

GPGPU

Allmänna beräkningar på grafikprocessorer

Kandidatarbete
Marcus Hellberg
Tekniska Fakulteten
Åbo Akademi
5 maj 2009

Sammanfattning

Denna uppsats diskuterar användningen av grafikprocessorer som strömprocessorer för att utföra allmänna beräkningar. Grafikprocessorer kan betraktas som massivt parallella processorer; moderna grafikprocessorer klarar av att exekvera flera hundra programtrådar parallellt. Tack vare sitt specifika syfte har grafikprocessorerna utvecklats i mycket snabbare takt än centralprocessorer och erbjuder idag flerfaldig beräkningskapacitet jämfört med centralprocessorer.

För att utföra beräkningar på grafikprocessorer krävs algoritmer som är speciellt utformade för att utnyttja den specifika arkitekturen som grafikprocessorer har. Uppsatsen går igenom grunderna i hur en grafikprocessor arbetar för att ge en bild över vilka delar av processorn som kan utnyttjas till allmänna beräkningar och hurdana problem som lämpar sig för beräkning på grafikprocessorer.

Texten tar upp grundläggande programmeringssätt och operationer inom GPGPU-programmering samt exempel på beräkning av snabba Fouriertransformationer med hjälp av grafikprocessorer.

Nyckelord: FFT, GPU, GPGPU, grafikprocessor, parallellprogrammering, strömprogrammering

Innehåll

Sammanfattning	i
1 Inledning	1
2 Grafikprocessorer	1
2.1 Utveckling	2
2.2 Grafikpipeline	3
2.3 Skillnader till centralprocessorer	7
2.4 Framtid	10
3 Programmering av grafikprocessorer	11
3.1 Strömprogrammering	11
3.2 Grundläggande operationer	13
3.3 Flödeskontroll	15
3.4 Programmeringsgränssnitt och bibliotek	16
3.5 Svårigheter med GPGPU-programmering	20
4 Snabba Fouriertransformen på grafikprocessorer	21
4.1 Matematisk bakgrund	22
4.2 Fastest Fourier Transform in the West	25
4.3 CUDA Fast Fourier Transform Library	26
4.4 Jämförelse av prestanda	26
5 Diskussion	29
Referenser	31
A Lista över använda begrepp	34

1 Inledning

GPGPU står för allmänna beräkningar på grafikprocessorer (eng. General-Purpose Computation on Graphics Processors)[OLG⁺07]. Med allmänna beräkningar anses sådana beräkningar som inte är traditionella grafikberäkningar.

GPGPU har blivit möjligt efter att den traditionella grafikprocessorn med fast funktionalitet har gett vika för en modell där grafikprocesseringen sköts av en stor mängd parallella processorer. Dessa processorer är tillräckligt flexibla för att kunna användas till att exekvera många beräkningsintensiva applikationer. Moderna grafikprocessorer stöder tillräckligt hög precision för att kunna användas även till mera krävande applikationer. Orsaken till att GPGPU har fått ett stort intresse inom vetenskapen är att de är mycket förmånliga i förhållande till deras beräkningskapacitet[Hou07].

För att förstå principerna i GPGPU-programmering är det viktigt att veta hur en grafikprocessor arbetar då den behandlar grafik. Först förklaras stegen i en grafikpipeline, därefter tas de väsentligaste delarna ur en GPGPU-programmerares synvinkel upp.

Programmeringsmodellen som används för att utföra GPGPU-beräkningar kallas för strömprogrammering. Uppsatsen tar upp grundläggande principer och operationer i strömprogrammering samt de viktigaste programmeringsgränssnitten som finns tillgängliga för allmän programmering av grafikprocessorer.

I sista delen av uppsatsen jämförs prestandan hos en implementation av snabba Fouriertransformen på en grafikprocessor mot en implementation för centralprocessorer för att åskådliggöra möjligheterna hos GPGPU-programmering.

2 Grafikprocessorer

Grafikprocessorer (GPU) är dedicerade mikroprocessorer som används för grafikrendering. Grafikprocessorer är till skillnad från centralprocessorer (CPU) specialprocessorer som är utvecklade för att snabbt utföra stora mängder beräkningsintensiva matematiska operationer som behövs för att köra tredimen-

sionella spel och applikationer.

Grafikprocessorer kan vara integrerade på moderkortet eller befinna sig på ett skilt tilläggskort. Skilda grafikkort har eget snabbt grafikminne med hög bandbredd på kortet. Integrerade grafikprocessorer använder sig ofta av systemets centralminne som arbetsminne med mindre bandbredd. I båda fallen är grafikprocessorerna skilda från centralprocessorn och accesseras via en databuss.

Termerna grafikprocessor och GPU används ofta för att beskriva hela grafikkortet med grafikminne. I denna uppsats används termen grafikprocessor endast för själva processorn, inte för tilläggskortet.

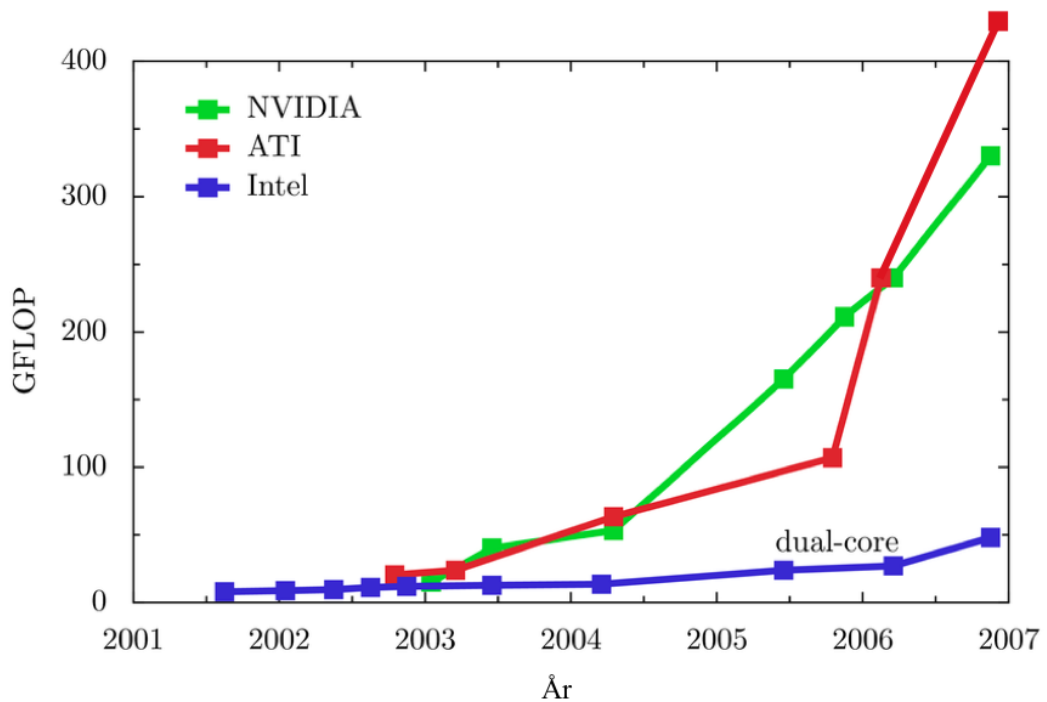
2.1 Utveckling

Eftersom grafikprocessering är en mycket beräkningsintensiv operation har man redan länge använt sig av skilda dedicerade grafikprocessorer för att avbelasta centralprocessorn. Ända till mitten av 1990-talet hanterade grafikprocessorer endast tvådimensionell grafik. Vid mitten av 1990-talet började tredimensionella (3D) datorspel bli populära. Dessa spel krävde mera beräkningskapacitet än vad en centralprocessor kunde erbjuda så behovet för 3D-accelerering var stort. De första 3D-grafikprocessorerna klarade endast av 3D-grafik och måste därför kopplas samman med en 2D-grafikprocessor för att kunna visa även tvådimensionell grafik[Wik08a].

Tillväxten av beräkningskapacitet hos grafikprocessorer är mycket snabb. Figur 2.1 visar hur beräkningskapaciteten hos grafikprocessorer och centralprocessorer av tre olika producenter har vuxit under en sex års period. Beräkningskapaciteten är angiven i GFLOP, vilket står för miljarder flyttalsoperationer. Centralprocessorer är optimerade för att köra sekventiella program snabbt och har därför mycket komplex logik för hoppförutsägning och cachehantering [hH05]. Denna komplexa logik har dock lett till att det är mycket arbetsamt och dyrt att höja på beräkningskapaciteten.

Grafikprocessorer använder sig av en specialiserad hårdvara som möjliggör massiva parallella beräkningar och enkel utökning av beräkningskapacitet. Grafikberäkning är av sin natur mycket enkelt att parallellisera eftersom man kan

beräkna varje bildpunkt skilt från andra bildpunkter[Buc07]. En mycket större andel av processorns yta kan ägnas åt aritmetisk-logiska enheter (ALU) på en grafikprocessor tack vare mindre behov av invecklad kontrolllogik. Tack vare denna specialiserade uppbyggnad går det enklare att lägga till flera transistorer för beräkning i grafikprocessorer än hos centralprocessorer[Owe07b].



Figur 2.1: Graf över utvecklingen i beräkningsförmåga av både grafikprocessorer och centralprocessorer.[Hou07]

2.2 Grafikpipeline

För att förstå hur man kan utnyttja grafikprocessorer till att utföra allmänna beräkningar är det viktigt att förstå för hur en grafikprocessor är uppbyggd och fungerar.

I grafikprocessering byggs bilden upp av små trianglar. Trianglarna kan enkelt användas som byggstenar för mer komplexa geometriska figurer. Trianglarna är även enkla att representera som datastrukturer och de möjliggör därför effektiva beräkningar. Hörnpunkterna av en triangel kallas vertex. Vertexen innehåller förutom hörnpunktens position även information om färg, textur och normaler

som behövs vid renderering av bilden[FQKYS04][Hed06]. Varje vertex processeras skilt för sig i pipelinen och sammansätts till trianglar vid behov [LH07].

