KTH Computer Science
and Communication

GPGPU - General-Purpose computing on Graphics Processing Units

Att utnyttja GPU istället för CPU

EXAMENSARBETE INOM DATALOGI, GRUNDNIVÅ (DD143X)

Namn:	Oskar Bodemyr	Jimmy Larsson
Adress:	Visättravägen 30	Studentbacken 21 -1010
Postnummer:	141 50 HUDDINGE	115 57 STOCKHOLM
Telefon:	073 805 36 17	073 588 74 61
E-post:	obodemyr@kth.se	jilars@kth.se

Examenrapport vid NADA
Handledare: Mads Dam
Examinator: Mårten Björkman

TRITA xxx yyyy-nn

Abstract

The following report has been written by Jimmy Larsson and Oskar Bodemyr as a bachelor's thesis in order to reach a bachelor's degree in the subject of Computer Engineering.

At the time of writing this report, the most common compute device is the CPU. This report looks at the possible usages of GPGPU and how one might instead use the GPU for General Purpose Computations. The main question that the report will face is the following:
Is it possible, using a non-trivial parallel algorithm, to show that a GPU is better suited to perform certain calculations than a CPU?

The implementation of two test algorithms in order to study the basic functions of GPGPU, followed by an implementation of a matrix-multiplier provided data supporting the question of the report. The data itself proves that certain algorithms such as the matrix-multiplier of large amounts of data might take about 30 seconds to calculate on a CPU, while the same calculations can be done on a GPU in less than half a second.

Innehåll

1 Inledning **1**
- 1.1 Problemformulering . . . 1
- 1.2 Syfte . . . 1
- 1.3 Tillvägagångssätt . . . 2
- 1.4 Arbetsfördelning . . . 2

2 Bakgrund **3**
- 2.1 CPU vs GPU . . . 3
- 2.2 GPGPU . . . 4
- 2.3 OpenCL . . . 6
  - 2.3.1 Definitioner . . . 6
  - 2.3.2 Programmering inom OpenCL . . . 8
  - 2.3.3 Körschema för programmering i OpenCL . . . 15

3 Resurser **19**
- 3.1 Hårdvara . . . 19
  - 3.1.1 GPU . . . 19
  - 3.1.2 CPU . . . 20
- 3.2 Mjukvara . . . 20

4 Algoritmer **21**
- 4.1 Testförsök 1: vectorSquare . . . 21
  - 4.1.1 Utförande . . . 21
  - 4.1.2 Resultat . . . 22
- 4.2 Testförsök 2: dotProduct . . . 23
  - 4.2.1 Utförande . . . 23
  - 4.2.2 Resultat . . . 24
- 4.3 Huvudförsök: matrixMultiplication . . . 25
  - 4.3.1 Utförande . . . 25
  - 4.3.2 Antagande . . . 27
  - 4.3.3 Resultat . . . 29

5 Diskussion **33**
- 5.1 Felkällor . . . 35

6 Slutsats **37**

7 Ordlista och förkortningar **39**

Litteraturförteckning **41**

Figurer **44**

Kapitel 1

Inledning

Följande kapitel beskriver problemformuleringen som denna rapport bygger på samt syftet och tillvägagångssättet.

1.1 Problemformulering

Denna rapport kommer att behandla huvudfrågan:
Kan man med hjälp av en icke-triviell parallell algoritm visa att GPU är mer lämpad att utföra vissa beräkningar än CPU?

Samt följdfrågan:
Vad gör att en GPU är bättre eller sämre lämpad än CPU i det här fallet?

Rapporten behandlar strikt effektivitetsaspekten mellan GPU och CPU i problemformuleringen. Detta innebär att inga resultat kommer att behandlas ur ett monetärt perspektiv.

1.2 Syfte

Syftet med detta projekt är att utvärdera huruvida GPU är bättre lämpad än CPU för vissa beräkningar. Med resultaten av detta kan man möjligtvis dra slutsatser om andra användningsområden för GPU där man i nuläget utnyttjar CPU. En annan del av syftet är även att införskaffa kunskaper inom ett område som fortfarande inte utnyttjas till sin fulla kapacitet, men som visar god potential inför framtiden.

1.3 Tillvägagångssätt

Tillvägagångssättet i denna rapport har skett i tre huvudfaser.

1) Efterforskningsfasen: Fasen består av efterforskning inom området GPGPU. Vad är GPGPU? Hur används det? Hur kan vi utnyttja detta?

2) Praktiska fasen: Denna fas består av praktiska försök där kunskaper från första fasen samt tidigare kunskaper utnyttjas till fullo. I denna fas implementeras olika algoritmer vars resultat används för fler praktiska försök i denna fas samt för resterande moment i sammanställningsfasen.

