nogen omkostning ved denne model.

8 Tidsforbrug

Vi har målt tidsforbruget for hver af de primitiver vi har implementeret og de funktioner vi har bygget ovenpå, ved at køre dem på en række vektorer af kvadratisk voksende størrelser, og tage gennemsnittet af ti kørsler for hver størrelse. Tallene kan ses i bilag A.

De funktioner der er bygget ovenpå scan, har vi målt både med vores egen implementation af scan, og med NVIDIAs implementation, som vi har modificeret til at bruge `int` istedet for `float`. Den modificerede kode kan ses i bilag B.14, B.15 og B.16.

Vi har hverken medtaget den indledende allokering af hukommelse eller den efterfølgende frigørelse i vores målinger. Ind- og uddata ligger på grafikkortet. På den måde kan vi få de mest sigende tal for mindre vektorer.

Vi har desuden implementeret CPU-versioner af funktionerne og målt tidsforbruget for dem for at have en reference. Funktionerne ganske simple, og CPU-versionen af radix-sort arbejder på samme måde som GPGPU-versionen med et bit ad gangen. Koden for disse implementationer kan ses i bilag B.17, B.18, B.19 og B.20.

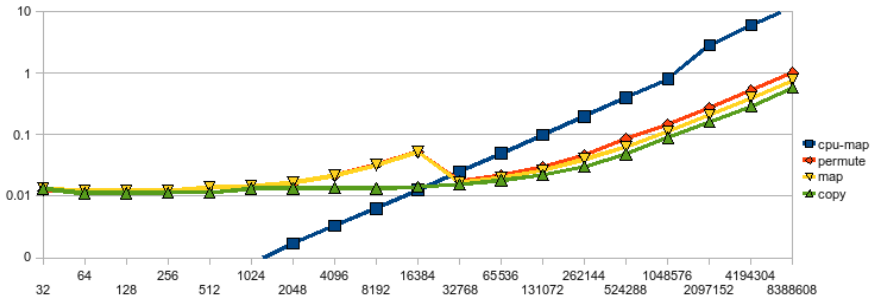
Koden for målingerne kan findes i bilag B.2.

Vi har målt op til 2^{23} elementer. Vores scan understøtter ikke flere, og NVIDIAs scan understøtter også kun op til 2^{24}.

Vi har målt på et grafikkort af typen NVIDIA GeForce GTX 260, og et værtssystem af typen Intel Core2 Quad CPU @ 2,66 GHz.

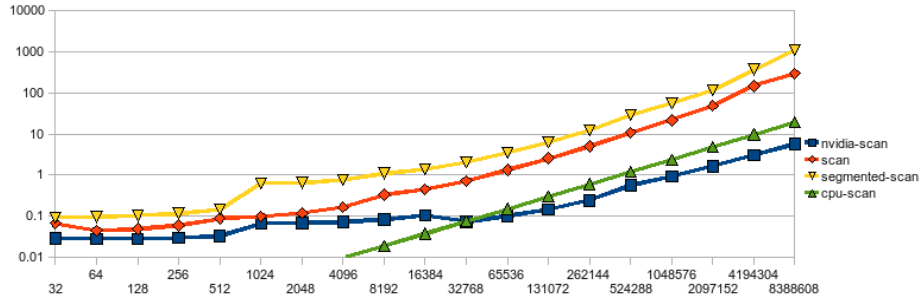
Alle diagrammerne i dette afsnit viser hvor lang tid (i millisekunder ud af y-aksen) det tager at køre en algoritme på vektorer af forskellige størrelser (antal elementer ud af x-aksen). Begge akser er logaritmiske.

8.1 Elementvise primitiver



Ved omkring 2^{15} elementer bliver vores GPGPU-implementation af map hurtigere end en simpel CPU-version, og er mod slutningen ca. 15 gange hurtigere. De andre elementvise primitiver ligger tæt op af map i tidsforbrug.

8.2 Scan



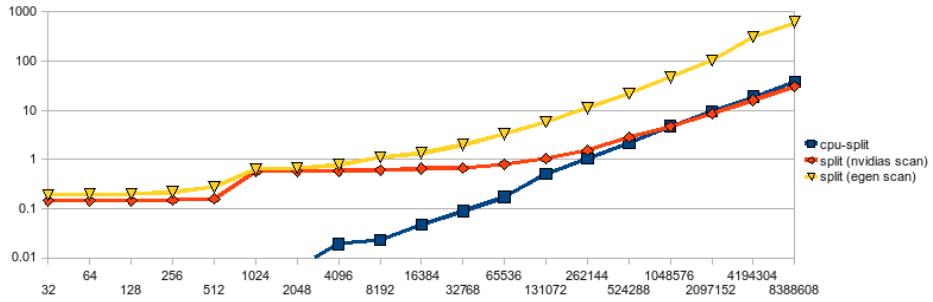
Ved omkring 2^{15} elementer bliver NVIDIAs GPGPU-baserede scan-implementation hurtigere end en simpel implementation på CPU'en.

Vores implementation af scan bliver derimod ikke hurtigere for de størrelser vi har målt, og hvis kurven ekstrapoleres nærmer de sig heller ikke CPU-versionen. Hen mod slutningen af kurven er den ca. 60 gange langsommere end NVIDIAs implementation. Vores segmented scan er til sidst ca. fire gange langsommere end vores scan.

For store vektorer er NVIDIAs scan ca. tre gange hurtigere end CPU-versionen.

Springet på kurven for mellem 512 elementer og 1024 elementer er sandsynligvis et resultat af at gå fra en enkelt til to blokke, idet der som beskrevet i afsnit 2 kun kan være 512 tråde i en blok.

8.3 Split



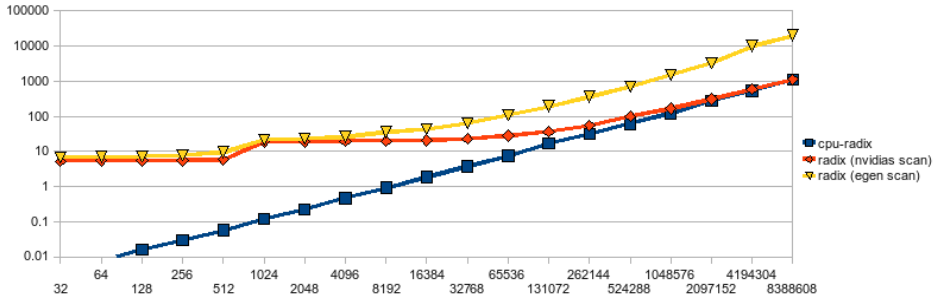
Springet fra scan skinner igennem på denne kurve, da vi bruger scan i implementationen.

Det relativt mindre forhold mellem implementationen med vores scan og NVIDIAs scan kommer af at vi udover scan også bruger de relativt hurtige elementvise primitiver.

GPGPU-versionen af split bliver aldrig markant hurtigere end den simple CPU-version, heller ikke selvom vi anvender NVIDIAs scan.

I starten af kurven er omkostningen ved NVIDIAs scan høj, men den bliver relativt mindre for større vektorer. NVIDIAs scan kun er tre gange hurtigere end en simpel summering på CPU'en (cpu-scan), og da hver iteration af cpu-radix ikke er meget mere kompleks end en summering giver det mening at den er stort set lige så hurtig.

8.4 Radix sort



Da størstedelen af arbejdet i vores radix sort foregår i split-funktionen, ligner kurven for de to funktioner ikke overraskende hinanden.

9 Analyse

9.1 Forbedringer

I løbet af udviklingen løb vi ofte ind i segmentation faults som følge af at følge en pointer til hukommelse der var allokeret på grafikkortet, eller fordi vi gav en pointer til hukommelse på værtsystemet videre til en funktion der forventede at det lå på grafikkortet. For at undgå dette kunne man ændre `PARACUDA_*_allocate_device` så den returnerede en `device_vector_t` der ikke kunne anvendes som en pointer af brugeren. Tilsvarende skulle `PARACUDA_*_copy_device_host` tage en `device_vector_t` og en normal pointer, istedet for at tage normale pointere som begge argumenter. Hvis man tilpassede hele biblioteket på den måde, kunne man helt undgå de segmentation faults der er relaterede til hvor hukommelsen er allokeret.

Radix- og split-algoritmerne bruger midlertidige vektorer, som de allokerer ved start og frigør i slutningen af funktionen. Dette gav ikke nogen mærkbar omkostning i vores tilfælde fordi kroppen tog relativt lang tid at køre. Hvis det ikke var tilfældet, kunne en måde at undgå det på være at lave en klasse der allokerede de midlertidige vektorer ved oprettelse og frigjorde dem ved nedlæggelse (construction og destruction), og så have selve funktionen som en metode. På den måde kunne man beholde klassen i hukommelsen og undgå allokeringer i hvert kald til funktionen.

NVIDIAs implementation af scan

NVIDIAs scan for små vektorer er implementeret så den starter med at hente elementer fra den globale hukommelse ind i shared memory. Alle beregninger foretages så på shared memory hvorefter resultatet skrives tilbage i den globale hukommelse.

De deler vektoren op i dele der passer i en blok og bruger scan for små vektorer på hver del i hver sin blok.

Derefter laver de scan på en vektor der indeholder summen af hver blok, og til sidst lægger de hvert af disse elementer til den tilsvarende bloks delvektor.

Dermed opnår de maksimal samtidighed, da hvert element har sin egen tråd, og trådene er delt ud over mange blokke så de kan køre på flere cores.

En mere udførlig beskrivelse kan findes i [Harris].

9.2 Arbejdsbyrde

Den parallele scan-algoritme er i [Shubhabrata] ca. 10 linjer lang. NVIDIAs implementation (i scanLargeArray-eksemplet i deres CUDA SDK) er ca. 500 linjer lang. I dette eksempel er 98% af koden altså dedikeret til C- eller CUDA-specifikke optimeringer eller problemer, og den kan stadig ikke håndtere vektorstørrelser på over 2^{24}.

Da det altid vil være en investering at bruge en ny teknologi, og da teknologien typisk ikke er lige så tilgængelig som konventionelle

CPU'er, er det nødvendigt at en implementation på CUDA kører betydeligt hurtigere end en normal CPU-implementation før det kan betale sig at benytte teknologien. I vores benchmarks er NVIDIAs CUDA-implementation af scan for store vektorer ca. tre gange hurtigere end den sekventielle version, hvilket ikke nødvendigvis er nok til at gøre det besværdet værd.

Set fra en anden vinkel, hvis dette er en indikator for hvor meget arbejde der skal lægges i at implementere noget på CUDA, bør man grundigt overveje hvorvidt den ekstra arbejdsbyrde er den potentielle hastighedsforøgelse værd.

NVIDIAs CUDA-hjemmeside har en oversigt over implementerede algoritmer på CUDA og hvor mange gange hurtigere de kører end deres CPU-implementation. Selvom de ikke nødvendigvis er perfekte sammenligninger, kan de give en ide om hvilke typer af algoritmer der er velegnet til CUDA og hvilke der ikke er.

9.3 Vector-scan-modellen på CUDA

NVIDIAs egen implementation af scan er begrænset til at arbejde på en vektor af floating point tal, men der er ingen teknisk grund til at den ikke kan generaliseres til vilkårlige vektorer. Dermed kan man erstatte den underliggende implementation af scan i vores bibliotek med deres langt hurtigere version.

For hver tredje Scan man laver kan man lave en simpel iteration svarende til en summering hen over sin vektor på CPU-en, hvilket betyder at hvis man har en sekventiel algoritme med relativt simple iterationer (som for eksempel radixsort), kan den parallele version højest benytte tre scans per iteration før den bliver langsommere (under antagelse af at de bruger samme antal iterationer).

Det er svært at forestille sig tilfælde hvor en scan eller Segmented Scan der kun er tre gange hurtigere på grafikkortet kan føre til signifikant mindre tidsforbrug.

Man skal naturligvis tage det hardware vi har målt på og vores metode i betragtning - det er ikke åbentlyst om det er fair at sammenligne de to typer hardware. Derudover giver CUDA mulighed for at afvikle algoritmerne på GPGPU'en mens CPU'en arbejder på noget andet, hvilket gør at det kan være en fordel selvom det ikke er hurtigere.

10 Konklusion

10.1 Hastighed

Som det klart fremgår i 8, er vores program meget langsommere end både NVIDIAs implementation, og en tilsvarende algoritme på CPU'en. Ud fra dette bliver vi nød til at konkludere at vores implementation ikke på nuværende tidspunkt er praktisk anvendeligt, og at yderligere arbejde ville være nødvendigt for at få den til at virke tilfredstillende. Vores målinger viser dog også at Radixsort, som implementeret efter [Blelloch] ikke er interessant, da scan er for langsom (hvis man arbejder på hardware med et lignende forhold mellem sig som det vi har målt på).

10.1.1 Fremtidig fokus på optimeringer

Ud fra de erfaringer vi har gjort os med NVIDIAs platform, vil vores råd være at fremtidigt arbejde vil fokusere på optimering af hukommelsen, hvilket i praksis betyder at den delte hukommelse skal udnyttes bedre og at bank conflicts skal undgås. Dette efter al sandsynlighed også betyde at der i fremtidige projektor også skal lægges vægt på at den udenomliggende kode der skal dele kaldene til kernelen op, da grafikkortet har begrænset delt hukommelse. Det er desuden klart at der i fremtidige biblioteker skal være fokus på både at håndtere data der er blevet kopieret data til GPU'en såvel som at dette ikke skal håndteres af kaldene til primitiverne.