2.2.1 Stegen i en grafikpipeline

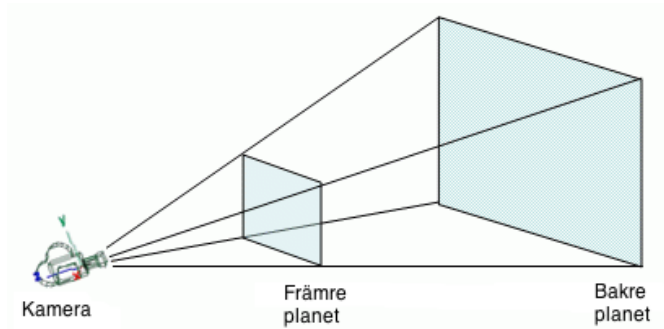
Grafikprocessorn får som indata en samling med tredimensionella objekt och information om deras position, texturer och ljussättning. Objekten är självständiga ännu i detta skede och har egna koordinatsystem i vilka de beskrivs. För att bygga upp en enhetlig bild transformeras alla objekt till samma koordinatsystem. Detta innebär att objekten blir skalade till rätt storlek, flyttas till korrekt plats och roteras vid behov [Hed06].

Genom att ha alla objekt i samma koordinatsystem kan man sedan enkelt med matris-vektor-multiplikeringar transformera objekten effektivt[LH07]. Dessa operationer är effektiva på grund av den parallella arkitekturen grafikprocessorer har.

Efter att alla objekt är transformerade till samma koordinatsystem räknar grafikprocessorn ut ljussättningen för scenen. Ljussättningen innebär en stor mängd beräkningar av skalära vektorprodukter[LH07]. Grafikprocessorn beräknar alla olika ljuskällors inverkan på alla ytor i scenen. En vanlig scen i ett datorspel kan innehålla tiotals olika ljuskällor och alla deras inverkan på ett objekt måste tas i beaktande; detta innebär att även detta skede av rendereringen kräver stor beräkningskapacitet.

Då scenen är ljussatt beräknar grafikprocessorn ut den del av en tredimensionella världen som kommer att vara synlig på skärmen. Figur 2.2 visar området som tas med i den slutliga bilden, allt som faller utanför området mellan främre och bakre planet förkastas eftersom de inte kommer att synas på skärmen[Fer08]. Genom att inte utföra onödiga beräkningar på objekt som inte syns kan grafikprocessorn uppnå högre effektivitet.

Följande steg i pipelinen är rasterisering. Rasterisering innebär att modellen av scenen diskretiseras till de bildpunkter som skärmen har. Varje enskild bildpunkt kan behandlas skilt härfter, vilket möjliggör parallell behandling av all data. Efter rasteriseringsskedet behandlas bilden som pixlar[LH07].



Figur 2.2: Simulering av kamera[Fer08].

Efter att bilden är rasteriserad, ökas verklighetstrogenheten hos bilden genom att lägga till olika texturer på ytor. Texturer hämtas ur grafikminnet och appliceras på varje pixel. I praktiken betyder detta att varje enskild tråd har läsaccess till grafikminnet. Eftersom närliggande pixlar ofta accesserar närliggande texturer kan texturerna effektivt cachas. Ofta appliceras ett flertal texturer per pixel[LH07].

Det sista som grafikprocessorn gör före den färdiga bilden ritas till skärmen är att den beräknar ifall en pixel blir bakom en redan ritad pixel. Ifall pixeln blir bakom en annan, slopas den för att spara beräkningstid. Detta skede av pipeline kallas även för Z-gallring eftersom den gallrar bort pixlar som blir täckta av andra på z-axeln[OLG⁺07].

2.2.2 Programmerbara delar av pipeline

Alla delar i grafikpipeline går inte att använda för allmän programmering på grund av sitt specifika användningssyfte. Delarna i pipeline som är viktigast för GPGPU-programmerare är vertexprocessorerna samt fragmentprocessorerna[Hed06]. Figur 2.3 visar med svärtade rutor de programmerbara delarna i en generell grafikpipeline.

Grafikpipeline utvecklas hela tiden mot en mera programmerbar modell. Från och med Nvidias NV20-seriens chipset år 2001 blev vertexprocessorer programmerbara, men fragmentprocessorerna hade ännu fast funktionalitet. Dessa processorer klarade dock inte ännu av förgreningar i program och var därför ännu inte användbara för de flesta beräkningar. Stöd för förgreningar kom först med NV40-

seriens chipset år 2004. Därmed blev allmänna beräkningar på grafikprocessorer möjliga i mycket bredare utsträckning[Wik08b].

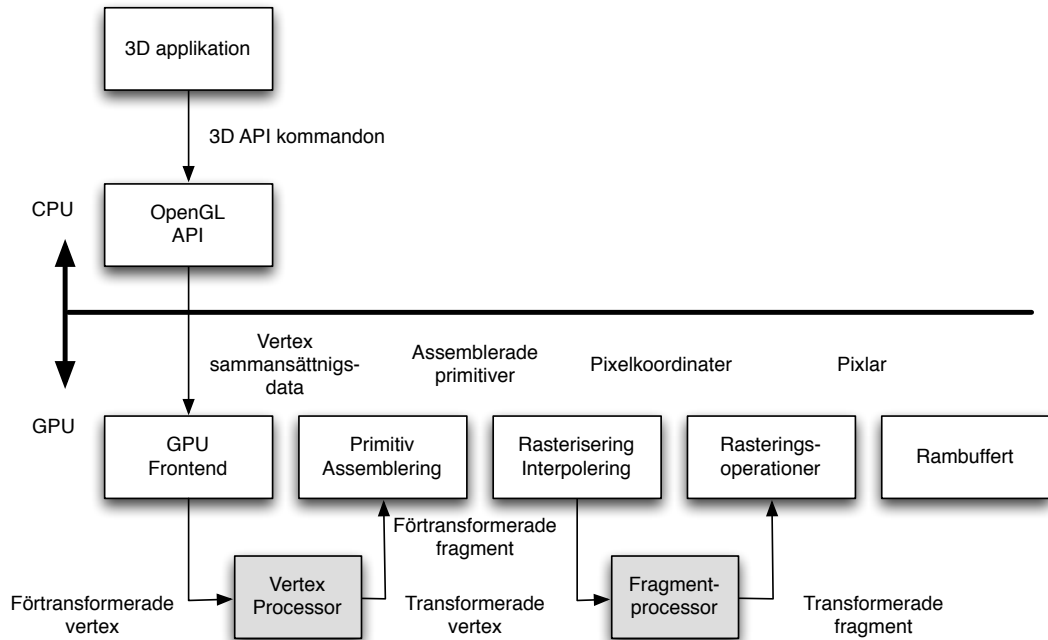
Vertexprocessorer är baserade på en Multiple Instruction stream, Multiple Data stream (MIMD)-struktur. I en MIMD-arkitektur används en mängd självständiga processorer som kan utföra olika instruktioner på olika data parallellt. Eftersom vertexprocessorer behandlar en stor mängd 4×4 transformationsmatriser har varje beräkningsenhet i vertexprocessorn en enhet för snabb beräkning av matriser. Vertexprocessorer läser in sin data från grafikminnet, centralprocessorn är ansvarig för att förflytta datan från huvudminnet till grafikminnet[IIH06].

Fragmentprocessorer är baserade på en Single Instruction, Multiple Data (SIMD)-struktur. För att snabbt kunna texturisera fragment är SIMD-arkitekturen ideal eftersom den möjliggör körningen av samma beräkningsoperation på en stor mängd data parallellt. Fragmentprocessorer har bra stöd för 4-komponenters vektorer eftersom de handskas med färger i RGBA-komponenter (Röd, Grön, Blå, Alfa-kanal)[IIH06].

På grund av sin SIMD-arkitektur och högre prestanda används oftast endast fragmentprocessorerna till GPGPU. Vertexprocessorerna används i detta fall endast till att begränsa beräkningsområdet i indatan[IIH06]. Detta leder till att beräkningskapaciteten i vertexprocessorerna i stort sett förblir outnyttjad. Ikeda, Ino och Hagihara utforskade möjligheten att flytta instruktioner från fragmentprocessorn till vertexprocessorn för att kunna effektivare utnyttja hårdvaran[IIH06]. Utnyttjandet av vertexprocessorerna snabbade märkbart upp flera algoritmer, men på grund av vertexprocessorernas arkitektur kan endast vissa typer av beräkningar flyttas över till dem.

Figur 2.4 visar hur en Nvidia 8800 GTX-grafikprocessors pipeline är uppbyggd. Jämfört med pipelinen i figur 2.3 ser man att Nvidia har övergått mot en uppbyggnad med ett större antal generellt programmerbara processorer i stället för specialiserade enheter. Fördelen med detta är att grafikprocessorn kan dynamiskt ändra andelen beräkningsenheter för olika jobb, och därmed undvika situationer där en del av processorn är överbelastad medan andra delar är helt outnyttjade. Ett större antal allmänna kärnor leder även till att programmere lättare kan utnyttja beräkningskapaciteten i grafikprocessorn för allmänna

beräkningar[Hou07].