3) Sammanställningsfasen: I den sista fasen behandlas och sammanställs all data för att kunna analyseras, jämföras samt för att dra slutsatser. Dessa slutsatser kommer sedan att avsluta denna rapport.

Genom att implementera en icke-triviell algoritm och utföra beräkningar med CPU respektive GPU och sedan analysera den data som fås ut, kommer frågorna i problemformuleringen förhoppningsvis att besvaras. Grova antaganden om hur resultaten borde se ut kommer att göras utifrån specifikationer på hårdvara och antalet beräkningar som krävs.

Huvudsakligen kommer mjukvaran att bestå av en implementation av OpenCL. Då OpenCL har stöd för ett flertal programmeringsspråk kommer denna rapport att hantera språket C++ då dokumentation för OpenCl hanterar detta språk först och främst. Vilken hårdvara som används kommer att avgöras av tillgänglighet och kompabilitet.

1.4 Arbetsfördelning

Följande uppdelningar har gjorts:

Oskar Bodemyr	**Jimmy Larsson**
2.2 GPGPU	2.3.2 Programmering inom OpenCL
2.3.1 Definitioner	2.3.3 Körschema för programmering inom OpenCL
3.1 Hårdvara	4.1 Testförsök 1: vectorSquare
4.2 Testförsök 2: dotProduct	5.1 Felkällor

Resterande delar har gjorts gemensamt.

Kapitel 2

Bakgrund

Följande kapitel kommer att handla om ämnet GPGPU. Det förklarar vad huvudskillnaden mellan GPU och CPU är, vad GPGPU är, samt OpenCL; ett ramverk som gör detta implementerbart på ett effektivt sätt.

Tekniska ord förklaras i ordlistan som finns i slutet av rapporten.

2.1 CPU vs GPU

Inom typisk programmering är processorn den vanligaste beräkningsenheten då den på ett smidigt och samtidigt effektivt sätt kan använda data för att utföra en mängd olika operationer. En del av syftet med denna rapport är att lyfta fram grafikkortet som en lämplig konkurrent för vissa typer av operationer.

En CPUs arkitektur består av få kärnor som delar på gemensamma cacheminnen. Med detta hanteras bara ett fåtal trådar samtidigt, till skillnad från en GPU som har förmågan att köra tusentals trådar samtidigt. Denna förmåga möjliggörs tack vare att en GPU kan ha hundratals kärnor i kontrast till en CPUs fåtal[1]. I dagsläget är quadcoretekniken väldigt populär vilket innebär att 4 kärnor används. En anledningen till varför CPU ändå lämpar sig bättre än GPU i många fall är beräkningshastigheten. Vilken beräkningsenhet man bör använda är alltså en fråga om att göra beräkningar snabbt, men endast en eller ett fåtal samtidigt, eller göra väldigt många beräkningar samtidigt, men där beräkningshastigheten är långsammare.

Inom datorgrafiksberäkningar används väldigt ofta stora mängder data i form av vektorer eller texturer för att utföra specifika algoritmer parallellt. Detta kan exempelvis vara att man för en viss bild, vill tona ner färgen för varje enskild pixel med ett visst proc entantal. Detta är en typisk beräkning som lämpar GPU mycket bättre än CPU på grund av dess arkitektur.

Moderna GPUer är ofta SIMD-implementationer. SIMD står för single instruction

multiple data, det vill säga, en instruktion till en stor mängd data. Vid beräkning på flera element med en CPU vore det typiska att hämta ett element, utföra en beräkning på detta och sedan hämta nästa. Med hjälp av SIMD kan man istället hämta flera element och utföra denna beräkning på alla samtidigt.

Ett av de större problemen med att använda GPU är att de har en väldigt specialiserad design[2]. Ett annat problem är det faktum att man vid programmering med en GPU måste föra över den data som används till GPU-enhetens egna minne innan den kan användas, något som tar relativt lång tid.

2.2 GPGPU

GPGPU står för *General-Purpose computation on Graphics Processing Units*, dvs beräkningar på grafikprocessor för allmänna ändamål. Området är även känt som *GPU Computing*. Tanken med GPGPU är att man utnyttjar datorns grafikprocessor (GPU) istället för centralprocessorn (CPU) vid beräkningar.

Tekniken började användas kring millennieskiftet av datavetare samt forskare inom bland annat medicin och elektromagnetism[3]. Man upptäckte att användande av GPU istället för CPU vid vissa vetenskapliga beräkningsprocesser ledde till en stor ökning av prestanda.