Desuden skal man lægge vægt på at opnå maksimal samtidighed ved at fylde så mange blokke op som muligt.

10.2 Mulighed for arbitrære operatorer

Dog har vores projekt vist at det er muligt at skabe et bibliotek af scan primitiver som kan arbejde på arbitrært data med en abitrær operator - og at når koden først er blevet skrevet, så er det let at define nye funktioner.

A Data for tidsforbrug

Første række er antal elementer og første søjle er navne på de kørte funktioner. Tidsforbruget er opgivet i millisekunder.

	32	64	128	256	512	1024	2048	4096	8192	16384
nvidia-scan	0.03	0.03	0.03	0.03	0.03	0.06	0.07	0.07	0.08	0.11
cpu-radix	0	0.01	0.02	0.03	0.06	0.12	0.23	0.47	0.91	1.85
cpu-split	0	0	0	0	0	0	0.01	0.01	0.02	0.05
cpu-map	0	0	0	0	0	0	0	0	0.01	0.01
cpu-scan	0	0	0	0	0	0	0	0.01	0.02	0.04
scan	0.07	0.04	0.05	0.06	0.09	0.1	0.12	0.17	0.33	0.45
segmented-scan	0.09	0.09	0.1	0.12	0.15	0.62	0.65	0.75	1.09	1.4
permute	0.01	0.01	0.01	0.01	0.01	0.01	0.02	0.02	0.03	0.05
map	0.01	0.01	0.01	0.01	0.01	0.01	0.02	0.02	0.03	0.05
copy	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
split	0.19	0.19	0.2	0.22	0.27	0.64	0.68	0.77	1.1	1.34
radix	6.66	6.84	7.14	7.68	9.52	21.38	22.55	25.56	35.64	42.87
nvidia-split	0.14	0.14	0.14	0.15	0.16	0.57	0.57	0.58	0.6	0.65
nvidia-radix	5.25	5.31	5.35	5.49	5.76	18.97	19.04	19.38	19.67	20.75

	32768	65536	131072	262144	524288	1048576	2097152	4194304	8388608
nvidia-scan	0.07	0.1	0.15	0.24	0.56	0.93	1.65	3.03	5.75
cpu-radix	3.72	7.39	16.58	31.56	61.42	122.36	269.17	537.05	1081.44
cpu-split	0.09	0.17	0.5	1	2.08	4.77	10.02	19.12	40.17
cpu-map	0.03	0.05	0.1	0.2	0.47	0.81	2.75	6.03	12.36
cpu-scan	0.07	0.15	0.3	0.59	1.19	2.37	4.77	10.2	20.4
scan	0.72	1.34	2.56	5	10.9	21.66	48.79	149.02	297.98
segmented-scan	2.04	3.47	6.33	12.19	29.35	56.79	116.53	369.59	1127.52
permute	0.02	0.02	0.03	0.05	0.09	0.15	0.28	0.53	1.02
map	0.02	0.02	0.03	0.04	0.07	0.12	0.21	0.4	0.77
copy	0.02	0.02	0.02	0.03	0.05	0.09	0.16	0.32	0.56
split	1.95	3.26	5.85	11.01	21.74	46.12	102.95	308.3	615.38
radix	64.41	106.5	190.81	358.25	695.45	1496.3	3329.77	9938.08	19847.44
nvidia-split	0.66	0.79	1.03	1.51	2.78	4.67	8.44	16.09	30.8
nvidia-radix	22.88	27.69	36.16	54.08	98.1	166.83	305.31	583.72	1127.22

B Kildekode

B.1 main.cpp

```
extern void runTest(int argc, char** argv);
extern void runQuicksort(int argc, char** argv);
extern void runTestRadix(int argc, char** argv);

#include <cstdio>
int main(int argc, char** argv)
{
  //runQuicksort(argc, argv);
  runTest(argc, argv);
  //runTestRadix(argc, argv);
  return 0;
}
```

B.2 test.cu

```
#include <cstdio>
#include <ctime>

#define USE_NVIDIA_SCAN
```

```
#define REPEATS 10

#include "arrayprint.h"
#include "paracuda.h"
#include "kernel.cu"
#include "split.h"
#include "copy.h"
#include "radix.h"
#include "cpu_radix.h"
#include "cpu_split.h"
#include "cpu_map.h"
#include "cpu_scan.h"
#include "nvidia_scan.h"

void testCpuRadix(int* data, int* gold, int length,
    unsigned int timer)
{
    srand(time(0));
    for(int i = 0; i < length; i++) {
        data[i] = rand() % 1000;
    }

    int* temp = (int*) malloc(length * sizeof(int));
    cutilCheckError(cutStartTimer(timer));
    cpu_radix(data, temp, length);
    cutilCheckError(cutStopTimer(timer));
    free(temp);
    memcpy(gold, data, length * sizeof(int)); // only test
        speed
}

void testCpuSplit(int* data, int* gold, int length,
    unsigned int timer)
{
    srand(time(0));
    for(int i = 0; i < length; i++) {
        data[i] = rand() % 1000;
    }

    int* temp = (int*) malloc(length * sizeof(int));
    cutilCheckError(cutStartTimer(timer));
    cpu_split(data, temp, temp, length);
    cutilCheckError(cutStopTimer(timer));
    free(temp);
    memcpy(gold, data, length * sizeof(int)); // only test
        speed
}

void testCpuMap(int* data, int* gold, int length, unsigned
    int timer)
{
    srand(time(0));
    for(int i = 0; i < length; i++) {
        data[i] = rand() % 1000;
    }

    cutilCheckError(cutStartTimer(timer));
    cpu_map(data, length);
    cutilCheckError(cutStopTimer(timer));
    memcpy(gold, data, length * sizeof(int)); // only test
        speed
}
```

```
void testCpuScan(int* data, int* gold, int length, unsigned
     int timer)
{
    srand(time(0));
    for(int i = 0; i < length; i++) {
        data[i] = rand() % 1000;
    }

    cutilCheckError(cutStartTimer(timer));
    cpu_scan(data, length);
    cutilCheckError(cutStopTimer(timer));
    memcpy(gold, data, length * sizeof(int)); // only test
        speed
}

void testNvidiaScan(int* data, int* gold, int length,
    unsigned int timer)
{
    srand(time(0));
    for(int i = 0; i < length; i++) {
        data[i] = rand() % 1000;
    }

    int* device_data = PARACUDA_int_allocate_device(length)
        ;
    PARACUDA_int_copy_host_device(device_data, data, length
        );
    preallocBlockSums(length);
    cutilCheckError(cutStartTimer(timer));
    prescanArray(device_data, device_data, length);
    cutilCheckError(cutStopTimer(timer));
    deallocBlockSums();
    PARACUDA_int_copy_device_host(data, device_data, length
        );
    PARACUDA_int_free_device(device_data);

    memcpy(gold, data, length * sizeof(int)); // only test
        speed
}

void testRadix(int* data, int* gold, int length, unsigned
    int timer)
{
    srand(time(0));
    for(int i = 0; i < length; i++) {
        data[i] = rand() % 1000;
    }

    for(int i = 0; i < length; i++) {
        gold[i] = data[i];
    }

    int* temp = (int*) malloc(length * sizeof(int));
    cpu_radix(gold, temp, length);
    free(temp);

    int* device_data = PARACUDA_int_allocate_device(length)
        ;
    PARACUDA_int_copy_host_device(device_data, data, length
        );
    cutilCheckError(cutStartTimer(timer));
```

```
    radix(device_data, length, PARACUDA_MAX_THREADS);
    cutilCheckError(cutStopTimer(timer));
    PARACUDA_int_copy_device_host(data, device_data, length
        );
    PARACUDA_int_free_device(device_data);
}

void testSplit(int* data, int* gold, int length, unsigned
    int timer)
{
    int* flag = PARACUDA_int_allocate_host(length);
    for(int i = 0; i < length; i++) {
        data[i] = i;
        flag[i] = i % 2 == 0;
    }
    int numZero = 0;
    int indexForZero = 0;
    for(int i = 0; i < length; i++) {
        if(flag[i] == 0)
        numZero++;
    }
    for(int i = 0; i < length; i++) {
        if(flag[i] == 1) {
            gold[numZero] = data[i];
            numZero++;
        } else {
            gold[indexForZero] = data[i];
            indexForZero++;
        }
    }
    int* d_data = PARACUDA_int_allocate_device(length);
    int* d_flags = PARACUDA_int_allocate_device(length);
    PARACUDA_int_copy_host_device(d_data, data, length);
    PARACUDA_int_copy_host_device(d_flags, flag, length);
    cutilCheckError(cutStartTimer(timer));
    split(d_data, d_flags, length);
    cutilCheckError(cutStopTimer(timer));
    PARACUDA_int_copy_device_host(data, d_data, length);
    PARACUDA_int_free_device(d_data);
    PARACUDA_int_free_device(d_flags);
    free(flag);
}

void testPermute(int* data, int* gold, int length, unsigned
     int timer)
{
    int* positions = (int*) malloc(length * sizeof(int));
    for(int i = 0; i < length; i++) {
      data[i] = i;
      positions[i] = length - i - 1;
    }
    int_permute(gold, gold, positions, length,
        PARACUDA_MAX_THREADS); // warmup
    for(int i = 0; i < length; i++) {
      gold[positions[i]] = data[i];
    }
    int* d_out = PARACUDA_int_allocate_device(length);
    int* d_data = PARACUDA_int_allocate_device(length);
    int* d_positions = PARACUDA_int_allocate_device(length)
        ;
    PARACUDA_int_copy_host_device(d_data, data, length);
```

```
    PARACUDA_int_copy_host_device(d_positions, positions,
        length);
    cutilCheckError(cutStartTimer(timer));
    PARACUDA_int_permute_run(d_out, d_data, d_positions,
        length, PARACUDA_MAX_THREADS);
    cutilCheckError(cutStopTimer(timer));
    PARACUDA_int_copy_device_host(data, d_out, length);
    PARACUDA_int_free_device(d_out);
    PARACUDA_int_free_device(d_data);
    PARACUDA_int_free_device(d_positions);
    free(positions);
}

void testScan(int* data, int* gold, int length, unsigned
    int timer)
{
    for(int i = 0; i < length; i++) {
        data[i] = i;
    }
    int sum = 0;
    for(int i = 0; i < length; i++) {
        gold[i] = sum;
        sum += data[i];
    }

    int* d_data = PARACUDA_int_allocate_device(length);
    PARACUDA_int_copy_host_device(d_data, data, length);

    cutilCheckError(cutStartTimer(timer));
    PARACUDA_plus_scan_run(0, d_data, length,
        PARACUDA_MAX_THREADS);
    cutilCheckError(cutStopTimer(timer));

    PARACUDA_int_copy_device_host(data, d_data, length);
    PARACUDA_int_free_device(d_data);
}

void testSegmentedScan(int* data, int* gold, int length,
    unsigned int timer)
{
    int* flags = (int*) calloc(length, sizeof(int));
    for(int i = 0; i < length; i++) {
        data[i] = i;
    }
    flags[length / 2] = 1;
    int sum = 0;
    for(int i = 0; i < length; i++) {
        if (flags[i]) sum = 0;
        gold[i] = sum;
        sum += data[i];
    }
    int* d_flags = PARACUDA_int_allocate_device(length);
    int* d_flags_copy = PARACUDA_int_allocate_device(length
        );
    int* d_data = PARACUDA_int_allocate_device(length);
    PARACUDA_int_copy_host_device(d_data, data, length);
    PARACUDA_int_copy_host_device(d_flags, flags, length);
    PARACUDA_int_copy_host_device(d_flags_copy, flags,
        length);
    cutilCheckError(cutStartTimer(timer));
    PARACUDA_segmented_plus_scan_run(0, d_data,
        d_flags_copy, d_flags, 0, length,
```

```
        PARACUDA_MAX_THREADS);
    cutilCheckError(cutStopTimer(timer));
    PARACUDA_int_copy_device_host(data, d_data, length);
    PARACUDA_int_free_device(d_flags);
    PARACUDA_int_free_device(d_flags_copy);
    PARACUDA_int_free_device(d_data);
    free(flags);
}

void testMap(int* data, int* gold, int length, unsigned int
    timer)
{
    for(int i = 0; i < length; i++) {
        data[i] = i % 2 == 0;
    }
    for(int i = 0; i < length; i++) {
        gold[i] = !data[i];
    }

    int* d_data = PARACUDA_int_allocate_device(length);
    PARACUDA_int_copy_host_device(d_data, data, length);

    cutilCheckError(cutStartTimer(timer));
    PARACUDA_negate_map_run(d_data, d_data, length,
        PARACUDA_MAX_THREADS);
    cutilCheckError(cutStopTimer(timer));

    PARACUDA_int_copy_device_host(data, d_data, length);
    PARACUDA_int_free_device(d_data);
}

void testCopy(int* data, int* gold, int length, unsigned
    int timer)
{
    int value;
    for(int i = 0; i < length; i++) {
        gold[i] = value;
    }

    int* d_data = PARACUDA_int_allocate_device(length);

    cutilCheckError(cutStartTimer(timer));
    PARACUDA_int_copy_run(d_data, value, length,
        PARACUDA_MAX_THREADS);
    cutilCheckError(cutStopTimer(timer));

    PARACUDA_int_copy_device_host(data, d_data, length);
    PARACUDA_int_free_device(d_data);
}

#define TEST(T, f, name, lengths, count, times) {\
    for(int a = 0; a < count; a++) {\
        unsigned int timer = 0;\
        cutilCheckError(cutCreateTimer(&timer));\
        PARACUDA##T##_struct* data = PARACUDA##T##
            _allocate_host(lengths[a]);\
        PARACUDA##T##_struct* gold = PARACUDA##T##
            _allocate_host(lengths[a]);\
        printf("Running %s for %d elements:\n", name,
            lengths[a]);\
        for(int h = 0; h < REPEATS; h++) f(data, gold,
            lengths[a], timer);\
```

```
        bool same = true;\
        for(int b = 0; b < lengths[a]; b++) {\
            int wrong = !PARACUDA##T##_equal(data, b, gold
                , b);\
            if (b < 10 || (wrong && same)) {\
                /*printf("%d:\t%d %d\t%d %d\n", b,*/
                                    \
                    /*data->x[b], data->y[b], gold->x[b],
                        gold->y[b]);*/          \
                printf("%d:\t%d\t%d\n", b,
                                \
                    data[b], gold[b]);  \
                if(wrong) printf("^^^^^^^^^^^^^^^^^^^\n")
                    ;\
            }\
            same &= !wrong;\
        }\
        if(same) {\
            printf("They are the SAME! Processing time: %f
                ms, average: %f ms.\n\n", \
                cutGetTimerValue(timer),
                    cutGetAverageTimerValue(timer));\
        } else {\
            printf("They are DIFFERENT! Processing time: %f
                 ms, average: %f ms.\n\n", \
                cutGetTimerValue(timer),
                    cutGetAverageTimerValue(timer));\
        }\
        if(!same) { printf("Errornous output, aborting test
            .\n"); exit(-1); }\
        times[a] = cutGetAverageTimerValue(timer);\
        cutilCheckError(cutDeleteTimer(timer));\
    }\
}\

void runTest(int argc, char** argv) {
    if(argc == 1) {
        printf("USAGE: scripts/makerun.sh paracuda <
            testname1> <testname2...>\n");
        printf("EXAMPLE: scripts/makerun.sh paracuda split\
            n");
        return;
    }
    //PARACUDA_INITIALIZE_CUDA(argc, argv);
    cutilCheckMsg("Test");
    int all = strcmp(argv[1], "all") == 0;
    int lengths[] = {32, 64, 128, 256, 512, 1024, 2 * 1024,
         4 * 1024, 8 * 1024, 16 * 1024,
        32 * 1024, 64 * 1024, 128 * 1024, 256 * 1024, 512 *
             1024, 1024 * 1024,
        2 * 1024 * 1024, 4 * 1024 * 1024, 8 * 1024 * 1024};
    int count = sizeof(lengths) / sizeof(*lengths);
    float times[argc - 1][count];
    for(int i = 1; i < argc; i++) {
        char* name = argv[i];
        float* local = times[i - 1];
        if(strcmp(name, "cpu-radix") == 0) TEST(int,
            testCpuRadix, name, lengths, count, local)
        else if(strcmp(name, "cpu-split") == 0) TEST(int,
            testCpuSplit, name, lengths, count, local)
        else if(strcmp(name, "cpu-map") == 0) TEST(int,
            testCpuMap, name, lengths, count, local)
```

```
        else if(strcmp(name, "cpu-scan") == 0) TEST(int,
            testCpuScan, name, lengths, count, local)
        else if(strcmp(name, "nvidia-scan") == 0) TEST(int,
             testNvidiaScan, name, lengths, count, local)
        else if(strcmp(name, "scan") == 0) TEST(int,
            testScan, name, lengths, count, local)
        else if(strcmp(name, "segmented-scan") == 0) TEST(
            int, testSegmentedScan, name, lengths, count,
            local)
        else if(strcmp(name, "permute") == 0) TEST(int,
            testPermute, name, lengths, count, local)
        else if(strcmp(name, "map") == 0) TEST(int, testMap
            , name, lengths, count, local)
        else if(strcmp(name, "copy") == 0) TEST(int,
            testCopy, name, lengths, count, local)
        else if(strcmp(name, "split") == 0) TEST(int,
            testSplit, name, lengths, count, local)
        else if(strcmp(name, "radix") == 0) TEST(int,
            testRadix, name, lengths, count, local)
        else
        {
            printf("Unknown test: %s\n", name);
            exit(-1);
        }
    }
#ifdef USE_NVIDIA_SCAN
    printf("# PLOT: (first row: elements, remaining rows:
        times in milliseconds) (NVIDIAs scan)\n");
#else
    printf("# PLOT: (first row: elements, remaining rows:
        times in milliseconds) (own scan)\n");
#endif
    printf("name");
    for(int l = 0; l < count; l++) {
        printf(",%d", lengths[l]);
    }
    printf("\n");
    for(int j = 1; j < argc; j++) {
        printf("%s", argv[j]);
        for(int l = 0; l < count; l++) {
            printf(",%f", times[j - 1][l]);
        }
        printf("\n");
    }
    //PARACUDA_EXIT_KEYPRESS(argc, argv);
}
```