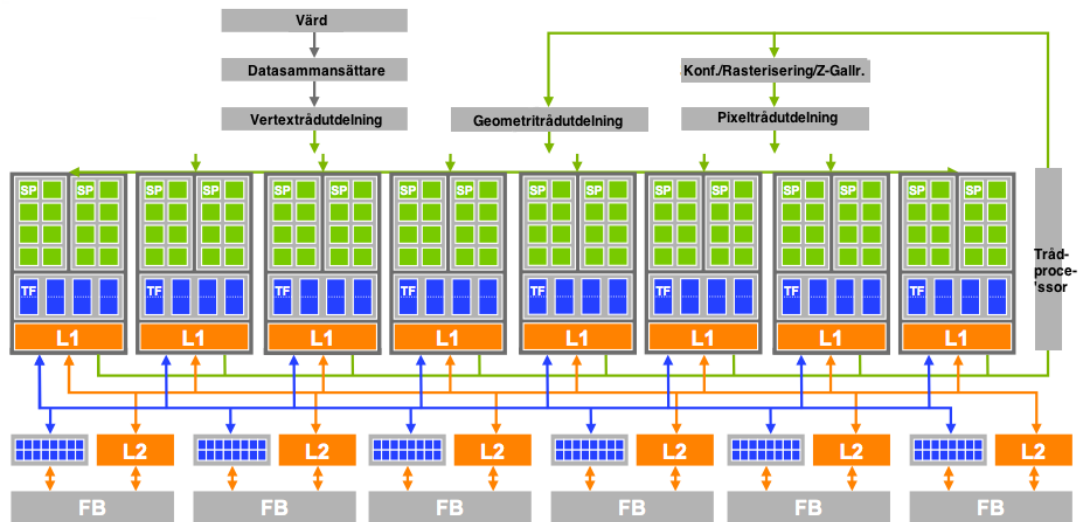


Figur 2.3: Schema över grafikpipeline[Wat05].

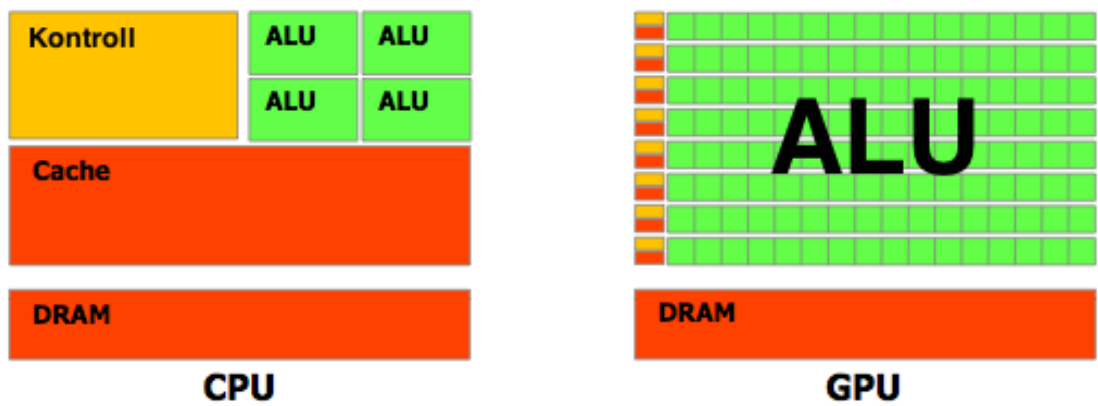
2.3 Skillnader till centralprocessorer

Att förstå skillnaderna mellan hur grafikprocessorer och centralprocessorer fungerar och är uppbyggda är nödvändigt för att förstå programmeringssätten för dessa två.

En grundläggande skillnad i hur centralprocessorer och grafikprocessorer fungerar är att centralprocessorer är optimerade för att utföra enskilda jobb med snabb svarstid. Grafikprocessorer är i sin tur optimerade att processera stora mängder data, även om det innebär längre svarstid[LH07]. Detta beror på att bilden på skärmen behöver uppdateras med några millisekunders mellanrum för att människoögat skall uppfatta bilden som rörlig. En latens på några nanosekunder kan bra tolereras i grafikprocessering. Denna filosofi har lett till att flera transistorer på grafikprocessorn kan tillägnas åt beräkningsenheter i stället för cache-minnen och kontrollhårdvara[Owe07b]. Figur 2.5 visar grafiskt skillnaden i ytarealen som ägnas åt ALU på en centralprocessor och en grafikprocessor.



Figur 2.4: Diagram över grafikipelinen för en Nvidia 8800 GTX GPU [Hou07].



Figur 2.5: Blockschema som illustrerar principkillnaderna i uppbyggnaden av en CPU och en GPU [NVI07a].

2.3.1 Trådar

Grafikprocessorer skiljer sig mycket även från flerkärnade centralprocessorer. Centralprocessorer använder sig av tunga trådar [Owe07b] som har bra prestanda per tråd och är avsedda för att köra sekventiella program. Även om centralprocessorer får allt flera kärnor håller de sig till dessa tunga trådar för att kunna erbjuda bra prestanda åt program som inte är flertrådade. Dessa trådar är dock tunga att skapa och att handskas med samtidigt som de har stor overhead.

I grafikprocessorer har man valt att använda sig av lätta trådar som är mindre resurskrävande att skapa och hantera. Prestandan hos en enskild tråd är sämre än hos tunga trådar och därför måste program som körs på grafikprocessorer använda sig av tusentals trådar för att utnyttja all beräkningskapacitet[Buc07].

2.3.2 Minne och latens

Grafikprocessorer har i de flesta fall eget dedicerat minne med hög bandbredd till sitt förfogande. Att minnet är dedicerat innebär dock att centralprocessorn är tvungen att flytta över datan till grafikkortets minne före den kan utföra operationer på det. AMD och Nvidia-grafikprocessorer med stöd för Close to Metal(CTM) eller Compute Unified Device Architecture (CUDA) har möjlighet att binda värddatorns minne och arbeta direkt på det, men detta innebär stora tidsförluster vid dataöverföring. För att kunna arbeta effektivt på centralminnet krävs mycket stora mängder data att beräkna[Hou07].

Grafikprocessorn är kopplad till centralprocessorn via en relativt långsam databuss, vilket innebär att beräkningar på grafikprocessorn alltid innebär overhead för överföring av datan. Följden av detta är att man kräver relativt stora datamängder för att kunna fullständigt utnyttja grafikprocessorns beräkningsförmåga[Hou07].

Grafikprocessorn har vanligen betydligt högre minnesbandbredd till sitt förfogande än centralprocessorn har till centralminnet[FQKYS04]. Grafikprocessorer kan även effektivt gömma mycket stor latens om den har tillräckligt mycket data att beräkna. Ifall en tråd stannar upp vid minnesaccess kan en annan tråd köras i dess ställe tills minnesaccessen är slutförd. På detta sätt utnytt-

jas grafikprocessorn hela tiden. Program på centralprocessorer är mycket mera utsatta för problem med minneslatens och kan därför inte alltid utnyttja sin beräkningskapacitet fullständigt[Hou07].

2.3.3 Precision

Traditionellt har grafikprocessorer hållit sig till 8 bitars precision för att förenkla beräkningarna och därmed höja prestandan. Krav från moderna spel och GPGPU-programmering har dock gjort att nya grafikprocessorer klarar av 32 bitars flyttal och heltal. Flyttalen uppfyller nästan kraven för IEEE 754-precision[NVI07a], förutom vissa avrundningsoperationer och undantagshantering. I de flesta fallen är det ändå möjligt att överföra SIMD kod skriven för centralprocessorer till grafikprocessorer utan förlust av precision[Hou07].

2.4 Framtid

Grafikprocessorers beräkningskapacitet ökar ungefär $1,7\times$ (pixel/sekund) till $2,4\times$ (vertex/sekund) årligen[OLG⁺07]. Centralprocessorer i sin tur har ungefärligt följt Moores lag¹ som hävdar att prestandan hos processorerna fördubblas med 18 månaders intervall, en årlig tillväxt på ca. $1,4\times$ [OLG⁺07]. Centralprocessorer har uppnått en gräns på klockhastighet och måste därför även börja använda sig av flera processorkärnor för att kunna fortsätta höja på sin prestanda. Effekten av övergången till parallellprocessering på centralprocessorer minskas av att program skrivna för enkärniga processorer inte automatiskt kan utnyttja den utökade beräkningskapaciteten. Grafikprocessorer i sin tur har från början bestått av parallella processorkärnor; program skrivna för dem går att skala till ett stort antal kärnor. Den snabba tillväxten i beräkningskapacitet hos grafikprocessorer jämfört med centralprocessorer förväntas fortsätta ännu i minst några hårdvarugenerationer[Hou07].

¹Teorin publicerades första gången år 1965 av Gordon E. Moore som var en av Intels grundare. Ursprungligen förutspådde Moore en årlig fördubbling av transistorer i en processor[Moo65], men var senare tvungen att ändra fördubblingsintervallet till två år. En av Moores kollegor räknade ut att prestandaökningen per transistor sammankopplat med ökningen av transistorantalet enligt Moore's lag kommer att leda till en fördubbling i prestanda med 18 månaders intervall, vilket även används ofta då Moore's lag nämns.

Allmän programmering av grafikprocessorer håller på att bli allt bättre stödd av grafikhårdvarutillverkare. Både AMD och Nvidia har gett ut GPGPU-programmeringsverktyg och infört ändringar i hårdvaran för att bättre stöda GPGPU-programmering, vilket behandlas mera detaljerat i nästa stycke.

3 Programmering av grafikprocessorer

Programmering av grafikprocessorer i allmänt syfte innebär en hel del svårigheter. För att kunna utnyttja tiotals processorkärnor och specialhårdvara krävs programmeringsmetoder som skiljer sig avsevärt från traditionell sekventiell programmering på centralprocessorer. På grund av sin specialiserade hårdvara är programmeringen även begränsad till att använda vissa speciella datastrukturer och operationer. Grafikprocessorer kan betraktas som strömprocessorer på grund av deras stora mängd parallella beräkningsenheter. Detta innebär att en strömprogrammeringsmodell lämpar sig mycket väl för programmering av grafikprocessorer.

Grafikbibliotek var länge det enda sättet att programmera grafikprocessorer. Även om detta fungerade till en viss grad var det mycket begränsande och krävde kännedom av grafikprogrammering. Grafikbibliotek stöder inte heller flera funktioner som är nödvändiga för att kunna programmera allmänna program.

Både Nvidia och AMD har gett ut utvecklingsverktyg för att underlätta programmerandet av grafikprocessorer till allmänt syfte utan att behöva gå igenom grafikprogrameringsgränssnitt[NVI07a][Adv06]. Dessa verktyg har betydligt underlättat GPGPU-programmering och möjliggjort det för personer som inte känner till grunderna i grafikprogrammering[Hou07].