CPU-tekniken har under en lång period utvecklats snabbt och till stor del följt den kända Moores lag[4], som säger att antalet transistorer på ett CPU-chip fördubblas ungefär vartannat år. Trots att lagen fortfarande stämmer kan man inte förbättra klockhastigheterna lika mycket som man har gjort tidigare. På grund av detta har utvecklare börjat fokusera mer och mer på parallellisering. Detta innebär i sin tur att utvecklare ser på andra alternativ där parallelliseringen kan utnyttjas till högre grad, vilket i sin tur leder till att utvecklingen och användningen av GPGPU börjar ske i högre grad.

I det tidiga stadiet av GPGPU var tekniken väldigt svår att använda. Man var tvungen att använda sig av grafik-API:er som t.ex. OpenGL och Direct3D där man saknade dokumentation om hur hårdvaran utnyttjades[5]. Även om detta var komplicerat och i många fall inte gav bra resultat har detta stadie bidragit med mycket till vad GPGPU är idag.

Det var inte förrän i slutet av 2006 som tekniken verkligen började ta fart då NVIDIAs *CUDA* och AMDs *Stream* lanserades[5]. CUDA-plattformen skapades av NVIDIA för att möjliggöra enkel GPU-programmering på NVIDIAs grafikprocessorer. De nämner själva att när CUDA demonstrerades 2006 var det världens första lösning för allmänna beräkningar på GPUer[6]. AMDs Stream, även känt som ATI Stream Technology, skapades på samma sätt för att möjliggöra GPU-

programmering på AMDs grafikprocessorer.

Ett annat landmärke inom GPGPU är Open Computing Language, även känt som OpenCL. Vad som gör OpenCL speciellt är att det är ett ramverk som gör det möjligt att skriva kod som kan exekveras både på GPU och CPU, utan att behöva ta hänsyn till den underliggande processortypen. OpenCL-projektet startades av Apple och lämnades över till Khronos Group 2008 för att utöka tekniken för att fungera på flera plattformar[7]. Första utgåvan av OpenCL (v1.0) släpptes med Apples operativsystemsuppdatering Mac OS X Snow Leopard. Khronos uppdaterade version, OpenCL v1.1, släpptes den 14:e juni 2010 och hade då stöd även för Windows och Linux.

Eftersom Khronos är en ideell grupp är OpenCL gratis att använda. Det stöds av både AMD och NVIDIA samt Intel och är därför ett utmärkt ramverk för att demonstrera fördelarna med GPGPU. Mer om hur OpenCL fungerar går att läsa senare i rapporten.

(a) ATI Stream (b) NVIDIA CUDA (c) OpenCL

Figur 2.1: Logos

2.3 OpenCL

OpenCL-ramverket använder sig av många termer som beskriver beståndsdelarna som behövs för att skapa ett fullständigt OpenCL-projekt.

Som tidigare nämnt är OpenCL en väldigt bra lösning för att jämföra CPU-programmering med GPU-programmering då ett program kan exekveras på båda. I denna del kommer man att kunna läsa definitioner av OpenCLs viktigaste termer och se programmeringsexempel på hur delarna i ett program skrivs. Tillslut visas ett körschema som demonstrerar hur ett projekt i OpenCL skrivs.

2.3.1 Definitioner

Denna del kommer att förklara grundläggande termer inom området som är nödvändiga att lära sig om man vill programmera i OpenCL. Dessa definitioner förklaras även i Khronos Groups OpenCL-specifikationer[8].

Device
För att kunna exekvera kod i OpenCL behövs någonting att exekvera koden på. För detta använder man inom OpenCL termen *device*. En device består av en flera *compute units*. En device kan exempelvis vara en GPU eller CPU.

Compute unit
Termen *compute unit* används för att beskriva en *devices* beståndsdelar, beräkningsenheter. En *compute unit* på CPU kan då alltså vara en processorkärna och en samling streamcores på GPU.

Work-group
En *work-group* är en samling av *work-items* som exekveras på en *compute unit*.

Work-item
Varje *work-item* i en *work-group* är en del av en samling parallella exekveringar av en *kernel*. Det är alltså *work-items* som delas upp för att köras parallellt. Ett *work-item* exekveras av ett eller flera *processing elements*.

Processing element
Varje *processing element* består av en virtuell skalärprocessor. En skalärprocessor tillhör de allra enklaste processortyperna. De klassas som SISD-processorer, dvs *single instruction, single data*: en instruktion för ett dataelement.

Kernel
Det enklaste sättet att se en *kernel* är som en funktion som kommer att köras på en *device*. Ett *program* kan byggas upp av en eller flera *kernels*.