B.3 kernel.cu

```
#include "paracuda.h"
#include "paracuda_segmentedScan.h"
#include "limits.h"

PARACUDA_SINGLE(int)

PARACUDA_COPY(int_copy, int);

/* Define the mapping (x, y) -> (x + y, x * y) */

PARACUDA_STRUCT_2(pair_t, pair_vector_t,
    int, x,
    int, y
```

```
)

PARACUDA_OPERATOR(plus_times, void, (pair_t* r, pair_t* v),
     ({
    r->x = v->x + v->y;
    r->y = v->x * v->y;
}))

PARACUDA_MAP(plus_times_map, plus_times, pair_t, pair_t)

/* Define the scan that can detect ordering */

PARACUDA_OPERATOR(ordered_zero, void, (pair_t* r), ({
    r->x = 0;
    r->y = 2;
}))

PARACUDA_OPERATOR(ordered, void, (pair_t* r, pair_t* a,
    pair_t* b), ({
    if(a->y == 2) {
        r->x = b->x;
        r->y = 1;
        return;
    }
    r->x = b->x;
    r->y = a->x <= b->x && a->y && b->y;
}))

PARACUDA_SCAN(ordered_scan, ordered, ordered_zero, pair_t)

/* Define the scan [a, b, ..., y, z] -> [0, a, a + b, ...,
    a + b + ... + y] */

PARACUDA_OPERATOR(plus, void, (int* r, int* a, int* b), ({
    *r = *a + *b;
}))

PARACUDA_OPERATOR(zero, void, (int* r), ({
    *r = 0;
}))

PARACUDA_OPERATOR(one, void, (int* r), ({
   *r = 1;
}));

PARACUDA_SCAN(plus_scan, plus, zero, int)

/* Define segmented scan via scan (experiment, non-working
    - ys are being set to non-zero by an evil magical force
    ) */

PARACUDA_OPERATOR(int_flag_zero, void, (pair_t* r), ({
    r->x = 0;
}))

PARACUDA_OPERATOR(int_flag_zero_seg, void, (int* r), ({
  *r = 0;
}))

PARACUDA_OPERATOR(segmented_plus, void, (int* r, int* a,
    int* b), ({
    *r = *a + *b;
```

```
}))

PARACUDA_OPERATOR(add, void, (int* r, pair_t* i), ({
    *r = i->x + i->y;
}))

PARACUDA_OPERATOR(minus, void, (int* r, pair_t* i), ({
   *r = i->x - i->y;
}))

PARACUDA_SEGMENTEDSCAN(segmented_plus_scan, segmented_plus,
     zero, int)

/* Negation [0, 1, 1, 0, 0, 0, 1, 0] -> [1, 0, 0, 1, 1, 1,
    0, 1] */

PARACUDA_OPERATOR(negate, void, (int* r, int* a), ({
    *r = (*a) ? 0 : 1;
}))

PARACUDA_MAP(negate_map, negate, int, int)

PARACUDA_STRUCT_3(split_t, split_vector_t,
    int, flags,
    int, left,
    int, right
)

PARACUDA_OPERATOR(split, void, (int* r, split_t* a), ({
    *r = (a->flags) ? a->right : a->left;
}))

PARACUDA_MAP(split_map, split, int, split_t)

PARACUDA_PERMUTE(int_permute, int)

// Radix

PARACUDA_STRUCT_2(bitwise_t, bitwise_vector_t,
    int, integer,
    int, number
)

PARACUDA_OPERATOR(int_bitwise, void, (int* o, bitwise_t* r)
    , ({
*o = (r->integer & r->number) == r->number;
}))

PARACUDA_MAP(int_bitwise_map, int_bitwise, int, bitwise_t)


// Transfer

PARACUDA_OPERATOR(int_transfer, void, (int* o, int* i), ({
*o = *i;
}))

PARACUDA_MAP(int_transfer_map, int_transfer, int, int)


PARACUDA_OPERATOR(int_copy_pivot, void, (int* r, pair_t* i)
    , ({
```

```
    *r = (i->x) ? i->y : 0;
}))

PARACUDA_MAP(int_copy_pivot_map, int_copy_pivot, int,
    pair_t)

PARACUDA_OPERATOR(int_add_pivot, void, (int* r, split_t* i)
    , ({
    *r = (i->flags) ? i->left + i->right : i->left;
}))

PARACUDA_MAP(int_add_pivot_map, int_add_pivot, int, split_t
    )

PARACUDA_OPERATOR(int_insert_pivot, void, (int* r, split_t*
     i), ({
    *r = (i->flags) ? i->left : i->right;
}))

PARACUDA_MAP(int_insert_pivot_map, int_insert_pivot, int,
    split_t);

PARACUDA_OPERATOR(and_distribute_op, void, (int* r, int* a,
     int* b), ({
    *r = (*a & *b);
}))

PARACUDA_OPERATOR(or_distribute_op, void, (int* r, int* a,
    int* b), ({
    *r = (*a | *b);
}))

PARACUDA_OPERATOR(max, void, (int* r, int* a, int* b), ({
  *r = (*a < *b) ? *b : *a;
}))


PARACUDA_SCAN(and_distribute_scan, and_distribute_op, one,
    int)

PARACUDA_SCAN(or_distribute_scan, or_distribute_op, zero,
    int)

PARACUDA_SEGMENTEDSCAN(and_distribute_segmented_scan,
    and_distribute_op, one, int)

PARACUDA_SEGMENTEDSCAN(or_distribute_segmented_scan,
    or_distribute_op, one, int)

PARACUDA_SEGMENTEDSCAN(max_segmented_scan, max, zero, int)

PARACUDA_OPERATOR(compare, void, (int* r, pair_t* i), ({
    *r = i->y <= i->x;
}))

PARACUDA_MAP(compare_map, compare, int, pair_t)

PARACUDA_MAP(map_add, add, int, pair_t)

PARACUDA_OPERATOR(map_or, void, (int* r, pair_t* i), ({
   *r = i->x | i->y;
}))
```

```
PARACUDA_MAP(or_map, map_or, int, pair_t)

PARACUDA_MAP(map_minus, minus, int, pair_t)


PARACUDA_OPERATOR(find_new_array, void, (int* r, split_t* i
    ), ({
    *r = ((i->flags < i->left && !(i->flags < i->right)))
        || ((i->flags > i->left && !(i->flags > i->right)))
         ? 1 : 0;
}))

PARACUDA_MAP(find_new_array_map, find_new_array, int,
    split_t)
```

B.4 copy.h

```
#include "arrayprint.h"

int* copy(int* array, int num, int length)

{
  // Clear the array
  memset(array, 0, length * sizeof(int));
  // Scan
  array[0] = num;
  plus_scan(0, array, array, length, PARACUDA_MAX_THREADS);
  array[0] = num;
  return array;
}
```

B.5 split.h

```
#include "arrayprint.h"
#include "nvidia_scan.h"

#ifdef USE_NVIDIA_SCAN
#define SCAN(ARRAY) nvidia_scan(ARRAY, ARRAY, length);
#else
#define SCAN(ARRAY) PARACUDA_plus_scan_run(0, ARRAY, length
    , PARACUDA_MAX_THREADS);
#endif

void split(int* array, int* flags, int length)
{
  int* posDown      = PARACUDA_int_allocate_device(length);
  int* posUp        = PARACUDA_int_allocate_device(length);
  int* positions    = PARACUDA_int_allocate_device(length);
  pair_vector_t* pair =
      PARACUDA_pair_t_shallow_allocate_device();
  split_vector_t* input =
      PARACUDA_split_t_shallow_allocate_device();

  int before;
  int computed_sum;
  PARACUDA_negate_map_run(posDown, flags, length,
      PARACUDA_MAX_THREADS);
  PARACUDA_int_peek(&before, posDown, length - 1);
  SCAN(posDown);
  PARACUDA_int_peek(&computed_sum, posDown, length - 1);
```

```
  computed_sum += before;
  PARACUDA_int_copy_run(positions, computed_sum, length,
      PARACUDA_MAX_THREADS);

  PARACUDA_int_copy_device_device(posUp, flags, length);
  SCAN(posUp);

  pair_vector_t host_pair;
  host_pair.x = positions;
  host_pair.y = posUp;
  PARACUDA_pair_t_shallow_copy_host_device(pair, &host_pair
      );
  PARACUDA_map_add_run(posUp, pair, length,
      PARACUDA_MAX_THREADS);

  split_vector_t host_input;
  host_input.flags = flags;
  host_input.left = posDown;
  host_input.right = posUp;
  PARACUDA_split_t_shallow_copy_host_device(input, &
      host_input);
  PARACUDA_split_map_run(positions, input, length,
      PARACUDA_MAX_THREADS);

  PARACUDA_int_permute_run(posDown, array, positions,
      length, PARACUDA_MAX_THREADS);
  PARACUDA_int_copy_device_device(array, posDown, length);

  PARACUDA_pair_t_shallow_free_device(pair);
  PARACUDA_split_t_shallow_free_device(input);
  PARACUDA_int_free_device(posDown);
  PARACUDA_int_free_device(posUp);
  PARACUDA_int_free_device(positions);
}
```