3.1 Strömprogrammering

Strömprogrammering är en programmeringsparadigm som effektivt stöder behandling av stora datamängder på en mängd parallella processorer. Strömprogrammering skiljer sig från flertrådsprogrammering på centralprocessorer genom att programmeraren inte behöver själv skapa och hantera trådar.

I strömprogrammering körs samma program, en kernel, över varje element i en dataström. Strömprogrammering är nära besläktad med SIMD. Både strömprogrammering och SIMD bygger på grundtanken att utföra samma operation på ett flertal dataelement parallellt. De skiljer sig i att SIMD-implementationer såsom SSE (Streaming SIMD Extension) i centralprocessorer endast kan utföra operationen med några dataelement i taget, medan strömprocessorer gör det med hundratals dataelement åt gången[Wik08b].

```
(a)
for(int j = 1; j < height; ++j)
{
    for(int i = 0; i < width; ++i)
    {
        // get velocity at this cell
        Vec2f = grid (x, y);

        //trace backwards along velocity field
        float x = (i - (v.x * timestamp / dx));
        float y = (j - (v.y * timestamp / dy));

        grid(x, y) = grid.bilerp (x, y);
    }
}

(b)
void advect (float2      uv      :WPOS,
            out float4   xNew    :COLOR,

            uniform float dt, //timestamp
            uniform float dx, //grid scale
            uniform samplerRECT u, //velocity
            uniform samplerRECT x) //state
{
    //trace backwards along velocity field
    float2 pos = ub - dt * f2texRECT(u, uv) / dx;

    xNew = f4texRECTbilerp(x, pos);
}
```

Figur 3.1: Exempel på beräkning av en funktion med a) sekventiell programmering i C++ och b) strömprogrammering i Cg[Hou07].

Figur 3.1 visar hur ett problem löses i vanlig sekventiell programmering och

i strömprogrammering. Den märkbara skillnaden är att de två slingorna som används i den sekventiella versionen för att gå igenom alla element kan slopas i strömprogrammering. Koden i figur 3.1 b) är kerneln som körs på alla elementen i strömmen. Eftersom programlogiken är inkapslad i en kernel som är samma för alla element kan strömprogram enkelt utnyttja ett godtyckligt antal parallella processorer för beräkning.

En annan fördel med strömprogrammeringsmodellen är att den uppmanar programmeraren att programmera enligt “samla, beräkna och sprid ut”-modellen[GR05]. Detta innebär att datan laddas in i en ström från minnet. Sedan utförs alla operationer på datan och till sist skrivs, eller sprids, datan ut till minnet. Genom att göra så här skiljs beräkningarna från minnesaccesserna och på det sättet kan problem med latens vid minnesläsningar undvikas. Begränsningen i strömprogrammering blir minnesbandbredden i stället för minneshastigheten. Eftersom datan blir läst i början av programmet kan kompilatorn optimera minnesaccesserna väl.

För att ett program skall effektivt kunna behandlas med strömprocessorer måste det ha hög beräkningsintensitet. Detta innebär att programmet skall utföra flera beräkningar på ett dataelement hämtat ur minnet för att kunna kompensera för tiden som går åt till att läsa från minnet[Hou07]. Datat skall helst vara oberoende av varandra så att alla dataelement kan processeras parallellt och utan att vänta på svar från tidigare beräkningar[Wik08b].

3.2 Grundläggande operationer

Map-operationen används för att utföra samma beräkning på alla element i en ström. Denna operation är den viktigaste operationen i GPGPU-programmering eftersom alla program och operationer bygger på den[OLG⁺07].

De mest grundläggande operationerna i strömprogrammering är spridning och samling. Spridningsoperationen används då man skriver data i minnet och samlingsoperationen då data läses ur minnet in i en variabel[Owe07a]. Figur 3.2 visar exempel på dessa operationer i pseudokod.

Traditionellt har grafikprocessorer endast stött samlingsoperationen, eftersom

```
a) data[i] = x;  
b) x = data[i];
```

Figur 3.2: Exempel på a) spridning, b) samling i pseudokod.

spridningsoperationen inte är nödvändig i grafikprocessering och skrivning till minnet leder till risk för kollisioner. Grafikprocessorn kunde endast skriva beräkningsresultatet till en förutbestämd minnesadress då beräkningen var slutförd. Den mycket begränsade minnesaccessen har varit ett hinder för GPGPU-programmerare och har inneburit grova begränsningar för hurdana algoritmer som kunde användas på grafikprocessorer.

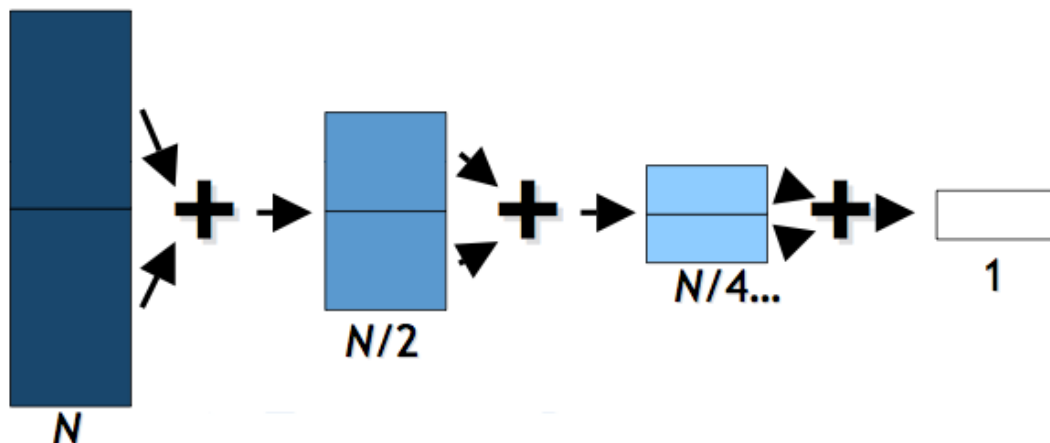
I nya grafikprocessorer tillåts skrivning till slumpmässiga minnesadresser, men ingen kollisionshantering eller kollisionsdetektering finns. Skrivning till minnet är långsammare än läsning på grund av grafikprocessorns arkitektur som tidigare har satt mera vikt på att snabbt läsa texturer ur minnet än skrivning till det [Owe07a].

Reduceringsoperationen är en operation som tar en dataström som indata och ger ut en mindre dataström. Reduceringsoperationen kan användas t.ex. till att beräkna summan av en mängd element i en ström. Reduceringsoperationen fungerar endast på associativa operatorer eftersom ingen garanti om beräkningsordningen finns [Owe07a]. Reduceringsoperationen fungerar genom att läsa in värden från motsvarande ställen från båda halvorna av dataströmmen och kombinera dem till en ström av halva längden. Genom att upprepa denna procedur fås resultatet till slut. Operationen illustreras i figur 3.3. Reduceringsoperationen kräver $O(\log n)$ steg och $O(n)$ arbete [OLG⁺07].

Prefix-summaoperationen räknar ut summan på alla föregående element i en serie. Operationen används mycket bl.a. för sorteringsalgoritmer. Operationen kräver $O(\log n)$ steg och $O(n \log n)$ additioner, vilket är $\log n$ mera än för en sekventiell implementation av algoritmen. [OLG⁺07].

Sökoperationen används för att söka ett specifikt element eller dess närliggande element ur en dataström. Sökningen är en sekventiell operation, men flera sökningar kan köras parallellt på grafikprocessorn [Owe07a].

Sorteringsoperationen används för att sortera elementen i en dataström enligt givna kriterier. Det finns flera olika implementationer för sökoperationer på gra-



Figur 3.3: Illustration av reduceringsoperationen [Owe07a].

fikprocessorer, men de flesta utnyttjar sorteringsnätverk. Dessa nätverk sorterar datan i ett bestämt antal steg oavsett datamängden. Noderna i nätverket har en förutbestämd kommunikationsmodell för att kunna undvika förgreningar i beräkningarna. Sökoperationer med sorteringsnätverk har en komplexitet på $O(n \log n)$ [OLG⁺07].

Filtreringsoperationen används för att filtrera ut element ur strömmen så endast önskade element kvarblir i strömmen. Mängden element och deras positioner är inte kända på förhand. Strömfiltrering implementeras ofta som en kombination av prefix-summa och sökoperationerna och har en körtid på $O(\log n)$ [OLG⁺07].

3.3 Flödeskontroll

Flödeskontroll i GPGPU-program har traditionellt varit mycket begränsat. Senaste hårdvarugenerationerna har medfört möjlighet till flödeskontroll. På grund av att förgreningar orsakar stor prestandaförlust bör de undvikas då det är möjligt. För att hantera förgreningar i programkod finns flera olika metoder.

För att hantera en villkorssats behöver en grafikprocessor tre pass. Vid första passet avgörs de två alternativa exekveringsstigarna och lagras i en matris. Sedan körs de två passen, en för varje exekveringsstig, som bestämdes i första passet. På detta sätt kan sedan datan ur den korrekta exekveringsstigen tas till vara [Hed06].

Det finns flera metoder som kan användas för att minska kostnaden på flödeskontroll. Metoderna innebär att villkoren evalueras i ett tidigare skede av pipelinen för att minska antalet onödiga beräkningar. Statisk bestämning av förgreningar är en metod som ofta används både på centralprocessorer och grafikprocessorer för att öka prestandan. Metoden går ut på att separera inre slingor till skilda strömmar för de olika exekveringsstigarna. På så sätt kan beräkningarna göras utan att förgreningar uppstår i exekveringen[OLG⁺07].