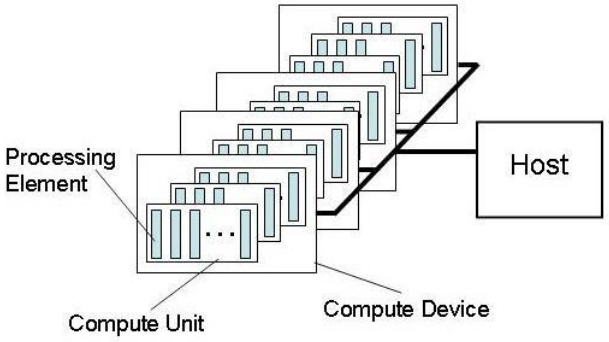
Program
En uppsättning *kernels*.

Command-Queue
En *command-queue* är en samling av kommandon som specificerar hur *kernels* ska exekveras på *devices*. Dessa kan exekveras både i ordning och i oordning.

Context
Termen *context* används för att beskriva miljön eller omgivningen: hur alla delar hör ihop. *Context* inkluderar en uppsättning *devices* och dess minne och minnesegenskaper samt en eller flera *command-queues* som används för att köra en eller flera *kernels*.

Host
För att kunna använda sig av OpenCL behövs något som samverkar med *context* med hjälp av OpenCLs API. För detta används termen *host*.

Platform
För att kunna köra applikationer, dela resurser och exekvera *kernels* behövs *platform*. Denna innehåller *host* och en samling *devices* att exekvera på.



Figur 2.2: En grafisk representation av hur delarna i *platform* är sammankopplade.

Global memory och global worksize
I OpenCLs exekveringsmodell kan data refereras både globalt och lokalt. *Global memory* är en region som kan nås av alla *work-items* som körs i en *context*. *Global worksize* säger hur stort problemet är och bestämmer därmed hur många minnespositioner behövs för *global memory*.

Local memory och local worksize
Den minnesregion som bara kan nås av en specifik *work-group* kallas för *local memory*. Därför kan bara *work-items* inom den gruppen nå detta minne. Detta är bundet

till *local worksize*, som säger hur många *work-items* som ska används samtidigt. Denna storlek får inte överstiga storleken på hela problemet. Dessutom bör *global worksize* vara delbart på *local worksize*.

Buffer objects
De minnesobjekt som en *kernel* behöver skapas som *buffer objects*. En *kernel* på en *device* kommer att nå dessa objekt med hjälp av pekare. Det här behövs för att kunna sätta argument till en *kernel*, vilket är jämförbart med ett metodanrop med argument inom exempelvis Java eller C++.

2.3.2 Programmering inom OpenCL

Att programmera med hjälp av OpenCL är jämförtbart med programmering i andra programmeringsspråk som exempelvis Java eller C++ även om skillnader uppenbarligen existerar. I det här avsnittet förklaras några av de olika funktionerna och dess argument.

För en komplett lista av funktioner och dess argument inom OpenCL 1.1 föreslås en .pdf publicerad av Khronos Group[8]

Platform

```
cl_int clGetPlatformIDs(cl_uint num_entries,
                        cl_platform_id *platforms,
                        _uint *num_platforms)
```

`clGetPlatformIDs` returnerar `CL_SUCCESS` om exekveringen av funktionen utförs korrekt.

`num_entries` är antalet `cl_platform_id` poster som platforms ska kunna hantera.
`platforms` returnerar en lista av möjliga OpenCL-plattformar.
`num_platforms` returnerar antalet tillgängliga plattformar.

Device

```
cl_int clGetDeviceIDs ( cl_platform_id platform,
                        cl_device_type device_type,
                        cl_uint num_entries,
                        cl_device_id *devices,
                        cl_uint *num_devices)
```

`clGetDeviceIDs` returnerar `CL_SUCCESS` om exekveringen av funktionen utförst korrekt.

`platform` tar emot ett platform ID som returnerats vid `clGetPlatformIDs`. Fördefinerade värden körs om `NULL` används som argument.
`device_type` används för att begära DeviceIDs av en viss typ enligt exempelvis följande lista:

- `CL_DEVICE_TYPE_CPU` för att använda sig av en CPU med en eller flera kärnor.
- `CL_DEVICE_TYPE_GPU` för att använda sig av en GPU.
- `CL_DEVICE_TYPE_ALL` använder alla möjliga `devices` som finns tillgängliga.

`num_entries` är antalet `cl_device` poster som kan hanteras som `devices`.
`devices` returnerar en lista av `devices` enligt tidigare argument.
`num_devices` returnerar antalet tillgängliga devices.

Work-group

```
cl_int clGetKernelWorkGroupInfo (cl_kernel kernel,
                                 cl_device_id device,
                                 cl_kernel_work_group_info param_name,
                                 size_t param_value_size,
                                 void *param_value,
                                 size_t param_value_size_ret)
```

`clGetKernelWorkGroupInfo` returnerar information angående en `device` i en given `kernel`.