B.6 radix.h

```
#import "arrayprint.h"

void radix(int* array, int length, int max_threads)
{
  int* t_numbers = PARACUDA_int_allocate_device(length);
  int* t_flags = PARACUDA_int_allocate_device(length);
  bitwise_vector_t* in =
      PARACUDA_bitwise_t_shallow_allocate_device();
  for(int i = 0; i < 32; ++i) {
    int num = (1 << i);
    PARACUDA_int_copy_run(t_numbers, num, length,
        max_threads);
    bitwise_vector_t input;
    input.number = t_numbers;
    input.integer = array;
    PARACUDA_bitwise_t_shallow_copy_host_device(in, &input)
        ;
    PARACUDA_int_bitwise_map_run(t_flags, in, length,
        max_threads);
    split(array, t_flags, length);
  }
  PARACUDA_bitwise_t_shallow_free_device(in);
  PARACUDA_int_free_device(t_numbers);
  PARACUDA_int_free_device(t_flags);
}
```

B.7 arrayprint.h

```
#ifndef ARRAYPRINT_H
#define ARRAYPRINT_H

#include "stdio.h"

#define PRINT_HOST(NAME) print_array(#NAME, NAME, length)

#define PRINT_DEVICE(NAME, TYPE) ({\
  PARACUDA_##TYPE##_struct* foo_992_avaq1 = PARACUDA_##TYPE
      ##_allocate_host(length);\
  PARACUDA_##TYPE##_copy_device_host(foo_992_avaq1, NAME,
      length);\
  print_array(#NAME, foo_992_avaq1, length);\
  PARACUDA_##TYPE##_free_host(foo_992_avaq1);\
})\

void print_array(char* name, int* array, int length)
{
  printf("%s: ", name);
  for(int i = 0; i < length; i++) printf("%d ", array[i]);
  printf("\n");
}

#endif
```

B.8 paracuda.h

```
#ifndef PARACUDA_H
#define PARACUDA_H

#include <cutil_inline.h>
#include "paracuda_struct.h"
#include "paracuda_map.h"
#include "paracuda_scan.h"
#include "paracuda_copy.h"
#include "paracuda_permute.h"
//#include "paracuda_segmentedScan.h"


/* Easy initialization of CUDA. It is optional to call this
    . */
/* You are not required to call this. */
void PARACUDA_INITIALIZE_CUDA(int argc, char** argv)
{
    if(cutCheckCmdLineFlag(argc, (const char**) argv, "
        device"))
        cutilDeviceInit(argc, argv);
    else
        cudaSetDevice(cutGetMaxGflopsDeviceId());
}

/* Exits the program, waiting for a keypress. */
/* You are not required to call this. */
void PARACUDA_EXIT_KEYPRESS(int argc, char** argv)
{
    cudaThreadExit();
    cutilExit(argc, argv);
}
```

```
/* Calculates the smallest power of two which is greater
    than or equal to number. */
int PARACUDA_NEXT_POWER_OF_TWO(
    size_t number)
{
    size_t count = 0;
    while(1 << count < number) count++;
    return count;
}

/* Asserts that a number is a power of two (prints an error
     and exits otherwise). */
void PARACUDA_ASSERT_POWER_OF_TWO(
    size_t number)
{
    if(1 << PARACUDA_NEXT_POWER_OF_TWO(number) != number) {
        fprintf(stderr, "Length must be a power of two, but
             was %d.\n", number);
        exit(1);
    }
}

/* Base 2 integer logarithm. */
size_t PARACUDA_LOG2(size_t n) {
    size_t l = 0;
    while(n > 1) {
        n >>= 1;
        l += 1;
    }
    return l;
}

#define PARACUDA_MAX_THREADS 512

/* Defines an operator. The parenthesis around the
    arguments and the body are required: */
/* PARACUDA_OPERATOR(name, void, (int x, int y), ({ ... }))
     */
/* It may create additional definitions that have NAME_ as
    a prefix. */
#define PARACUDA_OPERATOR(NAME, RETURN, ARGUMENTS, BODY) \
RETURN NAME ARGUMENTS { BODY; } \
__inline__ __device__ RETURN PARACUDA##NAME##_operator
    ARGUMENTS { BODY; }

#endif
```