Ifall villkorssatser är beroende på tid eller antal iterationer, kan de evalueras endast då de förändras. Då man inte behöver kontrollera villkoret vid varje iteration, sparas tid. Z-gallring som beskrevs i stycke 2.2.1 är även ett mycket användbart verktyg för flödeskontroll. Genom att ändra på “djupet” av en datapunkt i strömmen kommer den att antingen förkastas (ifall den blir bakom annan data), eller visas (om den är främst). Detta möjliggör beräkning av olika värden på olika datapunkter trots att samma kernel körs på dem[OLG⁺07].

3.4 Programmeringsgränssnitt och bibliotek

Det finns flera olika sätt att programmera allmänna program på grafikprocessorer. De tidigaste programmen använde sig exklusivt av grafikprogrammeringsgränssnitt. Allt efter att GPGPU har blivit mera stött av hårdvarutillverkare, har bättre och mera flexibla verktyg kommit till. Dessa verktyg är dock ofta bundna till en viss tillverkares hårdvara.

3.4.1 Grafikprogrammeringsgränssnitt

Förr skötte GPGPU-programmering enbart genom grafikprogrammeringsgränssnitt såsom Microsofts DirectX eller Open Graphics Library (OpenGL). För programmering av vertex- och fragmentprocessorerna användes högnivå shaderspråk såsom Microsofts High Level Shading Language (HLSL) eller OpenGL Shading Language (GLSL). Nvidia har även ett shaderspråk, Cg, som kan användas för att skriva shaderprogram till DirectX och OpenGL[hH05].

Största svårigheten i att programmera allmänna algoritmer med dessa verktyg är att de inte är konstruerade för annat än grafikprogrammering. Detta in-

nebär att programmerare är tvungna att använda datastrukturer och funktioner avsedda för grafikprogram. Prestandan på program skrivna med grafikbibliotek är också sämre än program skrivna med speciella GPGPU-bibliotek eftersom GPGPU-bibliotek är optimerade för icke-grafiska beräkningar. För att veta vilka datastrukturer och funktioner som lämpar sig för allmän programmering, krävdes en god kunskap av grafikprogrammering[Hou07].

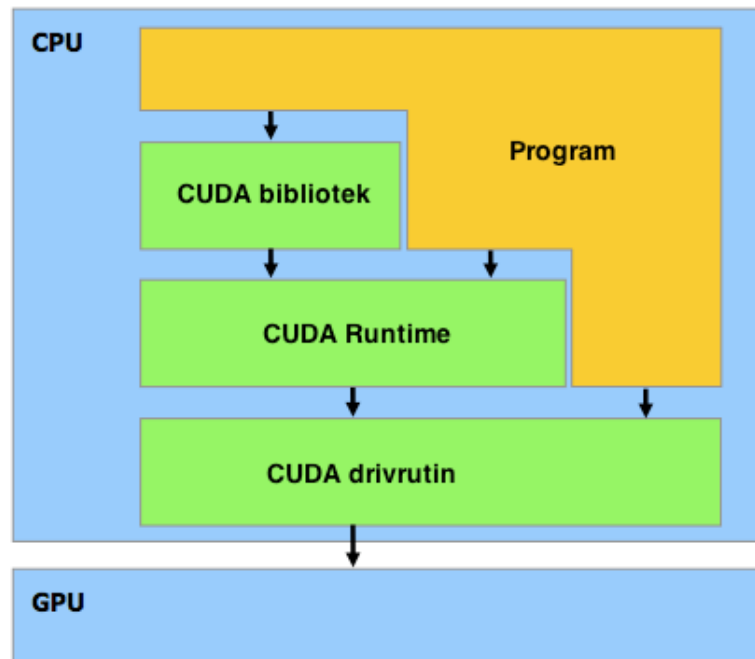
För att representera dataströmmar används texturer. Texturerna kan vara en, två eller tre dimensioner[Hed06]. För att utföra beräkningen på grafikprocessorn, ritas en geometri, vanligen en stor kvadrat[LHG⁺06]. Rasteriseringsenheten delar upp geometrin i små delar som kan betraktas som program. Fragmentprocessorn kör sedan shaderprogram på alla texturer och lagrar svaret i grafikminnet. Svaret kan läsas ur den färdiga bilden där en pixels färg motsvarar beräkningsresultatet[Buc07].

Andra problem med grafikprogrammeringsgränssnitten är att texturer är begränsade till sin storlek och att shaders inte kan skriva ut data till godtyckliga minnesadresser. Även kommunikation mellan trådar är mycket begränsad i grafikbibliotek, vilket begränsar typen av algoritmer som kan implementeras[Buc07].

3.4.2 Compute Unified Device Architecture

Compute Unified Device Architecture (CUDA) är en kombinerad hårdvaru- och mjukvaruarkitektur för GPGPU-programmering utvecklad av Nvidia. Arkitekturen stöds endast av de nyaste Nvidia-chipseten. CUDA består av tre lager: en hårdvarudrivrutin, ett programmeringsgränssnitt med runtime och två högnivåbibliotek för matematiska funktioner, CUFFT och CUBLAS[NVI07a]. Uppbyggnaden illustreras i figur 3.4. CUFFT är ett bibliotek för snabba Fouriertransformen (FFT), CUBLAS är ett bibliotek för linjära algebraiska funktioner. CUFFT beskrivs i mera detalj i stycke 4.3 samt jämförs med en CPU-baserad FFT implementation i stycke 4.4.

CUDA använder standardenlig C-programkod med vissa utökningar. CUDAs programmeringsmodell är inte en strömprogrammeringsmodell eftersom trådarna i programmet kan kommunicera med varann och skrivningar till minnet kan göras mitt i exekveringen, inte bara i slutet. Programmering med CUDA är mera ge-



Figur 3.4: Uppbyggnaden av Compute Unified Device Architecture[NVI07a].

nerell parallellprogrammering[Buc07].

CUDA ger programmerare möjlighet att direkt arbeta med centralminnet, med stöd för slumpmässigt adresserad skrivning och läsning. CUDA tillägger även stöd för pekare och ospecificerade datatyper; program är inte begränsade till att använda texturer som datatyper.

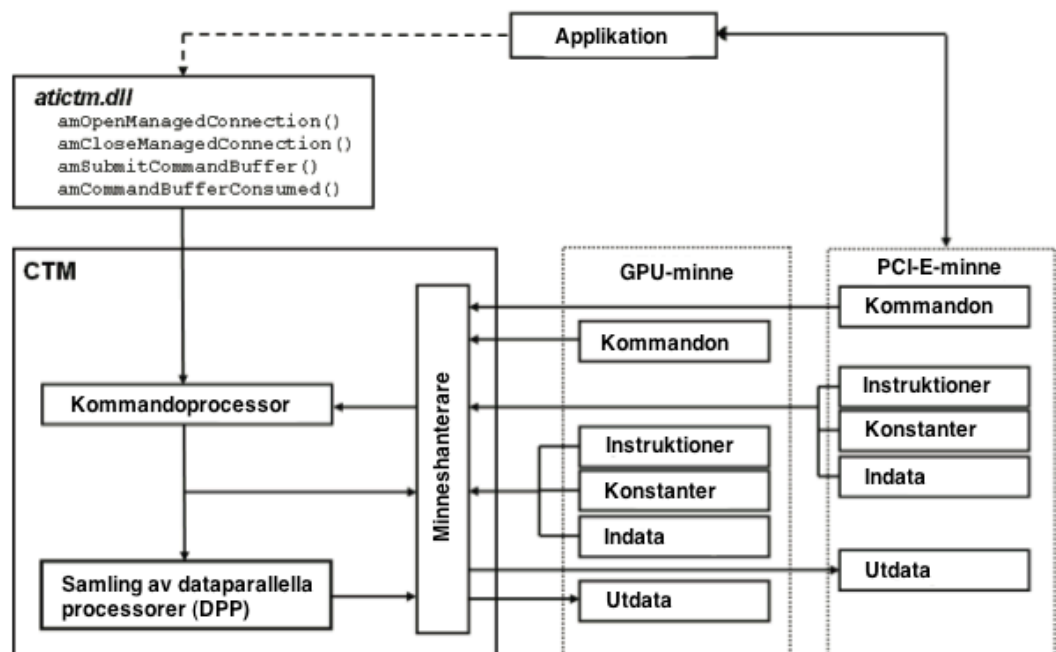
CUDA har ett parallellt datacache som trådar kan använda för att kommunicera med varann och för att minska beroendet på bandbredden till centralminnet[Buc07]. Datacachen är ett mycket snabbt minne, med accesstider jämförbara med register. Datacachen sköts av programmeraren.

3.4.3 Close to Metal

Close to Metal (CTM) är ett hårdvarugränssnitt utvecklat av AMD. Dess ändamål är att ge access till samlingen flyttalsprocessorer på AMD grafikprocessorer. CTM-arkitekturen är i liknelse till CUDA-arkitekturen en sammanbindning av både hårdvara och mjukvara[Adv06].

De fyra huvudsakliga delarna i CTM är samlingen av dataparallella processorer (Data Parallell Processor Array, DPP), processorexekveringsenheten (Processor Execution Unit, PE), villkorsenheten (Conditional Operation Unit, CO) och minneshanteraren (Memory Controller Unit, MC)[Adv06].

Processorexekveringsenheten distribuerar arbetsbördan till processorerna i DPP och delegerar kommandon åt andra enheter inom CTM. Villkorsenheten bestämmer om en processor skall användas i beräkningen och vart exekverande processorer skall skriva sina beräkningsresultat. Minneshanteraren hanterar alla minnesaccesser för de andra delarna i CTM.



Figur 3.5: Blockshema över Close To Metal-arkitekturen[Adv06].

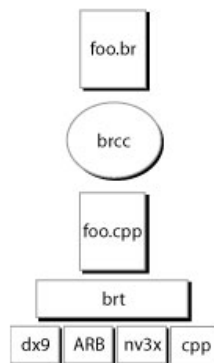
3.4.4 BrookGPU

Brook är ett strömprogrammeringsspråk som är utvecklat av Stanford University. BrookGPU är en anpassning av språket till grafikprocessorer samt behövliga bibliotek. BrookGPU är inte bunden till en viss hårdvaruleverantör; program skrivna i Brook går att köra på både Nvidia och AMD-grafikprocessorer. Brook GPU kan använda DirectX, OpenGL eller CTM som sin backend, beroende på

programmeringsomgivningen[BFH⁺].