`kernel` specifierar vilken `kernel` funktionen ska arbeta med.
`device` specifierar vilken `device`, associerad med en `kernel` som ska behandlas. Om `kernel` endast innehåller en `device` kan `NULL` användas som argument.
`param_name` är den parameter som motsvarar informationen man är intresserad av att få ut. Möjliga parametrar går att hitta i tabell 5.14 i Khronos OpenCL handbok[8].
`param_value_size` används för att specificera vilken storlek angiven i byte, resultatet som `param_value` argumentet pekar på, bör ha. Denna storlek måste vara lika stor som resultatet eller större.
`param_value` är den pekare som visar var det sökta resultatet ska hamna.
`param_value_size_ret` returnerar den korrekta storleken på den data som överförts till `param_value` positionen.

Kernel

```
cl_kernel clCreateKernel (cl_program program,
                          const char *kernel_name,
                          cl_int *errcode_ret)
```

`clCreateKernel` returnerar ett `kernel`-objekt vid felfri exekvering.

`program` är ett fungerande `cl_program`-objekt.
`kernel_name` är det funktionsnamn som användes i `cl_program`-objektet.
`errcode_ret` returnerar `CL_SUCCESS` om inga fel påträffades under exekveringen.

```
cl_int clEnqueueNDRangeKernel ( cl_command_queue command_queue,
                                cl_kernel kernel,
                                cl_uint work_dim,
                                count size_t *global_work_offset,
                                count size_t *global_work_size,
                                count size_t *local_work_size,
                                cl_uint num_events_in_wait_list,
                                const cl_event *event_wait_list,
                                cl_event *event)
```

`clEnqueueNDRangeKernel` returnerar `CL_SUCCESS` om inga fel påträffades under kernel-exekveringen.

`command_queue` är det `cl_command_queue` objekt innehållande den `device` som `kernel`-objeket ska använda sig av.
`kernel` är den `kernel` som ska användas. Denna `kernel` måste vara den samma för både det associerade `context` objektet samt `command-queue` argumentet.
`work_dim` motsvarar antalet globala dimensioner som ska användas vid exekveringen. Detta argument måste vara större än 0 men mindre eller ekvivalent med `devicens` maximala antal `work-item` dimensioner.
`global_work_offset` är den förskjutning som ska användas vid val av trådexekvering.
`global_work_size` motsvarar de globala dimensioner som ska användas.
`local_work_size` motsvarar de lokala dimensioner som ska användas inom varje `work-group`. Givet `NULL` delar OpenCL automatiskt upp `global_work_size` i godtyckliga mindre delar.
`num_events_in_wait_list` syftar på antalet händelser som ska ske innan denna exekvering påbörjas.
`event_wait_list` motsvarar de händelser, vars antal angavs i föregående argument, som ska ske innan exekveringen påbörjas.
`event` returnerar ett `event`-objekt som används för att identifiera den händelse som körs vid exekveringens tidpunkt.

Program

```
cl_program clCreateProgramWithSource (  cl_context context,
                                        cl_uint count,
                                        const char **strings,
```

```
                                        const size_t *length,
                                        cl_int *errcode_ret)
```

`clCreateProgramWithSource` returnerar ett körbart men ännu icke byggt `cl_program`-objekt med hjälp av en given källkodsvariabel.

`context` förväntar sig ett fungerande OpenCL `context`-objekt.
`count` motsvarar antalet avslutande icke-null karaktärer som finns i källkoden.
`strings` är en array av strängar som tillsammans utgör källkoden.
`length` motsvarar antalet tecken hos varje sträng i strings argumentet. Om length sätts till `NULL` antas varje sträng i strings avslutas med en null-brytning.
`errcode_ret` returnerar `CL_SUCCESS` om `cl_program`-objektet skapas utan problem.

```
cl_int clBuildProgram ( cl_program program,
                        cl_uint num_devices,
                        const cl_device_id *device_list,
                        const char *options,
                        void (CL_CALLBACK *pfn_notify),
                        void *user_data)
```

`clBuildProgram` returnerar `CL_SUCCESS` om programmet byggs utan problem.

`program` förväntar sig ett `cl_program`-objekt som ska byggas.
`num_devices` är antalet `devices` i `device_list`.
`device_list` är den lista med `devices` associerade till `program`. Om `device_list` sätts till `NULL` byggs programmet utan avseende på `device`-typ.
`options` argumentet är en pekare till en sträng innehållande programmets byggparametrar. Dessa finns att läsa mer om i kap. 5.6.3 i OpenCL handboken[8].
`pfn_notify` är den callback funktion som returnerar olika typer av felmeddelanden. Om denna är ekvivalent med `NULL` måste `clBuildProgram` köras klart innan den returnerar något. Är `pfn_notify` istället icke-ekvivalent med `NULL` kan `clBuildProgram` genast returnera ett svar.
`user_data` är ett arguments som används tillsammans med `pfn_notify` argumentet.