B.9 paracuda_struct.h

```
#ifndef PARACUDA_STRUCT_H
#define PARACUDA_STRUCT_H

#define PARACUDA_SINGLE(T0) \
typedef T0 PARACUDA##T0##_struct; \
__inline__ __device__ \
void PARACUDA##T0##_from_vector(T0* v, T0* r, size_t index
    ) \
{ \
    *v = r[index]; \
} \
void PARACUDA##T0##_from_vector_host(T0* v, T0* r, size_t
    index) \
```

```
{ \
    *v = r[index]; \
} \
__inline__ __device__ \
void PARACUDA##T0##_to_vector(T0* r, T0* v, size_t index)
    \
{ \
  r[index] = *v;                                    \
} \
/* TODO: eliminate this */ \
void PARACUDA##T0##_to_vector_poke(T0* r, T0* v, size_t
    index) \
{ \
    cutilSafeCall(cudaMemcpy(&(r[index]), v, sizeof(T0),
        cudaMemcpyHostToDevice)); \
} \
void PARACUDA##T0##_to_vector_host(T0* r, T0* v, size_t
    index) \
{ \
    r[index] = *v; \
} \
T0* PARACUDA##T0##_allocate_host(size_t length) \
{ \
    T0* r = (T0*) malloc(length * sizeof(T0)); \
    return r; \
} \
T0* PARACUDA##T0##_allocate_device(size_t length) \
{ \
  T0* r ; \
  cutilSafeCall(cudaMalloc((void**) &(r), length * sizeof(
      T0)));\
  return r; \
} \
void PARACUDA##T0##_free_device(T0* r) \
{ \
    cutilSafeCall(cudaFree(r)); \
} \
void PARACUDA##T0##_free_host(T0* r) \
{ \
    free(r); \
} \
void PARACUDA##T0##_copy_host_device(T0* out, T0* in,
    size_t length) \
{ \
    cutilSafeCall(cudaMemcpy(out, in, length * sizeof(T0),
        cudaMemcpyHostToDevice)); \
} \
void PARACUDA##T0##_copy_device_host(T0* out, T0* in,
    size_t length) \
{ \
    cutilSafeCall(cudaMemcpy(out, in, length * sizeof(T0),
        cudaMemcpyDeviceToHost)); \
} \
void PARACUDA##T0##_copy_device_device(T0* out, T0* in,
    size_t length) \
{ \
    cutilSafeCall(cudaMemcpy(out, in, length * sizeof(T0),
        cudaMemcpyDeviceToDevice)); \
} \
void PARACUDA##T0##_shallow_copy_host_device(T0* out, T0*
    in) \
{ \
```

```
    cutilSafeCall(cudaMemcpy(&out, &in, sizeof(T0*),
        cudaMemcpyHostToDevice)); \
} \
void PARACUDA##T0##_shallow_copy_device_host(T0* out, T0*
    in) \
{ \
    cutilSafeCall(cudaMemcpy(&out, &in, sizeof(T0*),
        cudaMemcpyDeviceToHost)); \
} \
T0* PARACUDA##T0##_shallow_allocate_device() \
{ \
    T0* r; \
    cutilSafeCall(cudaMalloc((void**) &r, sizeof(T0))); \
    return r; \
} \
void PARACUDA##T0##_shallow_free_device(T0* r) \
{ \
    cutilSafeCall(cudaFree(r)); \
} \
 void PARACUDA##T0##_peek(T0* out, T0* in, size_t index)
          \
{ \
    cutilSafeCall(cudaMemcpy(out, in + index, sizeof(T0),
        cudaMemcpyDeviceToHost)); \
} \
int PARACUDA##T0##_equal(T0* a, int ai, T0* b, int bi) \
{ \
    return a[ai] == b[bi]; \
} \
 void PARACUDA##T0##_add_to_device(T0 value, T0* array,
     size_t position) { \
  T0 temp;\
  cutilSafeCall(cudaMemcpy(&temp, &(array[position]),
      sizeof(T0), cudaMemcpyDeviceToHost)); \
  value += temp;\
  cutilSafeCall(cudaMemcpy(&(array[position]),&value,
      sizeof(T0), cudaMemcpyHostToDevice)); \
}\

#define PARACUDA_STRUCT_2(NAME, VECTOR, T0, N0, T1, N1) \
typedef struct NAME NAME; \
struct NAME { T0 N0; T1 N1; }; \
typedef struct VECTOR PARACUDA##NAME##_struct; \
typedef struct VECTOR VECTOR; \
struct VECTOR { T0* N0; T1* N1; }; \
__inline__ __device__ \
void PARACUDA##NAME##_from_vector(NAME* v, VECTOR* r,
    size_t index) \
{ \
    v->N0 = r->N0[index]; \
    v->N1 = r->N1[index]; \
} \
void PARACUDA##NAME##_from_vector_host(NAME* v, VECTOR* r,
     size_t index) \
{ \
    v->N0 = r->N0[index]; \
    v->N1 = r->N1[index]; \
} \
__inline__ __device__ \
void PARACUDA##NAME##_to_vector(VECTOR* r, NAME* v, size_t
     index) \
{ \
```

```
    r->N0[index] = v->N0; \
    r->N1[index] = v->N1; \
} \
void PARACUDA##NAME##_to_vector_host(VECTOR* r, NAME* v,
    size_t index) \
{ \
    r->N0[index] = v->N0; \
    r->N1[index] = v->N1; \
} \
VECTOR* PARACUDA##NAME##_allocate_host(size_t length) \
{ \
    VECTOR* r = (VECTOR*) malloc(sizeof(VECTOR)); \
    r->N0 = (T0*) malloc(length * sizeof(T0)); \
    r->N1 = (T1*) malloc(length * sizeof(T1)); \
    return r; \
} \
VECTOR* PARACUDA##NAME##_allocate_device(size_t length) \
{ \
    VECTOR t; \
    cutilSafeCall(cudaMalloc((void**) &(t.N0), length *
        sizeof(T0))); \
    cutilSafeCall(cudaMalloc((void**) &(t.N1), length *
        sizeof(T1))); \
    VECTOR* r; \
    cutilSafeCall(cudaMalloc((void**) &r, sizeof(VECTOR)));
        \
    cutilSafeCall(cudaMemcpy(r, &t, sizeof(VECTOR),
        cudaMemcpyHostToDevice)); \
    return r; \
} \
void PARACUDA##NAME##_free_device(VECTOR* r) \
{ \
    VECTOR t; \
    cutilSafeCall(cudaMemcpy(&t, r, sizeof(VECTOR),
        cudaMemcpyDeviceToHost)); \
    cutilSafeCall(cudaFree(t.N0)); \
    cutilSafeCall(cudaFree(t.N1)); \
    cutilSafeCall(cudaFree(r)); \
} \
void PARACUDA##NAME##_free_host(VECTOR* r) \
{ \
    free(r->N0); \
    free(r->N1); \
    free(r); \
} \
void PARACUDA##NAME##_copy_host_device(VECTOR* out, VECTOR
    * in, size_t length) \
{ \
    VECTOR temp; \
    cutilSafeCall(cudaMemcpy(&temp, out, sizeof(VECTOR),
        cudaMemcpyDeviceToHost)); \
    cutilSafeCall(cudaMemcpy(temp.N0, in->N0, length *
        sizeof(T0), cudaMemcpyHostToDevice)); \
    cutilSafeCall(cudaMemcpy(temp.N1, in->N1, length *
        sizeof(T1), cudaMemcpyHostToDevice)); \
} \
void PARACUDA##NAME##_copy_device_host(VECTOR* out, VECTOR
    * in, size_t length) \
{ \
    VECTOR temp; \
    cutilSafeCall(cudaMemcpy(&temp, in, sizeof(VECTOR),
        cudaMemcpyDeviceToHost));                \
```

```
    cutilSafeCall(cudaMemcpy(out->N0, temp.N0, length *
        sizeof(T0), cudaMemcpyDeviceToHost)); \
    cutilSafeCall(cudaMemcpy(out->N1, temp.N1, length *
        sizeof(T1), cudaMemcpyDeviceToHost)); \
} \
void PARACUDA##NAME##_copy_device_device(VECTOR* out,
    VECTOR* in, size_t length) \
{ \
    VECTOR output; \
    VECTOR input; \
    cutilSafeCall(cudaMemcpy(&input, in, sizeof(VECTOR),
        cudaMemcpyDeviceToHost));            \
    cutilSafeCall(cudaMemcpy(&output, out, sizeof(VECTOR),
        cudaMemcpyDeviceToHost));          \
    cutilSafeCall(cudaMemcpy(output.N0, input.N0, length *
        sizeof(T0), cudaMemcpyDeviceToDevice)); \
    cutilSafeCall(cudaMemcpy(output.N1, input.N1, length *
        sizeof(T1), cudaMemcpyDeviceToDevice)); \
} \
void PARACUDA##NAME##_shallow_copy_host_device(VECTOR* out
    , VECTOR* in) \
{ \
    cutilSafeCall(cudaMemcpy(out, in, sizeof(VECTOR),
        cudaMemcpyHostToDevice)); \
} \
void PARACUDA##NAME##_shallow_copy_device_host(VECTOR* out
    , VECTOR* in) \
{ \
    cutilSafeCall(cudaMemcpy(out, in, sizeof(VECTOR),
        cudaMemcpyDeviceToHost)); \
} \
VECTOR* PARACUDA##NAME##_shallow_allocate_device() \
{ \
    VECTOR* r; \
    cutilSafeCall(cudaMalloc((void**) &r, sizeof(VECTOR)));
        \
    return r; \
} \
void PARACUDA##NAME##_shallow_free_device(VECTOR* r) \
{ \
    cutilSafeCall(cudaFree(r)); \
} \
void PARACUDA##NAME##_to_vector_poke(VECTOR* r, NAME* v,
    size_t index) \
{ \
    cutilSafeCall(cudaMemcpy(&(r->N0[index]), &(v->N0),
        sizeof(T0), cudaMemcpyHostToDevice)); \
    cutilSafeCall(cudaMemcpy(&(r->N1[index]), &(v->N1),
        sizeof(T1), cudaMemcpyHostToDevice)); \
} \
void PARACUDA##NAME##_peek(NAME* r, VECTOR* v, size_t
    index)   \
{ \
    VECTOR temp; \
    cutilSafeCall(cudaMemcpy(&temp, v, sizeof(VECTOR),
        cudaMemcpyDeviceToHost));      \
    cutilSafeCall(cudaMemcpy(&(r->N0), temp.N0 + index,
        sizeof(T0), cudaMemcpyDeviceToHost)); \
    cutilSafeCall(cudaMemcpy(&(r->N1), temp.N1 + index,
        sizeof(T1), cudaMemcpyDeviceToHost)); \
} \
```

```
int PARACUDA_##NAME##_equal(VECTOR* a, int ai, VECTOR* b,
    int bi) \
{ \
    return a->N0[ai] == b->N0[bi] && a->N1[ai] == b->N1[bi
        ]; \
} \


#define PARACUDA_STRUCT_3(NAME, VECTOR, T0, N0, T1, N1, T2,
     N2) \
typedef struct NAME NAME; \
struct NAME { T0 N0; T1 N1; T2 N2; };   \
typedef struct VECTOR PARACUDA_##NAME##_struct; \
typedef struct VECTOR VECTOR; \
struct VECTOR { T0* N0; T1* N1; T2* N2; };      \
__inline__ __device__ \
void PARACUDA_##NAME##_from_vector(NAME* v, VECTOR* r,
    size_t index) \
{ \
    v->N0 = r->N0[index]; \
    v->N1 = r->N1[index]; \
    v->N2 = r->N2[index]; \
} \
void PARACUDA_##NAME##_from_vector_host(NAME* v, VECTOR* r,
     size_t index) \
{ \
    v->N0 = r->N0[index]; \
    v->N1 = r->N1[index]; \
    v->N2 = r->N2[index]; \
} \
__inline__ __device__ \
void PARACUDA_##NAME##_to_vector(VECTOR* r, NAME* v, size_t
     index) \
{ \
    r->N0[index] = v->N0; \
    r->N1[index] = v->N1; \
    r->N2[index] = v->N2; \
} \
void PARACUDA_##NAME##_to_vector_host(VECTOR* r, NAME* v,
    size_t index) \
{ \
    r->N0[index] = v->N0; \
    r->N1[index] = v->N1; \
    r->N2[index] = v->N2; \
} \
VECTOR* PARACUDA_##NAME##_allocate_host(size_t length) \
{ \
    VECTOR* r = (VECTOR*) malloc(sizeof(VECTOR)); \
    r->N0 = (T0*) malloc(length * sizeof(T0)); \
    r->N1 = (T1*) malloc(length * sizeof(T1)); \
    r->N2 = (T2*) malloc(length * sizeof(T2)); \
    return r; \
} \
VECTOR* PARACUDA_##NAME##_allocate_device(size_t length) \
{ \
    VECTOR t; \
    cutilSafeCall(cudaMalloc((void**) &(t.N0), length *
        sizeof(T0))); \
    cutilSafeCall(cudaMalloc((void**) &(t.N1), length *
        sizeof(T1))); \
    cutilSafeCall(cudaMalloc((void**) &(t.N2), length *
        sizeof(T2))); \
```

```
    VECTOR* r; \
    cutilSafeCall(cudaMalloc((void**) &r, sizeof(VECTOR))); \
    cutilSafeCall(cudaMemcpy(r, &t, sizeof(VECTOR), \
        cudaMemcpyHostToDevice)); \
    return r; \
} \
void PARACUDA##NAME##_free_device(VECTOR* r) \
{ \
    VECTOR t; \
    cutilSafeCall(cudaMemcpy(&t, r, sizeof(VECTOR), \
        cudaMemcpyDeviceToHost)); \
    cutilSafeCall(cudaFree(&(r->N0)));\
    cutilSafeCall(cudaFree(&(r->N1)));\
    cutilSafeCall(cudaFree(&(r->N2)));\
    cutilSafeCall(cudaFree(r)); \
} \
void PARACUDA##NAME##_free_host(VECTOR* r) \
{ \
    free(r->N0); \
    free(r->N1); \
    free(r->N2); \
    free(r); \
} \
void PARACUDA##NAME##_copy_host_device(VECTOR* out, VECTOR
    * in, size_t length) \
{ \
    VECTOR temp; \
    cutilSafeCall(cudaMemcpy(&temp, out, sizeof(VECTOR), \
        cudaMemcpyDeviceToHost)); \
    cutilSafeCall(cudaMemcpy(&(out->N0), &(in->N0), length \
        * sizeof(T0), cudaMemcpyHostToDevice)); \
    cutilSafeCall(cudaMemcpy(&(out->N1), &(in->N1), length \
        * sizeof(T1), cudaMemcpyHostToDevice)); \
    cutilSafeCall(cudaMemcpy(&(out->N2), &(in->N2), length \
        * sizeof(T2), cudaMemcpyHostToDevice)); \
} \
void PARACUDA##NAME##_copy_device_host(VECTOR* out, VECTOR
    * in, size_t length) \
{ \
    VECTOR temp; \
    cutilSafeCall(cudaMemcpy(&temp, in, sizeof(VECTOR), \
        cudaMemcpyDeviceToHost));                \
    cutilSafeCall(cudaMemcpy(out->N0, temp.N0, length * \
        sizeof(T0), cudaMemcpyDeviceToHost)); \
    cutilSafeCall(cudaMemcpy(out->N1, temp.N1, length * \
        sizeof(T1), cudaMemcpyDeviceToHost)); \
    cutilSafeCall(cudaMemcpy(out->N2, temp.N2, length * \
        sizeof(T2), cudaMemcpyDeviceToHost)); \
} \
void PARACUDA##NAME##_copy_device_device(VECTOR* out, \
    VECTOR* in, size_t length) \
{ \
    VECTOR output; \
    VECTOR input; \
    cutilSafeCall(cudaMemcpy(&input, in, sizeof(VECTOR), \
        cudaMemcpyDeviceToHost));                \
    cutilSafeCall(cudaMemcpy(&output, out, sizeof(VECTOR), \
        cudaMemcpyDeviceToHost));                \
    cutilSafeCall(cudaMemcpy(output.N0, input.N0, length * \
        sizeof(T0), cudaMemcpyDeviceToDevice)); \
```

```
    cutilSafeCall(cudaMemcpy(output.N1, input.N1, length *
        sizeof(T1), cudaMemcpyDeviceToDevice)); \
    cutilSafeCall(cudaMemcpy(output.N2, input.N2, length *
        sizeof(T2), cudaMemcpyDeviceToDevice)); \
} \
void PARACUDA##NAME##_shallow_copy_host_device(VECTOR* out
    , VECTOR* in) \
{ \
    cutilSafeCall(cudaMemcpy(out, in, sizeof(VECTOR),
        cudaMemcpyHostToDevice)); \
} \
void PARACUDA##NAME##_shallow_copy_device_host(VECTOR* out
    , VECTOR* in) \
{ \
    cutilSafeCall(cudaMemcpy(out, in, sizeof(VECTOR),
        cudaMemcpyDeviceToHost)); \
} \
VECTOR* PARACUDA##NAME##_shallow_allocate_device() \
{ \
    VECTOR* r; \
    cutilSafeCall(cudaMalloc((void**) &r, sizeof(VECTOR))); 
        \
    return r; \
} \
void PARACUDA##NAME##_shallow_free_device(VECTOR* r) \
{ \
    cutilSafeCall(cudaFree(r)); \
} \
void PARACUDA##NAME##_peek(NAME* r, VECTOR* v, size_t
    index)   \
{ \
    VECTOR temp; \
    cutilSafeCall(cudaMemcpy(&temp, v, sizeof(VECTOR),
        cudaMemcpyDeviceToHost));     \
    cutilSafeCall(cudaMemcpy(&(r->N0), temp.N0 + index,
        sizeof(T0), cudaMemcpyDeviceToHost)); \
    cutilSafeCall(cudaMemcpy(&(r->N1), temp.N1 + index,
        sizeof(T1), cudaMemcpyDeviceToHost)); \
    cutilSafeCall(cudaMemcpy(&(r->N2), temp.N2 + index,
        sizeof(T2), cudaMemcpyDeviceToHost)); \
} \
int PARACUDA##NAME##_equal(VECTOR* a, int ai, VECTOR* b,
    int bi) \
{ \
    return a->N0[ai] == b->N0[bi] && a->N1[ai] == b->N1[bi]
         && a->N2[ai] == b->N2[bi]; \
} \


#endif
```

B.10 paracuda_map.h

```
#ifndef PARACUDA_MAP_H
#define PARACUDA_MAP_H

#include <cutil_inline.h>

/* Defines the parallel algorithm. */
#define PARACUDA_MAP(NAME, OPERATOR, OUTPUT, INPUT) \
/* The part that is run in parralel. Output and input
    should be on the device. */ \
```

```
/* You don't normally need to call this directly. */ \
__global__ \
 void PARACUDA##NAME##_kernel(PARACUDA##OUTPUT##_struct*
     output, PARACUDA##INPUT##_struct* input, size_t
     length) \
{ \
  /*__shared__ PARACUDA##INPUT##_struct s_data[512];*/ \
    size_t nomialOffset = blockIdx.x * blockDim.x +
        threadIdx.x; \
    OUTPUT out; \
    INPUT in; \
    if (length <= nomialOffset) return; \
    PARACUDA##INPUT##_from_vector(&in, input, nomialOffset
        );\
    PARACUDA##OPERATOR##_operator(&out, &in);\
    PARACUDA##OUTPUT##_to_vector(output, &out,
        nomialOffset);\
} \
/* Prepares the kernel and calls it. Output and input
    should be on the device. */ \
__host__ \
void PARACUDA##NAME##_run(PARACUDA##OUTPUT##_struct*
    output, PARACUDA##INPUT##_struct* input, size_t length
    , size_t max_threads) \
{ \
    int numBlocks = length / max_threads;\
    if (numBlocks == 0) numBlocks = 1;\
    if (length < max_threads) max_threads = length;\
    int numThreadsPerBlock = max_threads;\
    dim3 dim(numThreadsPerBlock);\
    PARACUDA##NAME##_kernel<<<numBlocks, dim>>>(output,
        input, length);\
    cutilCheckMsg("Map"); \
    cudaThreadSynchronize();\
} \
\
/* Executes the algorithm and returns the output. */ \
/* It is OK to have output == input. */ \
/* If output == NULL, memory will be allocated for it. */ \
__host__ \
PARACUDA##OUTPUT##_struct* NAME(PARACUDA##OUTPUT##_struct
    * output, PARACUDA##INPUT##_struct* input, size_t
    length, size_t max_threads) \
{ \
  /*PARACUDA_ASSERT_POWER_OF_TWO(length);*/\
  if(output == 0) output = PARACUDA##OUTPUT##
      _allocate_host(length);\
  PARACUDA##INPUT##_struct* device_input = PARACUDA##
      INPUT##_allocate_device(length); \
  PARACUDA##INPUT##_copy_host_device(device_input, input,
      length); \
  if((void*) input == (void*) output)\
  {\
    PARACUDA##NAME##_run((PARACUDA##OUTPUT##_struct*)
        device_input, device_input, length, max_threads); \
    PARACUDA##OUTPUT##_copy_device_host(output, (PARACUDA_
        ##OUTPUT##_struct*) device_input, length); \
  }\
  else\
  {\
    PARACUDA##OUTPUT##_struct* device_output = PARACUDA##
        OUTPUT##_allocate_device(length); \
```

```
    PARACUDA##NAME##_run(device_output, device_input,
        length, max_threads); \
    PARACUDA##OUTPUT##_copy_device_host(output,
        device_output, length); \
    PARACUDA##OUTPUT##_free_device(device_output);\
  }\
  PARACUDA##INPUT##_free_device(device_input); \
  cutilCheckMsg("Map"); \
  return output; \
} \

#endif
```