BrookGPU-arkitekturen består av två komponenter: BrookGPU-kompilatorn (BRCC) samt Brook RunTime Library (BRT)[BFH⁺]. BRCC är en metakompilator som ändrar om Brook källkod till C++-källkod genom att konvertera Brook-datatyper till datatyper i C++. BRT-biblioteket innehåller implementationer av Brook-datatyperna för all den hårdvara BrookGPU stöder. Med hjälp av BRT kan BRCC kompilera optimerad kod för många olika arkitekturer. Uppbyggnaden av BrookGPU illustreras i figur 3.6

BrookGPU är licenserat under Berkeley Software Distribution (BSD)-licens, med delar under GNU General Public License (GPL)-licens. Detta innebär att källkoden för projektet är öppet för granskning åt alla intresserade. Detta innebär att BrookGPU lämpar sig mycket väl till akademiska och personliga projekt som inte vill binda sig till en viss hårdvarutillverkare. AMD har stött utvecklingen av BrookGPU och har get ut en utvidgning till språket, Brook+, som bättre utnyttjar AMD-hårdvara.



Figur 3.6: Blockschema över BrookGPU-arkitekturen[BFH⁺].

3.5 Svårigheter med GPGPU-programmering

Det finns flera svårigheter vid programmering av grafikprocessorer. Grafikprocessorer utvecklas i snabb takt, vilket innebär att även grundläggande funktionalitet kan ändras i ny hårdvara. Samma slags bakåtkompatibilitet som hos centralprocessorer finns inte, vilket innebär att program skrivna för grafikprocessorer är mer

arbetskrävande att uppehålla än program skrivna för centralprocessorer[Hou07]. Det är inte heller möjligt att porta kod skriven för centralprocessorer till grafikprocessorer på grund av de skiljande programmeringsmodellerna. Verktyg såsom CTM och CUDA underlättar protningsarbetet, men de algoritmer som används måste ändå skrivas om för att kunna användas på grafikprocessorer. Det finns inget sätt att automatiskt kompilera ett sekventiellt program att använda grafikprocessorer.

Ett stort problem för GPGPU-programmerare är att den underliggande strukturen i grafikprocessorer är ofta hemlig. Till skillnad från centralprocessorer där programmerare känner till vilka register och resurser finns till förfogande, är GPGPU-programmerare tvungna att hålla sig till de givna gränssnitten eller att själv försöka lista ut vilka resurser som finns[Hou07].

Grafikprocessorer har inte stöd för bitvisa operationer på data eller skiftningsoperationer[OLG⁺07]. I regel är centralprocessorer bättre på att bygga upp datastrukturer medan grafikprocessorer är snabbare på att processera dem.

4 Snabba Fouriertransformen på grafikprocessorer

Fouriertransformen är ett mycket använt verktyg inom vetenskapen och är grunden till många bildbehandlings- och signalbehandlingstekniker[MA03][Toi99].

Fouriertransformen transformerar en signal från tidsplanet till dess frekvensplan. Detta innebär att signalen delas upp i de olika frekvenskomponenterna den består av[Toi99]. Genom att behandla en signal i frekvensplanet kan ofta mycket effektivare algoritmer användas[Hed06].

Fouriertransformen för diskreta serier beräknas med hjälp av diskreta Fouriertransformen (DFT). För att den diskreta Fouriertransformen skall vara användbar i system där beräkningen måste utföras måste snabbare implementationer på beräkning av DFT användas[MA03]. I detta stycke beskrivs den snabba Fouriertransformen (FFT) samt två olika implementationer av denna.

4.1 Matematisk bakgrund

Eftersom diskreta Fouriertransformen kräver n^2 stycken räkneoperationer, är den inte användbar som sådan för transformering av stora datamängder eller för realtidstillämpningar. Antalet beräkningsoperationer i diskreta Fouriertransformen kan lyckligtvis drastiskt minskas genom att observera att Fouriertransformen kan uttryckas som en summa av kortare transformer[CT65]. Denna metod, utvecklad av Cooley och Tukey år 1965, kallas för den snabba Fouriertransformen (FFT). FFT är en "splittra och härska"-algoritm som delar upp transformen rekursivt i halvor. Genom att ordna om indatan kan rekursion undvikas i uppdelningsstadiet.

Den snabba Fouriertransformen för en sekvens

$$\{x(n)\} = \{x(0), x(1), \dots, x(N-1)\}$$

kan härledas ur den diskreta Fouriertransformen på följande sätt:

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi kn/N}, \quad k = 0, 1, \dots, N-1 \quad (4.1)$$

Genom att införa beteckningen $W_N = e^{-j2\pi/N}$ kan ekvationen skrivas i formen

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^k n, \quad k = 0, 1, \dots, N-1 \quad (4.2)$$

Om N är jämn kan sekvensen $\{x(n)\}$ delas upp i underserierna $\{x_{11}(n)\}$ som består av alla element med jämmt ordningsnummer och $\{x_{12}(n)\}$ som består av alla element med udda ordningsnummer.

De diskreta Fouriertransformerna för sekvenserna $\{x_{11}(n)\}$ och $\{x_{12}(n)\}$ ges då av

$$X(k) = \sum_{n=0}^{N/2-1} x_{11}(n)W_{N/2}^{kn}, \quad k = 0, 1, \dots, N/2 - 1 \quad (4.3)$$

$$X(k) = \sum_{n=0}^{N/2-1} x_{12}(n)W_{N/2}^{kn}, \quad k = 0, 1, \dots, N/2 - 1 \quad (4.4)$$

där $W_{N/2}$ definieras som $W_{N/2} = e^{-j2\pi/(N/2)} = e^{(-j2\pi/N)^2} = W_N^2$

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x(n)W_N^{kn} \\ &= \sum_{n=0}^{N/2-1} x(2n)W_N^{k2n} + \sum_{n=0}^{N/2-1} x(2n+1)W_N^{k(2n+1)} \\ &= \sum_{n=0}^{N/2-1} x_{11}(n)W_N^{k2n} + \sum_{n=0}^{N/2-1} x_{12}(n)W_N^{k2n} \cdot W_N^k \\ &= \sum_{n=0}^{N/2-1} x_{11}(n)W_{N/2}^{kn} + \sum_{n=0}^{N/2-1} x_{12}(n)W_{N/2}^{kn} \cdot W_N^k \\ &= X_{11}(k) + X_{12}(k)W_N^k, \quad k = 0, 1, \dots, N/2 - 1 \end{aligned} \quad (4.5)$$

Den komplexa exponentialfunktionens periodicitet ger att

$$\begin{aligned} W_{N/2}^{(k+N/2)n} &= W_{N/2}^{kn} \\ \Rightarrow X_{11}(k + N/2) &= X_{11}(k) \quad \text{och} \quad X_{12}(k + N/2) = X_{12}(k) \end{aligned}$$

Senare delen av $\{X(k)\}$ ges då av

$$X(k + N/2) = X_{11}(k) + X_{12}(k)W_N^{k+N/2}, \quad k = 0, 1, \dots, N/2 - 1 \quad (4.6)$$

där $W_N^{k+N/2} = -W_N^k$, vilket medför att

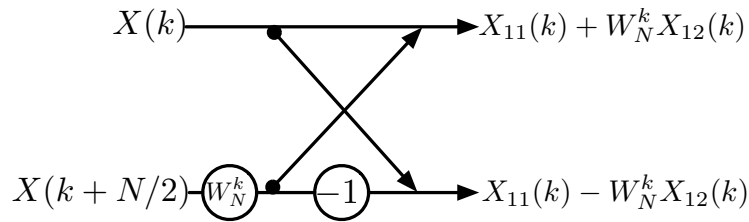
$$X(k + N/2) = X_{11}(k) - X_{12}(k)W_N^k, \quad k = 0, 1, \dots, N/2 - 1 \quad (4.7)$$

Fouriertransformen $X(k)$ för den ursprungliga sekvensen kan alltså uttryckas med hjälp av Fouriertransformerna $X_{11}(k)$ och $X_{12}(k)$, räknade ur underserierna $x_{12}(k)$ respektive $x_{12}(k)$.

$$X(k) = X_{11}(k) + W_N^k X_{12}(k), \quad k = 0, 1, \dots, N/2 - 1 \quad (4.8)$$

$$X(k + N/2) = X_{11}(k) - W_N^k X_{12}(k), \quad k = 0, 1, \dots, N/2 - 1 \quad (4.9)$$

Denna procedur upprepas på underserier tills serier med endast ett element kvarstår. Därefter transformeras underserierna. Fouriertransformen för den ursprungliga serien fås genom successiv användning av 4.8 och 4.9. Ekvationerna 4.8 och 4.9 kallas för "butterfly"-operationer på grund av sin struktur, jämför med figur 4.1.



Figur 4.1: Grafisk representation av butterfly-operationen[Hed06].

Före FFT kan köras måste elementen i indatan ordnas om i bitreverserad ordning av ordningsnumret. Tabell 4.1 visar ordningen på ursprungliga datan och indatan till FFT. Genom att ordna om sekvensen på detta sätt undviks den rekursiva uppsjälkningen av indatan.

FFT kräver $\frac{n}{2} \log n$ stycken komplexa multiplikationer och $n \log n$ stycken komplexa additioner jämfört med den diskreta Fouriertransformen som kräver n^2 komplexa multiplikationer och $n(n - 1)$ komplexa additioner. Minskningen av krävda räkneoperationer innebär även mindre avrundningsfel[Toi99]. FFT kan beräknas på plats, d.v.s. att samma minnespositioner varifrån indatan lästes används för

Ursprunglig sekvens	Adress	Bitreverserad adress	FFT inserie
x(0)	000	000	x(0)
x(1)	001	100	x(4)
x(2)	010	010	x(2)
x(3)	011	110	x(6)
x(4)	100	001	x(1)
x(5)	101	101	x(5)
x(6)	110	011	x(3)
x(7)	111	111	x(7)

Tabell 4.1: Ordning av element för FFT[Toi99].

lagring av utdata.