Context

```
cl_context clCreateContext (    const cl_context_properties *properties,
                                cl_uint num_devices,
                                const cl_device_id *devices
                                void (CL_CALLBACK *pfn_notify)
                                void *user_data
                                cl_int *errcode_ret)
```

`clCreateContext` returnerar en fungerande `context` om exekveringen utfördes utan problem, detta kan verifieras med hjälp av `errcode_ret`.

`properties` förväntar sig en lista med egenskaper som ska användas vid skapandet av en `context`. Listan avslutas med värdet `0`. Alternativt kan `NULL` för att användas som argument, för att använda fördefinierade värden.
`num_devices` motsvarar antalet `devices` i `devices`-argumentet.
`devices` är en pekare som pekar på en lista av tillgängliga `devices`.
`pft_notify` är en callback-funktion som används för att förmedla olika typer av felmeddelanden.
`user_data` är ett argument som används när `pft_notify` körs.
`errcode_ret` returnerar `CL_SUCCESS` om `clCreateContext` exekverades utan problem.

Command-Queue

```
cl_command_queue clCreateCommandQueue ( cl_context context,
                                        cl_device_id device,
                                        cl_command_queue_properties properties,
                                        cl_int *errcode_ret)
```

`clCreateCommandQueue` returnerar en fungerande `command-queue` vid korrekt exekvering.

`context` förväntar sig en fungerande `context` given vid exekvering av `clCreateContext`.
`device` måste vara en `device` som redan anslutits till den `context` som körs.
`properties` är den lista med egenskaper som `clCreateCommandQueue` ska skapas med. Genom att sätta värdet till `0` avslutas listan. `errcode_ret` returnerar `CL_SUCCESS` vid problemfri exekvering.

Buffer Objects

```
cl_mem clCreateBuffer ( cl_context context,
                        cl_mem_flags flags,
                        size_t size,
                        void *host_ptr,
                        cl_int *errcode_ret)
```

`clCreateBuffer` returnerar `CL_SUCCESS` om ett minnesobjekt skapats felfritt för en given `context`.

`context` representerar det `context` där man vill skapa ett minnesobjekt.
`flags` avgör vilken typ av objekt som skall skapas. Detta argument bör vara ekvivalent med exempelvis någon av följande värden:

- `CL_MEM_READ_WRITE` för att kunna läsa och skriva till objektet.

- `CL_MEM_READ_ONLY` för att enbart kunna läsa från objektet.
- `CL_MEM_WRITE_ONLY` för att enbart kunna skriva till objektet.

`size` förväntar sig storleken som skall allokeras, angiven i byte.
`host_ptr` är ett argument som används för andra, mindre vanliga flags än de tre angivna ovan.
`errcode_ret` returnerar ett felmeddelande om ett sådant existerar. Sätts argumentet till `NULL` skickas inga felmeddelanden ut.

```
cl_int clEnqueueWriteBuffer(    cl_command_queue command_queue,
                                cl_mem buffer,
                                cl_bool blocking_write,
                                size_t offset,
                                size_t cb,
                                const void *ptr
                                cl_uint num_events_in_wait_list,
                                const cl_event *event_wait_list,
                                cl_event *event)
```

`clEnqueueWriteBuffer` returnerar `CL_SUCCESS` givet att operationen utfördes felfritt.

`command_queue` är det argument inom vilket skrivningsoperationerna kommer att köas upp.
`buffer` förväntar sig ett fungerande `buffer`-objekt.
`blocking_write` förväntar sig antingen `CL_TRUE` eller `CL_FALSE` beroende på om operationen ska blockera andra operationer tills det att den själv är klar eller inte.
`offset` är den förskjutning som ska användas vid skrivning till `buffer`-objektet.
`cb` är data storleken angiven i bytes som förväntas överföras.
`ptr` är den pekare som används för att visa vart skrivningen ska ske.
`num_events_in_wait_list` är antalet händelser som måste ske innan denna operation utförs. Vilka dessa händelser är anges i nästa argument.
`event_wait_list` representerar vilka händelser som måste ske innan operationen utförs. Antalet händelser anges i det tidigare argumentet.
`event` används för att identifiera vilken skrivning alternativt läsning som sker under ett visst tillfälle.

```
cl_int clEnqueueReadBuffer(     cl_command_queue command_queue,
                                cl_mem buffer,
                                cl_bool blocking_read,
                                size_t offset,
```

```
                                    size_t cb,
                                    void *ptr,
                                    cl_uint num_events_in_wait_list,
                                    const cl_event *event_wait_list,
                                    cl_event *event)
```

`clEnqueueReadBuffer` returnerar `CL_SUCCESS` givet att operationen utfördes felfritt.