B.11 paracuda_permute.h

```
#ifndef PARACUDA_PERMUTE_H
#define PARACUDA_PERMUTE_H

#include <cutil_inline.h>

/* Defines the parallel algorithm. */
#define PARACUDA_PERMUTE(NAME, TYPE) \
/* The part that is run in parralel. Output and input
    should be on the device. */ \
/* You don't normally need to call this directly. */ \
__global__ \
 void PARACUDA##NAME##_kernel(PARACUDA##TYPE##_struct*
     output, PARACUDA##TYPE##_struct* input, int*
     positions, size_t length) \
{ \
    size_t offset = blockIdx.x * blockDim.x + threadIdx.x;
        \
    TYPE temp; \
    if (length <= offset) return;        \
    PARACUDA##TYPE##_from_vector(&temp, input, offset); \
    PARACUDA##TYPE##_to_vector(output, &temp, positions[
        offset]); \
} \
/* Prepares the kernel and calls it. Output and input
    should be on the device. */ \
__host__ \
void PARACUDA##NAME##_run(PARACUDA##TYPE##_struct* output
    , PARACUDA##TYPE##_struct* input, int* positions,
    size_t length, size_t max_threads) \
{ \
    int numBlocks = length / max_threads;\
    if (numBlocks == 0) numBlocks = 1;\
    if (length < max_threads) max_threads = length;\
    int numThreadsPerBlock = max_threads;\
    dim3 dim(numThreadsPerBlock);\
    PARACUDA##NAME##_kernel<<<numBlocks, dim>>>(output,
        input, positions, length); \
    cutilCheckMsg("Permute"); \
} \
\
/* Executes the algorithm and returns the output. */ \
/* It is OK to have output == input. */ \
/* If output == NULL, memory will be allocated for it. */ \
__host__ \
PARACUDA##TYPE##_struct* NAME(PARACUDA##TYPE##_struct*
    output, PARACUDA##TYPE##_struct* input, int* positions
    , size_t length, size_t max_threads) \
```

```
{ \
    PARACUDA_ASSERT_POWER_OF_TWO(length); \
    if(output == 0) output = PARACUDA##TYPE##
        _allocate_host(length); \
    \
    PARACUDA##TYPE##_struct* device_output = PARACUDA##
        TYPE##_allocate_device(length); \
    PARACUDA##TYPE##_struct* device_input = PARACUDA##
        TYPE##_allocate_device(length); \
    int* device_positions = PARACUDA_int_allocate_device(
        length); \
    \
    PARACUDA##TYPE##_copy_host_device(device_input, input,
         length); \
    PARACUDA_int_copy_host_device(device_positions,
        positions, length); \
    \
    PARACUDA##NAME##_run(device_output, device_input,
        device_positions, length, max_threads); \
    PARACUDA##TYPE##_copy_device_host(output,
        device_output, length); \
    PARACUDA##TYPE##_free_device(device_output);\
    PARACUDA##TYPE##_free_device(device_input);\
    PARACUDA_int_free_device(device_positions); \
    return output; \
} \


#endif
```

B.12 paracuda_scan.h

```
#ifndef PARACUDA_SCAN_H
#define PARACUDA_SCAN_H

#include <cutil_inline.h>

/* Defines the parallel algorithm. */
#define PARACUDA_SCAN(NAME, OPERATOR, NEUTRAL, TYPE)\
/* The part that is run in parralel. Output and input
    should be on the device. */ \
/* You don't normally need to call this directly. */ \
__global__ \
 void PARACUDA##NAME##_upsweep_kernel(PARACUDA##TYPE##
     _struct* array, size_t length, size_t d_stop, size_t
     max_threads) \
{ \
    for(int d = 0; d < d_stop; ++d) \
    { \
      __syncthreads(); \
      int thread_count = (int) floorf((length - 1) / (1 <<
          (d + 1))) + 1; \
      int job_count = 1; \
      while(thread_count > max_threads) \
        { \
            thread_count >>= 1; \
            job_count <<= 1; \
        } \
      size_t offset = threadIdx.x * job_count; \
      size_t step_child = 1 << (d + 1); \
      size_t step_self =  1 << d; \
```

```
      for(size_t job = 0; job < job_count; ++job) \
        { \
          __syncthreads(); \
          if (threadIdx.x < thread_count) { \
          size_t k = (offset + job) * step_child; \
          \
          size_t self = k + step_self - 1; \
          size_t child = k + step_child - 1; \
          \
          TYPE out; \
          TYPE self_in; \
          TYPE child_in; \
          PARACUDA_##TYPE##_from_vector(&self_in, array, \
              self); \
          PARACUDA_##TYPE##_from_vector(&child_in, array, \
              child); \
          PARACUDA_##OPERATOR##_operator(&out, &self_in, & \
              child_in); \
          PARACUDA_##TYPE##_to_vector(array, &out, child); \
          \
          } \
        } \
    } \
} \
__global__ \
 void PARACUDA_##NAME##_downsweep_kernel(PARACUDA_##TYPE## \
     _struct* array, size_t length, int d_start, int \
     max_threads) \
{\
for(int d = d_start; d >= 0; d--) {\
    __syncthreads();\
    int thread_count = (int) floorf(((length - 1) / (1 << ( \
        d + 1)))) + 1; \
    int job_count = 1;\
    while(thread_count > max_threads)\
      {\
        thread_count >>= 1;\
        job_count <<= 1;\
      }\
    \
    size_t offset = threadIdx.x * job_count; \
    size_t step_child = 1 << (d + 1); \
    size_t step_self = 1 << d; \
    for(size_t job = 0; job < job_count; ++job) \
      {\
        __syncthreads();\
        if(threadIdx.x < thread_count) { \
          size_t k = (offset + job) * step_child;\
          \
          size_t self = k + step_self - 1;\
          size_t child = k + step_child - 1;\
          \
          TYPE out;\
          TYPE self_in;\
          TYPE child_in;\
          PARACUDA_##TYPE##_from_vector(&self_in, array, \
              self); \
          PARACUDA_##TYPE##_from_vector(&child_in, array, \
              child);\
          PARACUDA_##OPERATOR##_operator(&out, &self_in, & \
              child_in);\
```

```
            PARACUDA##TYPE##_to_vector(array, &child_in,
                self);\
            PARACUDA##TYPE##_to_vector(array, &out, child);\
          }\
        }\
    }\
} \
 \
/* Prepares the kernel and calls it. Output and input
    should be on the device. */ \
/* You don't normally need to call this directly. */ \
__host__ \
void PARACUDA##NAME##_upsweep(PARACUDA##TYPE##_struct*
    array, size_t length, size_t max_threads) \
{ \
    if(length == 0) return; \
    dim3 grid(1, 1, 1); \
    if(length < max_threads) max_threads = length; \
    dim3 threads(max_threads, 1, 1); \
    int d_stop = PARACUDA_LOG2(length) -1; \
    PARACUDA##NAME##_upsweep_kernel<<<grid, threads>>>(
        array, length, d_stop, max_threads); \
    } \
void PARACUDA##NAME##_downsweep(PARACUDA##TYPE##_struct*
    array, size_t length, size_t max_threads) \
{ \
    if(length == 0) return; \
    /* TODO: Consider moving the neutralization to the
        kernel to avoid memcopy from host to device */ \
    TYPE out; \
    NEUTRAL(&out);\
    PARACUDA##TYPE##_to_vector_poke(array, &out, length -
        1); \
    dim3 grid(1, 1, 1); \
    const int d_start = PARACUDA_LOG2(length) - 1;\
    \
    if (length < max_threads) max_threads = length; \
    dim3 threads(max_threads, 1, 1);\
    PARACUDA##NAME##_downsweep_kernel<<<grid, threads>>>(
        array, length, d_start, max_threads); \
} \
 \
/* On-device memory assumed. In place. */\
__host__ \
 void PARACUDA##NAME##_run(TYPE* sum, PARACUDA##TYPE##
     _struct* array, size_t length, size_t max_threads) \
{\
  if(sum != 0) PARACUDA##TYPE##_peek(sum, array, length -
      1); \
    PARACUDA##NAME##_upsweep(array, length, max_threads);
        \
    PARACUDA##NAME##_downsweep(array, length, max_threads)
        ; \
    cutilCheckMsg("Scan"); \
    if(sum != 0) {                    \
      TYPE last;\
      PARACUDA##TYPE##_peek(&last, array, length - 1); \
      OPERATOR(sum, &last, sum);\
    }\
}\
/* Executes the algorithm and returns the output. */ \
/* It is OK to have output == input. */ \
```

```
/* If output == NULL, memory will be allocated for it. */ \
__host__ \
PARACUDA##TYPE##_struct* NAME(TYPE* sum, PARACUDA##TYPE##
    _struct* output, PARACUDA##TYPE##_struct* input,
    size_t length, size_t max_threads) \
{\
    PARACUDA_ASSERT_POWER_OF_TWO(length);\
    if(sum != 0) PARACUDA##TYPE##_from_vector_host(sum,
        input, length - 1); \
    if(output == 0) output = PARACUDA##TYPE##
        _allocate_host(length); \
    PARACUDA##TYPE##_struct* device_array = PARACUDA##
        TYPE##_allocate_device(length); \
    PARACUDA##TYPE##_copy_host_device(device_array, input,
         length); \
    \
    PARACUDA##NAME##_run(sum, device_array, length,
        max_threads);       \
    \
    PARACUDA##TYPE##_copy_device_host(output, device_array
        , length); \
    PARACUDA##TYPE##_free_device(device_array);
                        \
    if(sum != 0) { \
        TYPE last; \
        PARACUDA##TYPE##_from_vector_host(&last, output,
            length - 1); \
        OPERATOR(sum, &last, sum);\
    }\
    return output;\
} \

#endif
```