Denna implementation av FFT fungerar endast på indata av längden 2^n , men flera andra implementationer klarar av även godtyckliga längder. Både FTTW och CUFFT som diskuteras i senare stycken klarar av transformer av alla längder, även om vissa längder kan leda till förlängd körtid.

4.2 Fastest Fourier Transform in the West

Fastest Fourier Transform in the West (FFTW) är ett mycket optimerat och använt mjukvarubibliotek för beräkning av diskreta Fouriertransformen. FFTW är inte bunden till någon specifik hårdvara, utan använder sig av en planeringsprogram som adapterar algoritmen för den befintliga hårdvaran[FJ05].

Körningen av FFTW delas upp i två stadier. Först körs ett planeringsprogram som jämför körtiden hos olika metoder för beräkning av FFT och väljer den snabbaste metoden för den givna hårdvaran. Detta planeringsskede är tidskrävande, men gör själva beräkningen av DFT snabbare. Om endast ett fåtal transformer behöver beräknas, kan planeringsprogrammet köras så att den snabbt ger en bra metod, men inte nödvändigtvis den bästa, för att minska körtiden i detta stadiet. Efter planeringsskedet delar FFTW rekursivt upp problemet i mindre problem tills de är tillräckligt små. Dessa delar transformeras sedan med en optimerad algoritm[FJ05].

FFTW-biblioteket är skrivet i C och går att använda på flera olika hårdvaruarkitekturer. Körtiden för FFTW är $O(n \log n)$ oavsett längden på n .

Flera andra FFT implementationer kräver $O(n^2)$ körtid för vissa n . FFTW klarar av att beräkna transformer av flerdimensionell indata och är inte begränsad till en viss mängd dimensioner[FJ05].

4.3 CUDA Fast Fourier Transform Library

CUDA Fast Fourier Transform Library (CUFFT) ett bibliotek för beräkning av FFT och hör till Nvidias CUDA. CUFFT biblioteket är modellerat efter FFTW[BDH⁺07]. CUFFT stöder en, två och tredimensionella transformer med reell och imaginär data. Två och tredimensionella transformerna är begränsade till en storlek på 16384 element, endimensionella till 8 miljoner element. Flera endimensionella transformer kan köras parallellt med CUFFT[NVI07b].

CUDA använder olika algoritmer för FFT beroende på indatan. Ifall indatan är tillräckligt liten för att rymmas i CUDA:s delade minne och transformerna är potenser av samma faktor är CUFFT effektivast. Ifall indatans storlekar inte är potenser av samma faktor använder, CUFFT en mer generell algoritm för beräkning som har större felmarginer och är långsammare. Ifall ingendera av villkoren uppfylls används en algoritm som sparar alla mellanresultat i grafikkminnet. CUFFT:s användbarhet är begränsad då det endast kan användas på Nvidia-grafikprocessorer[BDH⁺07].

4.4 Jämförelse av prestanda

Prestandan av FFTW och CUFFT har jämförts av Hugh Merz på University of Waterloo[Mer07]. Som testutrustning användes en dator med 2 stycken tvåkärniga Intel Xeon 5150 centralprocessorer med en klockfrekvens på 2.66GHz. Moderkortet hade ett Intel 5000X chipset och som grafikkort användes ett NVIDIA Quadro FX 4600 kopplat fast i PCI Express 1.0 x16 buss. I testen användes FFTW, version 3.1.2, med SSE stöd och FFTW_MEASURE för genererande av planen. CUDA och CUFFT var båda version 1.1 [Mer07].

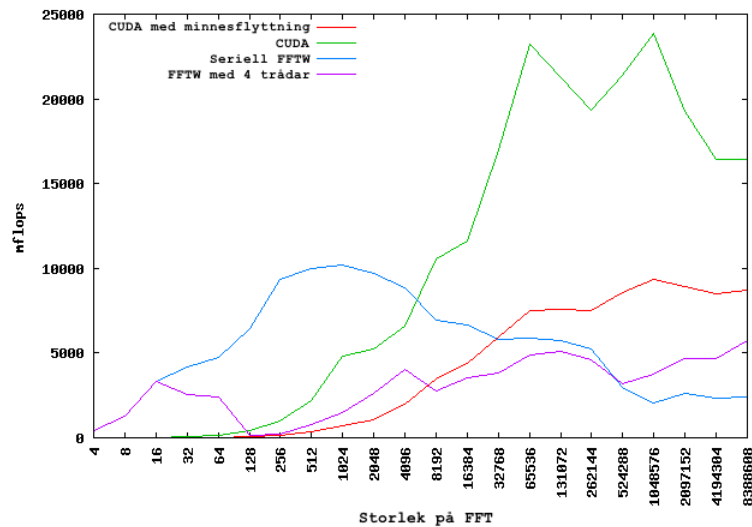
Enheten i graferna är MFLOPS, miljoner flyttalsoperationer per sekund, beräknas enligt

$$MFLOPS = \frac{5N \log_2(N)}{T_{FFT}},$$

där T_{FFT} är tiden för en Fouriertransform i mikrosekunder. Ifall datan är endast reell divideras värdet med 2. mflops är enheten som används av benchFFT[FJ08], som är ett standard prestandatest för olika implementationer av snabba Fouriertransformen.

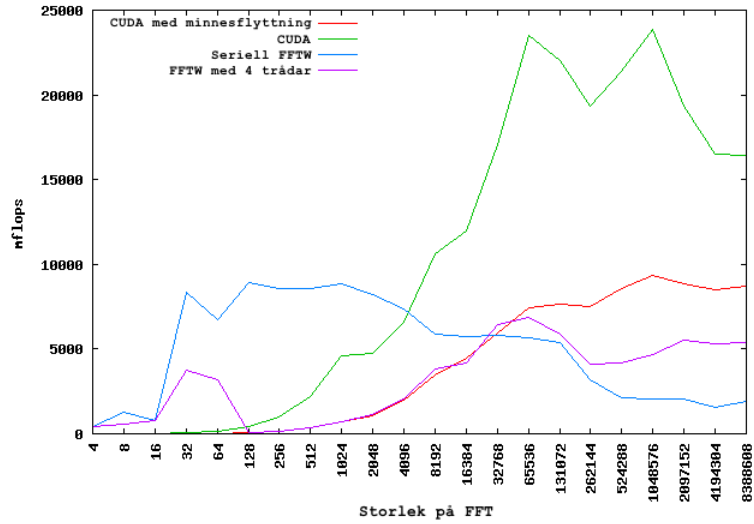
Figurerna 4.2 – 4.5 visar prestandan för de två implementationerna i två olika fall:

- CUFFT med tiden för dataöverföringen till grafikminnet medräknat
- CUFFT utan dataöverföringstid
- Seriell körning av FFTW
- FFTW med 4 programtrådar

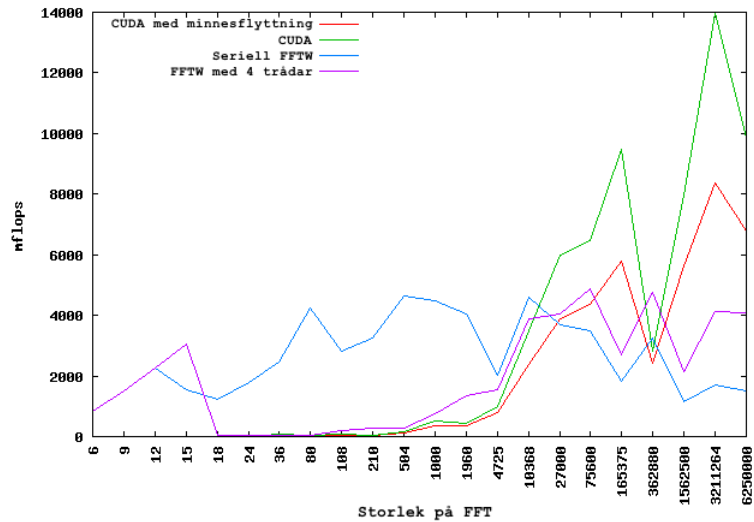


Figur 4.2: Beräkning av FFT med storlekar i potenser av två. Beräknad på plats[Mer07].

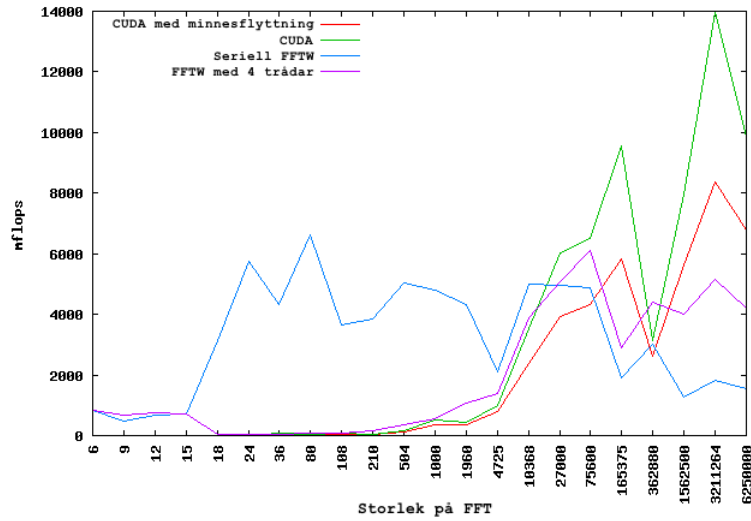
Graferna visar att det krävs stora transformeringar för att kompensera för tiden som går åt att flytta över data till och från grafikkortet. I stora transformeringar är CUFFT



Figur 4.3: Beräkning av FFT med storlekar i potenser av två. Inte beräknad på plats[Mer07].



Figur 4.4: Beräkning av FFT med storekar som inte är potenser av två. Beräknad på plats[Mer07].