`command_queue` är det argument inom vilket läsningsoperationerna kommer att köas upp.
`buffer` förväntar sig ett fungerande `buffer`-objekt.
`blocking_read` förväntar sig antingen `CL_TRUE` eller `CL_FALSE` beroende på om operationen ska blockera andra operationer tills det att den själv är klar eller inte.
`offset` är den förskjutning som ska användas vid skrivning till `buffer`-objektet.
`cb` är data storleken angiven i bytes som förväntas överföras.
`ptr` är den pekare som används för att visa varifrån läsningen ska ske.
`num_events_in_wait_list` är antalet händelser som måste ske innan denna operation utförs. Vilka dessa händelser är anges i nästa argument.
`event_wait_list` representerar vilka händelser som måste ske innan operationen utförs. Antalet händelser anges i det tidigare argumentet.
`event` används för att identifiera vilken skrivning alternativt läsning som sker under ett visst tillfälle.

```
cl_int clSetKernelArg ( cl_kernel kernel,
                        cl_uint arg_index,
                        size_t arg_size,
                        const void *arg_value)
```

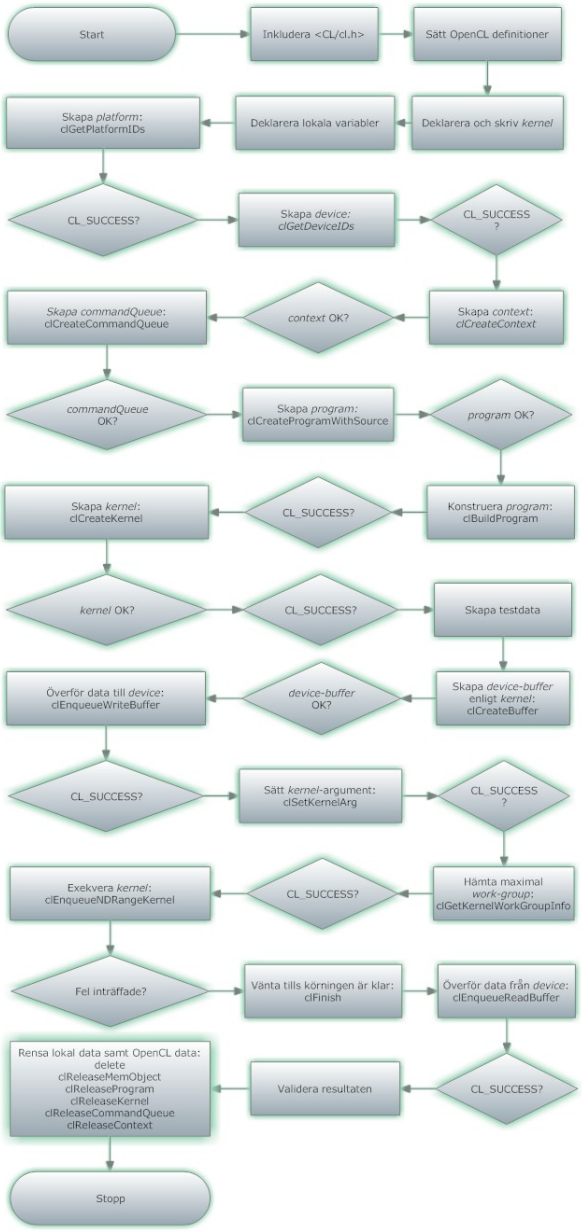
`clSetKernalArg` används för att sätta argumenten till en `kernel` och returnerar sedan `CL_SUCCESS` om operationen utfördes felfritt.

`kernel` förväntar sig den `kernel` som argumenten ska sättas till.
`arg_index` förväntar sig vilken position argumentet har i `kerneln`. Första argumenten har alltså `arg_index` ekvivalent med 0.
`arg_size` förväntar sig storleken på argumentet. Dvs stoleken på det minnesobjektet som ska användas.
`*arg_value` förväntar sig en pekare till den data som används som argument.

2.3.3 Körschema för programmering i OpenCL

Följande körschema visar ett sätt som kan användas för att skriva ett program i OpenCL, det är definitivt inte det ända sättet men det är ett av de möjliga sätten. Se nästa sida.

Genom att använda körschemat samt förklaringarna i 2.3.1 och funktionsdefinitionerna i 2.3.2 kan man enkelt skapa fungerande OpenCL program.



Figur 2.3: OpenCL - Körschema

Körschemat visar att man till en början måste inkludera `<CL/cl.h>`. Detta följs oftast av vissa definitioner som exempelvis `__NO_STD_STRING` eller `__NO_STD_VECTOR` för att istället använda OpenCLs egna string och vector.