B.13 paracuda_segmentedScan.h

```
#ifndef PARACUDA_SEGMENTEDSCAN_H
#define PARACUDA_SEGMENTEDSCAN_H
#include "arrayprint.h"

#include <cutil_inline.h>

/* Defines the parallel algorithm. */
#define PARACUDA_SEGMENTEDSCAN(NAME, OPERATOR, NEUTRAL,
    TYPE)           \
/* The part that is run in parralel. Output and input
    should be on the device. */ \
/* You don't normally need to call this directly. */ \
__global__ \
 void PARACUDA##NAME##_upsweep_kernel(PARACUDA##TYPE##
     _struct* array, int* flags, int* iFlags, size_t length
     , size_t d_stop, size_t max_threads) \
{ \
 for(int d = 0; d <= d_stop; d++) {\
    int thread_count = (int) floorf(((length - 1) / (1 << (
        d + 1)))) + 1; \
    int job_count = 1;\
    while(thread_count > max_threads)\
      {\
        thread_count >>= 1;\
        job_count <<= 1;\
```

```
      }\
    \
    size_t offset = threadIdx.x * job_count; \
    size_t step_child = 1 << (d + 1); \
    size_t step_self = 1 << d; \
    for(size_t job = 0; job < job_count; ++job)\
        {\
          __syncthreads();\
          size_t k = (offset + job) * step_child;\
          if (k < length) {\
          size_t self = k + step_self - 1;\
          size_t child = k + step_child - 1;\
          \
          TYPE out;\
          TYPE self_in;\
          TYPE child_in;\
          if(!flags[child]) {\
            PARACUDA_##TYPE##_from_vector(&self_in, array,
                self); \
            PARACUDA_##TYPE##_from_vector(&child_in, array,
                 child); \
            PARACUDA_##OPERATOR##_operator(&out, &self_in,
                &child_in);\
            PARACUDA_##TYPE##_to_vector(array, &out, child)
                ;\
          }\
          flags[child] = flags[self] | flags[child];\
          }\
        }\
    }\
}\
__global__ \
 void PARACUDA_##NAME##_downsweep_kernel(PARACUDA_##TYPE##
     _struct* sumArray, PARACUDA_##TYPE##_struct* array,
     int* flags, int* iFlag, int* sumArrayIndex, size_t
     length, int d_start, int max_threads) \
{\
for(int d = d_start; d >= 0; d--) {\
    __syncthreads();\
    int thread_count = (int) floorf(((length - 1) / (1 << (
        d + 1)))) + 1; \
    int job_count = 1;\
    while(thread_count > max_threads)\
      {\
        thread_count >>= 1;\
        job_count <<= 1;\
      }\
    \
    size_t offset = threadIdx.x * job_count; \
    size_t step_child = 1 << (d + 1); \
    size_t step_self = 1 << d; \
    for(size_t job = 0; job < job_count; ++job) \
      {\
        __syncthreads();\
        size_t k = (offset + job) * step_child;\
        if(k < length) { \
          \
          size_t self = k + step_self - 1;\
          size_t child = k + step_child - 1;\
          \
          TYPE out;\
          TYPE child_in;\
```

```
            TYPE temp; \
            TYPE backup;\
            \
            PARACUDA_##TYPE##_from_vector(&backup, array,
                self); \
            \
            PARACUDA_##TYPE##_from_vector(&child_in, array,
                child);\
            PARACUDA_##TYPE##_to_vector(array, &child_in,
                self); \
            if (iFlag[self + 1]) {\
              if(sumArray) { \
                PARACUDA_##TYPE##_from_vector(&child_in,
                    array, child);\
                PARACUDA_##OPERATOR##_operator(&out, &backup,
                     &child_in); \
                PARACUDA_##TYPE##_to_vector(sumArray, &out,
                    sumArrayIndex[self]); \
              } \
              PARACUDA_##NEUTRAL##_operator(&temp);\
              PARACUDA_##TYPE##_to_vector(array, &temp, child
                  );\
            } else if (flags[self]) {\
              PARACUDA_##TYPE##_to_vector(array, &backup,
                  child);\
            } else {\
              PARACUDA_##TYPE##_from_vector(&child_in, array,
                   child);\
              PARACUDA_##OPERATOR##_operator(&out, &backup, &
                  child_in); \
              PARACUDA_##TYPE##_to_vector(array, &out, child)
                  ;\
            }\
            flags[self] = 0; \
          }\
        }\
    }\
} \
 \
/* Prepares the kernel and calls it. Output and input
    should be on the device. */ \
/* You don't normally need to call this directly. */ \
__host__ \
void PARACUDA_##NAME##_upsweep(PARACUDA_##TYPE##_struct*
    array, int* flags, int* iFlags, size_t length, size_t
    max_threads) \
{\
    if(length == 0) return; \
    if(length < max_threads) max_threads = length; \
    dim3 grid(1, 1, 1); \
    dim3 threads(max_threads, 1, 1); \
    int d_stop = PARACUDA_LOG2(length) -1; \
    PARACUDA_##NAME##_upsweep_kernel<<<grid, threads>>>(
        array, flags, iFlags, length, d_stop, max_threads);
         \
} \
void PARACUDA_##NAME##_downsweep(PARACUDA_##TYPE##_struct*
    sumArray, PARACUDA_##TYPE##_struct* array, int* flags,
    int* iFlag, int* sumArrayIndex, size_t length, size_t
    max_threads) \
{ \
    if(length == 0) return; \
```

```
    /* TODO: Consider moving the neutralization to the
        kernel to avoid memcopy from host to device */ \
    \
    TYPE out; \
    NEUTRAL(&out);\
    const int d_start = PARACUDA_LOG2(length) - 1;\
    PARACUDA##TYPE##_to_vector_poke(array, &out, length -
        1); \
    if (length < max_threads) max_threads = length; \
    dim3 grid(1, 1, 1); \
    dim3 threads(max_threads, 1, 1);\
    PARACUDA##NAME##_downsweep_kernel<<<grid, threads>>>(
        sumArray, array, flags, iFlag, sumArrayIndex,
        length, d_start, max_threads); \
} \
 \
/* On-device memory assumed. In place. A copy of flags must
     be passed as flags_copy, and it will be scrambled. */\
__host__ \
void PARACUDA##NAME##_run(PARACUDA##TYPE##_struct*
    sumArray, PARACUDA##TYPE##_struct* array, int* flags,
    int* flags_copy, int* sumArrayIndex, size_t length,
    size_t max_threads) \
{\
  PARACUDA##NAME##_upsweep(array, flags_copy, flags,
      length, max_threads);      \
    PARACUDA##NAME##_downsweep(sumArray, array, flags_copy
        , flags, sumArrayIndex, length, max_threads); \
    cutilCheckMsg("Segmented scan");\
}\
/* Executes the algorithm and returns the output. */ \
/* It is OK to have output == input. */ \
/* If output == NULL, memory will be allocated for it. */ \
__host__ \
 PARACUDA##TYPE##_struct* NAME(PARACUDA##TYPE##_struct*
     sumArray, PARACUDA##TYPE##_struct* output, PARACUDA_
     ##TYPE##_struct* input, int* flags, int* sumArrayIndex
     , size_t length, size_t max_threads) \
{\
    PARACUDA_ASSERT_POWER_OF_TWO(length);\
    if(output == 0) output = PARACUDA##TYPE##
        _allocate_host(length); \
    PARACUDA##TYPE##_struct* device_array = PARACUDA##
        TYPE##_allocate_device(length); \
    \
    int arrayLength = (sumArray) ? length : 0;
                \
    int* device_flags      = PARACUDA_int_allocate_device(
        length);\
    int* device_iFlags     = PARACUDA_int_allocate_device(
        length);\
    int* device_sumArray   = PARACUDA_int_allocate_device(
        arrayLength);\
    int* device_arrayIndex = PARACUDA_int_allocate_device(
        length);\
    if(sumArray) {\
      PARACUDA##TYPE##_copy_host_device(device_sumArray,
          sumArray, length); \
      PARACUDA##TYPE##_copy_host_device(device_arrayIndex,
          sumArrayIndex, length);\
    }\
    \
```

```
    PARACUDA##TYPE##_copy_host_device(device_array, input,
        length); \
    PARACUDA_int_copy_host_device(device_flags, flags,
        length); \
    PARACUDA_int_copy_host_device(device_iFlags, flags,
        length);\
      \
    \
    PARACUDA##NAME##_upsweep(device_array, device_flags,
        device_iFlags, length, max_threads); \
    PARACUDA##NAME##_downsweep(device_sumArray,
        device_array, device_flags, device_iFlags,
        device_arrayIndex, length, max_threads);  \
    \
    PARACUDA##TYPE##_copy_device_host(output, device_array
        , length); \
    /*PARACUDA##TYPE##_copy_device_host(flags,
        device_iFlags, length);*/ \
    \
    PARACUDA##TYPE##_free_device(device_array); \
    PARACUDA_int_free_device(device_flags); \
    PARACUDA_int_free_device(device_iFlags);\
    if(sumArrayIndex) {\
        TYPE last;\
        TYPE valueToAdd;\
        PARACUDA##TYPE##_from_vector_host(&last, output,
            length - 1); \
        PARACUDA##TYPE##_from_vector_host(&valueToAdd,
            input, length -1);\
        OPERATOR(&last, &last, &valueToAdd);\
        PARACUDA##TYPE##_to_vector_host(device_sumArray, &
            last,sumArrayIndex[length -1]);\
        PARACUDA##TYPE##_copy_device_host(sumArray,
            device_sumArray, length); \
      }\
    PARACUDA##TYPE##_free_device(device_sumArray);\
    PARACUDA##TYPE##_free_device(device_arrayIndex);\
    cutilCheckMsg("Kernel execution failed");    \
    return output;\
} \
__host__ \
 PARACUDA##TYPE##_struct* NAME(PARACUDA##TYPE##_struct*
     output, PARACUDA##TYPE##_struct* input, int* flags,
     size_t length, size_t max_threads) \
{\
  return NAME(0, output, input, flags, 0, length,
      max_threads);          \
}\

#endif
```

B.14 nvidia_scan.h

```
#ifndef NVIDIA_SCAN
#define NVIDIA_SCAN

extern void preallocBlockSums(unsigned int maxNumElements);
extern void deallocBlockSums();
extern void prescanArray(int *outArray, int *inArray, int
    numElements);

void nvidia_scan(int* output, int* input, size_t length) {
```

```
    preallocBlockSums(length);
    prescanArray(output, input, length);
    deallocBlockSums();
}

#endif
```

B.15 nvidia_kernel.cu

```
/*
 * Copyright 1993-2006 NVIDIA Corporation.  All rights
     reserved.
 *
 * NOTICE TO USER:
 *
 * This source code is subject to NVIDIA ownership rights
     under U.S. and
 * international Copyright laws.
 *
 * NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF
     THIS SOURCE
 * CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT
     EXPRESS OR
 * IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL
     WARRANTIES WITH
 * REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED
     WARRANTIES OF
 * MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A
     PARTICULAR PURPOSE.
 * IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL,
     INDIRECT, INCIDENTAL,
 * OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER
     RESULTING FROM LOSS
 * OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF
     CONTRACT, NEGLIGENCE
 * OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
     CONNECTION WITH THE USE
 * OR PERFORMANCE OF THIS SOURCE CODE.
 *
 * U.S. Government End Users.  This source code is a "
     commercial item" as
 * that term is defined at 48 C.F.R. 2.101 (OCT 1995),
     consisting  of
 * "commercial computer software" and "commercial computer
     software
 * documentation" as such terms are used in 48 C.F.R.
     12.212 (SEPT 1995)
 * and is provided to the U.S. Government only as a
     commercial end item.
 * Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1
      through
 * 227.7202-4 (JUNE 1995), all U.S. Government End Users
     acquire the
 * source code with only those rights set forth herein.
 */
#ifndef _PRESCAN_CU_
#define _PRESCAN_CU_

// includes, kernels
#include <cutil_inline.h>
#include <nvidia_large_kernel.cu>
#include <assert.h>
```

```
inline bool
isPowerOfTwo(int n)
{
    return ((n&(n-1))==0) ;
}

inline int
floorPow2(int n)
{
#ifdef WIN32
    // method 2
    return 1 << (int)logb(n);
#else
    // method 1
    int exp;
    frexp(n, &exp);
    return 1 << (exp - 1);
#endif
}

#define BLOCK_SIZE 256

int** g_scanBlockSums;
unsigned int g_numEltsAllocated = 0;
unsigned int g_numLevelsAllocated = 0;

void preallocBlockSums(unsigned int maxNumElements)
{
    assert(g_numEltsAllocated == 0); // shouldn't be called

    g_numEltsAllocated = maxNumElements;

    unsigned int blockSize = BLOCK_SIZE; // max size of the
        thread blocks
    unsigned int numElts = maxNumElements;

    int level = 0;

    do
    {
        unsigned int numBlocks =
            max(1, (int)ceil(numElts / (2 * blockSize)));
        if (numBlocks > 1)
        {
            level++;
        }
        numElts = numBlocks;
    } while (numElts > 1);

    g_scanBlockSums = (int**) malloc(level * sizeof(int*));
    g_numLevelsAllocated = level;

    numElts = maxNumElements;
    level = 0;

    do
    {
        unsigned int numBlocks =
            max(1, (int)ceil(numElts / (2 * blockSize)));
        if (numBlocks > 1)
        {
```

```
            cutilSafeCall(cudaMalloc((void**) &
                g_scanBlockSums[level++],
                                     numBlocks * sizeof(
                                         int)));
        }
        numElts = numBlocks;
    } while (numElts > 1);

    cutilCheckMsg("preallocBlockSums");
}

void deallocBlockSums()
{
    for (unsigned int i = 0; i < g_numLevelsAllocated; i++)
    {
        cudaFree(g_scanBlockSums[i]);
    }

    cutilCheckMsg("deallocBlockSums");

    free((void**)g_scanBlockSums);

    g_scanBlockSums = 0;
    g_numEltsAllocated = 0;
    g_numLevelsAllocated = 0;
}


void prescanArrayRecursive(int *outArray,
                           const int *inArray,
                           int numElements,
                           int level)
{
    unsigned int blockSize = BLOCK_SIZE; // max size of the
        thread blocks
    unsigned int numBlocks =
        max(1, (int)ceil((int)numElements / (2.f *
            blockSize)));
    unsigned int numThreads;

    if (numBlocks > 1)
        numThreads = blockSize;
    else if (isPowerOfTwo(numElements))
        numThreads = numElements / 2;
    else
        numThreads = floorPow2(numElements);

    unsigned int numEltsPerBlock = numThreads * 2;

    // if this is a non-power-of-2 array, the last block
        will be non-full
    // compute the smallest power of 2 able to compute its
        scan.
    unsigned int numEltsLastBlock =
        numElements - (numBlocks-1) * numEltsPerBlock;
    unsigned int numThreadsLastBlock = max(1,
        numEltsLastBlock / 2);
    unsigned int np2LastBlock = 0;
    unsigned int sharedMemLastBlock = 0;

    if (numEltsLastBlock != numEltsPerBlock)
    {
```

```
            np2LastBlock = 1;

            if(!isPowerOfTwo(numEltsLastBlock))
                numThreadsLastBlock = floorPow2(
                    numEltsLastBlock);

            unsigned int extraSpace = (2 * numThreadsLastBlock)
                 / NUM_BANKS;
            sharedMemLastBlock =
                sizeof(int) * (2 * numThreadsLastBlock +
                    extraSpace);
        }

        // padding space is used to avoid shared memory bank
            conflicts
        unsigned int extraSpace = numEltsPerBlock / NUM_BANKS;
        unsigned int sharedMemSize =
            sizeof(int) * (numEltsPerBlock + extraSpace);

    #ifdef DEBUG
        if (numBlocks > 1)
        {
            assert(g_numEltsAllocated >= numElements);
        }
    #endif

        // setup execution parameters
        // if NP2, we process the last block separately
        dim3  grid(max(1, numBlocks - np2LastBlock), 1, 1);
        dim3  threads(numThreads, 1, 1);

        // make sure there are no CUDA errors before we start
        cutilCheckMsg("prescanArrayRecursive before kernels");

        // execute the scan
        if (numBlocks > 1)
        {
            prescan<true, false><<< grid, threads,
                sharedMemSize >>>(outArray,
                                                                    inArray
                                                                        ,
                                                                    g_scanBlockSums
                                                                        [
                                                                        level
                                                                        ],
                                                                    numThreads
                                                                        *
                                                                        2,
                                                                        0,
                                                                        0)
                                                                        ;

            cutilCheckMsg("prescanWithBlockSums");
            if (np2LastBlock)
            {
```

```
        prescan<true, true><<< 1, numThreadsLastBlock,
            sharedMemLastBlock >>>
            (outArray, inArray, g_scanBlockSums[level],
                 numEltsLastBlock,
             numBlocks - 1, numElements -
                 numEltsLastBlock);
        cutilCheckMsg("prescanNP2WithBlockSums");
    }

    // After scanning all the sub-blocks, we are mostly
        done.  But now we
    // need to take all of the last values of the sub-
        blocks and scan those.
    // This will give us a new value that must be sdded
        to each block to
    // get the final results.
    // recursive (CPU) call
    prescanArrayRecursive(g_scanBlockSums[level],
                          g_scanBlockSums[level],
                          numBlocks,
                          level+1);

    uniformAdd<<< grid, threads >>>(outArray,
                                    g_scanBlockSums[
                                        level],
                                    numElements -
                                        numEltsLastBlock
                                        ,
                                    0, 0);
    cutilCheckMsg("uniformAdd");
    if (np2LastBlock)
    {
        uniformAdd<<< 1, numThreadsLastBlock >>>(
            outArray,
                                                 g_scanBlockSums
                                                     [
                                                     level
                                                     ],
                                                 numEltsLastBlock
                                                     ,
                                                 numBlocks
                                                      -

                                                     1,
                                                 numElements
                                                      -

                                                     numEltsLastBlock
                                                     );
        cutilCheckMsg("uniformAdd");
    }
}
else if (isPowerOfTwo(numElements))
{
    prescan<false, false><<< grid, threads,
        sharedMemSize >>>(outArray, inArray,
                                                                 0,

                                                                      numThreads

                                                                      *
```

```
                                                                    2,
                                                                    0,
                                                                    0)
                                                                    ;

            cutilCheckMsg("prescan");
        }
        else
        {
             prescan<false, true><<< grid, threads,
                 sharedMemSize >>>(outArray, inArray,
                                                                0,
                                                                    numElements
                                                                    ,
                                                                    0,
                                                                    0)
                                                                    ;

            cutilCheckMsg("prescanNP2");
        }
}

void prescanArray(int *outArray, int *inArray, int
    numElements)
{
    prescanArrayRecursive(outArray, inArray, numElements,
        0);
}


#endif // _PRESCAN_CU_
```