Figur 4.5: Beräkning av FFT med storlekar som inte är potenser av två. Inte beräknad på plats[Mer07].

10 gånger snabbare än seriell FFTW och 5 gånger snabbare än den 4-trådade FFTW:n. I små transformeringar är FFTW snabbare då den inte behöver flytta datan lika långt och dess cacheminne räcker till för att lagra all data.

Jämförandet visar klart att grafikprocessorer lämpar sig mycket väl till beräkning av Fouriertransformer. Även om beräkningsintensiteten hos FFT inte är helt optimal för GPGPU, kan detta åtgärdas med att ha tillräckligt med data att beräkna. Även i fallen där grafikprocessorn inte är nämnvärt snabbare än centralprocessorn är det bra att inse att belastningen för beräkningen av FFT är borttagen från centralprocessorn som kan ge mera resurser åt andra processer.

5 Diskussion

Jämförandet av FFT-prestanda var ursprungligen menad att utföras mellan FFTW och GPUFFT-biblioteken[GM08]. Tyvärr fanns inte nödvändig hårdvara tillgänglig för att utföra jämförandet själv. Ingen bra studie som jämförde dessa två metoders prestanda fanns heller tillgänglig. Det var svårt att hitta objektiv information om CUFFT då majoriteten av materialet härstammade

från Nvidia. Detta innebar att materialet sällan tog upp negativa sidor hos CUFFT.

En detalj i prestandatestet som skulle ha varit intressant att undersöka vidare var det drastiska fallet i MFLOPS för CUFFT vid 362880 element, medan FFTW ökar i prestanda vid samma punkt. Då resultaten är baserade på egen data kunde avvikelserna inte undersökas noggrannare, men den berodde antagligen på att CUDA bytte till en långsammare algoritm för den mängden element.

En av de största problemen i GPGPU-programmering ur en utvecklarens synpunkt är att både CTM och CUDA är hårdvarubundna tillämpningar. Att utveckla skilda program för olika hårdvaror är mycket arbetsamt och dyrt. Genom att Nvidia och AMD skulle stöda ett gemensamt programmeringsspråk t.ex. Brook och hjälpa dem göra optimeringar för bådas grafikprocessorer kunde de gynna hela GPGPU-programmeringssamhället. För att GPGPU skall bli mera allmänt använd måste ett hårdvaruobundet programmeringssätt finnas.

Ett annat problem GPGPU-programmerare hamnar ut för är att den underliggande hårdvarustrukturen i stort sett är hemlig för programmerare, vilket gör hårdvaruoptimeringar mycket svåra att göra. Denna situation håller som tur på att förbättras. I september 2007 gav AMD ut hårdvarubeskrivningar av några sina grafikchipset för att stöda utvecklingen av öppna drivrutiner för Linux [AMD07].

GPGPU kan vara till mycket stor nytta inom vetenskaplig forskning på grund av den låga kostnaden och höga prestandan. Som exempel kan nämnas Folding@Home-projektet som utför medicinsk forskning på ett stort nätverk som består av beräkningsklienter installerade på personliga datorer av frivilliga personer. I oktober 2006 utgavs en GPGPU-klient och inom en månad hade GPU-klienterna utfört en beräkningsmängd motsvarande 18% av all beräkning utförd av Windows-klienter på 6 år[LH07].

GPGPU-programmering möjliggör parallellprogrammering av hundratals processorkärnor redan idag. Eftersom mängden kärnor i centralprocessorer ökar hela tiden kommer kunskap i parallellprogrammering att vara viktigt i framtiden. Att undervisa kurser i GPGPU-programmering kunde ge studerande en värdefull insikt i parallellprogrammering.

Referenser

- [Adv06] Advanced Micro Devices. *ATI CTM Guide*, 2006.
- [AMD07] AMD Details Strategic Open Source Graphics Driver Development Initiative, September 2007. URL: http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543~119372,00.html.
- [BDH⁺07] Ian Buck, Bernard Deschizeaux, Mark Harris, John Owens, Jim Phillips och John Stone. Supercomputing 2007 tutorial: High performance computing with cuda. 2007. URL: <http://www.gpgpu.org/sc2007/>.
- [BFH⁺] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugarman, Pat Hanrahan, Mike Houston och Kayvon Fatahalian. Brook language.
- [Buc07] Ian Buck. GPU computing with NVIDIA CUDA. I: *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, s 6, New York, NY, USA, 2007. ACM.
- [CT65] James W. Cooley och John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, April 1965.
- [Fer08] Antonio Ramires Fernandes. View frustum culling tutorial, Februari 2008. URL: <http://www.lighthouse3d.com/opengl/viewfrustum/>.
- [FJ05] Matteo Frigo och Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special Issue on Program Generation, Optimization, and Platform Adaptation”.
- [FJ08] Matteo Frigo och Steven G. Johnson. benchFFT, 2008. URL: <http://www.fftw.org/benchfft/>.
- [FQKYS04] Zhe Fan, Feng Qiu, A. Kaufman och S. Yoakum-Stover. GPU cluster for high performance computing. *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, ss 47–47, 06-12 Nov. 2004.

- [GM08] Naga K. Govindaraju och Dinesh Manocha. GPUFFT: High Performance Power-of-Two FFT Library using Graphics Processors, 2008. URL: <http://gamma.cs.unc.edu/GPUFFT/>.
- [GR05] Jayanth Gummaraju och Mendel Rosenblum. Stream programming on general-purpose processors. I: *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, ss 343–354, Washington, DC, USA, 2005. IEEE Computer Society.
- [Hed06] Johan Hedborg. GPGPU. Examensarbete, Linköpings Universitet, Maj 2006.
- [hH05] Shih hsuan Hsu. GPGPU programming, Augusti 2005. URL: http://www.cmlab.csie.ntu.edu.tw/~vincent/resource/gpgpu/GPGPU_Programming.pdf.
- [Hou07] Mike Houston. Welcome & overview. I: *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, s 1, New York, NY, USA, 2007. ACM.
- [IIH06] T. Ikeda, F. Ino och K. Hagihara. A code motion technique for accelerating general-purpose computation on the GPU. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, ss 10 pp.–, 25-29 April 2006.
- [LH07] David Luebke och Greg Humphreys. How GPUs work. *Computer*, 40(2):96–100, Feb. 2007.
- [LHG⁺06] David Luebke, Mark Harris, Naga Govindaraju, Aaron Lefohn, Mike Houston, John Owens, Mark Segal, Matthew Papanikolaou och Ian Buck. Gpgpu: general-purpose computation on graphics hardware. I: *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, s 208, New York, NY, USA, 2006. ACM.
- [MA03] Kenneth Moreland och Edward Angel. The FFT on a GPU. I: *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ss 112–119, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

- [Mer07] Hugh Merz. CUFFT vs FFTW comparison, 2007. URL: http://www.science.uwaterloo.ca/~hmerz/CUDA_benchFFT/.
- [Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 19 April 1965.
- [NVI07a] NVIDIA Corporation. *NVIDIA CUDA — Compute Unified Device Architecture, Programming Guide*, November 2007.
- [NVI07b] NVIDIA Corporation. *CUFFT Library*, Oktober 2007.
- [OLG⁺07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn och Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.
- [Owe07a] John Owens. Data-parallel algorithms and data structures. I: *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, s 3, New York, NY, USA, 2007. ACM.
- [Owe07b] John Owens. Gpu architecture overview. I: *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, s 2, New York, NY, USA, 2007. ACM.
- [Toi99] Hannu Toivonen. *Introduktion till signalbehandling*. Åbo Akademi, 1999.
- [Wat05] Alan Watt. *Advanced game development with programmable graphics hardware*. Wellesley, MA : A K Peters, 2005.
- [Wik08a] Graphics processing unit, Februari 2008. URL: http://en.wikipedia.org/wiki/Graphics_processing_unit.
- [Wik08b] Stream Processing, Februari 2008. URL: http://en.wikipedia.org/wiki/Stream_processing.

A Lista över använda begrepp

Alfa-kanal	Ett värde som bestämmer genomskinligheten för en färg
ALU	Aritmetisk-Logisk Enhet — utför beräkningar och logiska funktioner i en processor
CPU	Centralprocessor
CUFFT	Ett bibliotek för att beräkna FFT på Nvidia grafikprocessorer
DFT	Digital Fouriertransform
FFT	Den snabba Fouriertransformen, en optimerad version av DFT
FFTW	Ett bibliotek för att beräkna FFT på centralprocessorer
FLOP	Flyttalsoperation
Fragment	All den data som behövs för att beskriva en pixel
Fragmentprocessor	Del av grafikpipelinen som har SIMD-arkitektur och utför operationer på fragment.
GPU	Grafikprocessor
GPGPU	General-Purpose Computing on Graphics Processing Units, allmänna beräkningar på grafikprocessorer
IEEE	Institute of Electrical and Electronics Engineers — standardiseringsorganisation
Kernel	Ett program som körs på element i strömprogrammering
MIMD	Multiple Instruction stream, Multiple Data stream. Olika beräkningar körs på olika data parallellt.
Pipeline	Ett antal seriellt sammanbundna processteg
Pixel	Bildpunkt

Rasterisering	Förvandling av vektorgrafik till diskreta bildpunkter
Renderering	Processen att omvandla en 3-dimensionell modell till den 2-dimensionella modell som visas på skärmen
SIMD	Single Instruction, Multiple Data. Samma beräkningsoperation körs över flera dataelement samtidigt
Shader	Ett specialprogram för programmering av grafikpipelinen
Ström	En samling av data
Strömprocessor	En samling parallella processorer som fungerar enligt SIMD-principen
Textur	Bitkarta som används för att tillägga färger och detaljer till ytor i datorgrafik
Vertex	Hörnpunkt i triangel
Vertexprocessor	Del av grafikpipelinen som har SIMD-arkitektur och utför beräkningar på vertex.
Z-gallring	Bortlämning av punkter som blir bakom andra punkter