B.16 nvidia_large_kernel.cu

```
/*
 * Copyright 1993-2006 NVIDIA Corporation.  All rights
     reserved.
 *
 * NOTICE TO USER:
 *
 * This source code is subject to NVIDIA ownership rights
     under U.S. and
 * international Copyright laws.
 *
 * NVIDIA MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF
     THIS SOURCE
 * CODE FOR ANY PURPOSE.  IT IS PROVIDED "AS IS" WITHOUT
     EXPRESS OR
 * IMPLIED WARRANTY OF ANY KIND.  NVIDIA DISCLAIMS ALL
     WARRANTIES WITH
 * REGARD TO THIS SOURCE CODE, INCLUDING ALL IMPLIED
     WARRANTIES OF
 * MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A
     PARTICULAR PURPOSE.
 * IN NO EVENT SHALL NVIDIA BE LIABLE FOR ANY SPECIAL,
     INDIRECT, INCIDENTAL,
```

```
 * OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER
     RESULTING FROM LOSS
 * OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF
     CONTRACT, NEGLIGENCE
 * OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
     CONNECTION WITH THE USE
 * OR PERFORMANCE OF THIS SOURCE CODE.
 *
 * U.S. Government End Users.  This source code is a "
     commercial item" as
 * that term is defined at 48 C.F.R. 2.101 (OCT 1995),
     consisting  of
 * "commercial computer software" and "commercial computer
     software
 * documentation" as such terms are used in 48 C.F.R.
     12.212 (SEPT 1995)
 * and is provided to the U.S. Government only as a
     commercial end item.
 * Consistent with 48 C.F.R.12.212 and 48 C.F.R. 227.7202-1
      through
 * 227.7202-4 (JUNE 1995), all U.S. Government End Users
     acquire the
 * source code with only those rights set forth herein.
 */

#ifndef _SCAN_BEST_KERNEL_CU_
#define _SCAN_BEST_KERNEL_CU_

// Define this to more rigorously avoid bank conflicts,
// even at the lower (root) levels of the tree
// Note that due to the higher addressing overhead,
    performance
// is lower with ZERO_BANK_CONFLICTS enabled.  It is
    provided
// as an example.
//#define ZERO_BANK_CONFLICTS

// 16 banks on G80
#define NUM_BANKS 16
#define LOG_NUM_BANKS 4

#ifdef ZERO_BANK_CONFLICTS
#define CONFLICT_FREE_OFFSET(index) ((index) >>
    LOG_NUM_BANKS + (index) >> (2*LOG_NUM_BANKS))
#else
#define CONFLICT_FREE_OFFSET(index) ((index) >>
    LOG_NUM_BANKS)
#endif

////////////////////////////////////////////////////////////////////////////////

// Work-efficient compute implementation of scan, one
    thread per 2 elements
// Work-efficient: O(log(n)) steps, and O(n) adds.
// Also shared storage efficient: Uses n + n/NUM_BANKS
    shared memory -- no ping-ponging
// Also avoids most bank conflicts using single-element
    offsets every NUM_BANKS elements.
//
// In addition, If ZERO_BANK_CONFLICTS is defined, uses
//     n + n/NUM_BANKS + n/(NUM_BANKS*NUM_BANKS)
```

```
// shared memory. If ZERO_BANK_CONFLICTS is defined, avoids
    ALL bank conflicts using
// single-element offsets every NUM_BANKS elements, plus
    additional single-element offsets
// after every NUM_BANKS^2 elements.
//
// Uses a balanced tree type algorithm.  See Blelloch, 1990
    "Prefix Sums
// and Their Applications", or Prins and Chatterjee PRAM
    course notes:
// http://www.cs.unc.edu/~prins/Classes/203/Handouts/pram.
    pdf
//
// This work-efficient version is based on the algorithm
    presented in Guy Blelloch's
// excellent paper "Prefix sums and their applications".
// http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/
    public/papers/CMU-CS-90-190.html
//
// Pro: Work Efficient, very few bank conflicts (or zero if
    ZERO_BANK_CONFLICTS is defined)
// Con: More instructions to compute bank-conflict-free
    shared memory addressing,
// and slightly more shared memory storage used.
//

template <bool isNP2>
__device__ void loadSharedChunkFromMem(int *s_data,
                                       const int *g_idata,
                                       int n, int baseIndex
                                           ,
                                       int& ai, int& bi,
                                       int& mem_ai, int&
                                           mem_bi,
                                       int& bankOffsetA,
                                           int& bankOffsetB
                                           )
{
    int thid = threadIdx.x;
    mem_ai = baseIndex + threadIdx.x;
    mem_bi = mem_ai + blockDim.x;

    ai = thid;
    bi = thid + blockDim.x;

    // compute spacing to avoid bank conflicts
    bankOffsetA = CONFLICT_FREE_OFFSET(ai);
    bankOffsetB = CONFLICT_FREE_OFFSET(bi);

    // Cache the computational window in shared memory
    // pad values beyond n with zeros
    s_data[ai + bankOffsetA] = g_idata[mem_ai];

    if (isNP2) // compile-time decision
    {
        s_data[bi + bankOffsetB] = (bi < n) ? g_idata[
            mem_bi] : 0;
    }
    else
    {
        s_data[bi + bankOffsetB] = g_idata[mem_bi];
    }
```

```
}

template <bool isNP2>
__device__ void storeSharedChunkToMem(int* g_odata,
                                      const int* s_data,
                                      int n,
                                      int ai, int bi,
                                      int mem_ai, int
                                          mem_bi,
                                      int bankOffsetA, int
                                          bankOffsetB)
{
    __syncthreads();

    // write results to global memory
    g_odata[mem_ai] = s_data[ai + bankOffsetA];
    if (isNP2) // compile-time decision
    {
        if (bi < n)
            g_odata[mem_bi] = s_data[bi + bankOffsetB];
    }
    else
    {
        g_odata[mem_bi] = s_data[bi + bankOffsetB];
    }
}

template <bool storeSum>
__device__ void clearLastElement(int* s_data,
                                 int *g_blockSums,
                                 int blockIndex)
{
    if (threadIdx.x == 0)
    {
        int index = (blockDim.x << 1) - 1;
        index += CONFLICT_FREE_OFFSET(index);

        if (storeSum) // compile-time decision
        {
            // write this block's total sum to the
                corresponding index in the blockSums array
            g_blockSums[blockIndex] = s_data[index];
        }

        // zero the last element in the scan so it will
            propagate back to the front
        s_data[index] = 0;
    }
}



__device__ unsigned int buildSum(int *s_data)
{
    unsigned int thid = threadIdx.x;
    unsigned int stride = 1;

    // build the sum in place up the tree
    for (int d = blockDim.x; d > 0; d >>= 1)
    {
        __syncthreads();
```

```
        if (thid < d)
        {
            int i  = __mul24(__mul24(2, stride), thid);
            int ai = i + stride - 1;
            int bi = ai + stride;

            ai += CONFLICT_FREE_OFFSET(ai);
            bi += CONFLICT_FREE_OFFSET(bi);

            s_data[bi] += s_data[ai];
        }

        stride *= 2;
    }

    return stride;
}

__device__ void scanRootToLeaves(int *s_data, unsigned int
    stride)
{
     unsigned int thid = threadIdx.x;

    // traverse down the tree building the scan in place
    for (int d = 1; d <= blockDim.x; d *= 2)
    {
        stride >>= 1;

        __syncthreads();

        if (thid < d)
        {
            int i  = __mul24(__mul24(2, stride), thid);
            int ai = i + stride - 1;
            int bi = ai + stride;

            ai += CONFLICT_FREE_OFFSET(ai);
            bi += CONFLICT_FREE_OFFSET(bi);

            int t  = s_data[ai];
            s_data[ai] = s_data[bi];
            s_data[bi] += t;
        }
    }
}

template <bool storeSum>
__device__ void prescanBlock(int *data, int blockIndex, int
     *blockSums)
{
    int stride = buildSum(data);               // build the
         sum in place up the tree
    clearLastElement<storeSum>(data, blockSums,
                               (blockIndex == 0) ? blockIdx
                                   .x : blockIndex);
    scanRootToLeaves(data, stride);            // traverse
        down tree to build the scan
}

template <bool storeSum, bool isNP2>
__global__ void prescan(int *g_odata,
                        const int *g_idata,
```

```
                        int *g_blockSums,
                        int n,
                        int blockIndex,
                        int baseIndex)
{
    int ai, bi, mem_ai, mem_bi, bankOffsetA, bankOffsetB;
    extern __shared__ int s_data[];

    // load data into shared memory
    loadSharedChunkFromMem<isNP2>(s_data, g_idata, n,
                                  (baseIndex == 0) ?
                                  __mul24(blockIdx.x, (
                                      blockDim.x << 1)):
                                      baseIndex,
                                  ai, bi, mem_ai, mem_bi,
                                  bankOffsetA, bankOffsetB)
                                      ;
    // scan the data in each block
    prescanBlock<storeSum>(s_data, blockIndex, g_blockSums)
        ;
    // write results to device memory
    storeSharedChunkToMem<isNP2>(g_odata, s_data, n,
                                 ai, bi, mem_ai, mem_bi,
                                 bankOffsetA, bankOffsetB);
}


__global__ void uniformAdd(int *g_data,
                           int *uniforms,
                           int n,
                           int blockOffset,
                           int baseIndex)
{
    __shared__ int uni;
    if (threadIdx.x == 0)
        uni = uniforms[blockIdx.x + blockOffset];

    unsigned int address = __mul24(blockIdx.x, (blockDim.x
        << 1)) + baseIndex + threadIdx.x;

    __syncthreads();

    // note two adds per thread
    g_data[address]              += uni;
    g_data[address + blockDim.x] += (threadIdx.x + blockDim
        .x < n) * uni;
}


#endif // #ifndef _SCAN_BEST_KERNEL_CU_
```

B.17 cpu_map.h

```
void cpu_map(int* array, int length)
{
  for(int i = 0; i < length; i++)
  {
    array[i] = array[i] + 1;
  }
}
```

B.18 cpu_scan.h

```
void cpu_scan(int* array, int length)
{
  for(int i = 2; i < length; i++)
  {
    array[i] = array[i - 2] + array[i - 1];
  }
  array[1] = array[0];
  array[0] = 0;
}
```

B.19 cpu_split.h

```
void cpu_split(int* array, int* flags, int* store, int
    length)
{
  int left = 0;
  int right = 0;
  int temp;
  for(int current = 0; current < length; current += 1)
  {
    if(flags[current])
    {
      store[right] = array[current];
      right += 1;
    }
    else
    {
      temp = array[current];
      array[current] = array[left];
      array[left] = temp;
      left += 1;
    }
  }
  for(int current = left; current < length; current += 1)
  {
    array[current] = store[current - left];
  }
}
```

B.20 cpu_radix.h

```
void cpu_radix(int* array, int* store, int length)
{
  for(int i = 0; i < 32; ++i)
  {
    int num = 1 << i;
    int left = 0;
    int right = 0;
    for(int current = 0; current < length; current += 1)
    {
      if(array[current] & num)
      {
        store[right] = array[current];
        right += 1;
      }
      else
      {
        int t = array[current];
        array[current] = array[left];
```

```
        array[left] = t;
        left += 1;
      }
    }
    for(int current = left; current < length; current += 1)
    {
      array[current] = store[current - left];
    }
  }
}
```

Litteratur

[Blelloch] Guy E. Blelloch, Vector Models for Data-Parallel Computing, 1990.

[Harris] M Harris, Parallel Prefix Sum (Scan) with CUDA, available at `http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/scan/doc/scan.pdf`, January 2008.

[Shubhabrata] Sengupta, Harris, Zhang and Owens, Scan primitives for GPU computing, available at `http://www.idav.ucdavis.edu/func/return_pdf?pub_id=915`, April 2001.

[Guide] NVIDIA, CUDA 2.1 Programming Guide, available at `http://www.nvidia.com/object/cuda_develop.html`, November 2008