När filerna har inkluderats och eventuella definitioner har satts, bör man skriva en kernel, antingen i en egen fil eller som en char array direkt i koden. Detta följs vanligtvis av deklarationer av de olika OpenCL-variablerna som kommer att användas.

Stegen där man skapar platform, device, context och så vidare behöver nödvändigtvis inte ske i ordningen enligt körschemat, dock så behöver vissa av objekten en specifik ordning då beroenden finns.

Steget som visar att testdata bör skapas är ett steg som självfallet kan göras tidigare i programmet men behövs innan stegen då datat överförs till enhetens minne.

Steget: ''Hämta maximal work-group": är ett steg som lämpar sig när det data som används är exempelvis en vektor. Vid data som har mer än en dimension kan det istället vara bättre att själv ange vilken storlek varje work-group ska ha. Notera att även en work-group kan ha flera dimensioner.

Efter dessa steg kommer man till själva körningen, följt av clFinish som väntar på att alla kommandon körs klart, clEnqueueReadBuffer som läser data från device samt valideringen och rensningen av gamla variabler.

Kapitel 3

Resurser

3.1 Hårdvara

3.1.1 GPU

I de två datorerna som används för körning används ett NVIDIA-grafikkort i ena och ett ATI-grafikkort i den andra.

NVIDIA GeForce GTX 560 TI (ASUS Fabriksklockad)
Klockhastighet: 900 MHz
Minnesklockhastighet: 4.2 GHz
Antal kärnor: 384[1]
Minnesstorlek: 1GB

Fler specifikationer kan hittas på NVIDIAs hemsida[9].

ATI RADEON HD5870
Klockhastighet: 850 MHz
Minnesklockhastighet: 1.2 GHz
Antal kärnor: 320[2]
Minnesstorlek: 1GB

Fler specifikationer kan hittas på AMDs hemsida[10].

[1] NVIDIA anger dessa som så kallade CUDA cores
[2] AMD anger 1600 Processing units. Varje kärna innehåller fem av dessa

3.1.2 CPU

Båda datorerna som testerna kör på har likadana Intel-processorer.

Intel Core i5 2500k
Klockhastighet: 3.3 GHz (max turbo: 3.7 GHz)
Antal kärnor: 4
Cachestorlek: 6 MB
L3 Cache latens: 8.18 ns

Detta är taget från Intels egna specifikationer[11] samt en artikel om latenser från AnandTech[12].

3.2 Mjukvara

Följande mjukvara används för att kunna köra ett projekt i OpenCL.

OpenCL
Som tidigare nämnt använder vi oss av OpenCL för att skriva program som kan exekveras på både GPU och på CPU. Detta kom som ett ganska naturligt val då CUDA är gjort för NVIDIA kort och då vi, enligt hårdvaruspecifikationerna, även använder ett ATI kort. Vi funderade över att använda JavaCL, en implementation av OpenCL för Java, men insåg då att dokumentationerna för JavaCL inte är i närheten så bra som de för OpenCL vilket i sin tur avgjorde det hela.

Microsoft Visual Studio 2010
Då C++ används som programmeringsspråk kändes Visual Studio som ett utmärkt val, då Visual Studio är ett stort och välarbetat utvecklingsverktyg med bra dokumentationer samt med stöd för OpenCL.

NVIDIA GPU Computing SDK och AMD APP SDK
För att kunna använda sig av OpenCL behövs ett *software development kit* som har stöd för OpenCL då man använder specifik hårdvara. NVIDIA GPU Computing SDK[13] låter oss köra OpenCL-program på NVIDIA-GPUer. För att kunna köra OpenCL-program på ATI-grafikkort använder vi oss av AMD APP SDK[14].

För att göra C++-projekten i Visual Studio kompatibla med OpenCL får man lägga till *additional include directories, additional library directories* och *additional dependencies* som alla finns tillgängliga i NVIDIAS, respektive ATIs, SDK.

Kapitel 4

Algoritmer

4.1 Testförsök 1: vectorSquare

Algoritmen 'vectorSquare' är en simpel algoritm som beräknar kvadraten av varje element i en vektor och returnerar resultatet.

Indata: En vektor med slumpade heltalsvärden.
Utdata: En vektor där alla element i indata har kvadrerats.

4.1.1 Utförande

En av de enklaste algoritmerna där man kan utnyttja parallellisering är vektorkvadrering. En vektor fylls på med slumpade värden. Varje element i vektorn kvadreras för att sedan returneras.

På CPU itererar man igenom varje position för att kvadrera varje element, ett i taget. På GPU kan man beräkna stora mängder data samtidigt och därför kvadrera flera element samtidigt. Det är alltså tack vare detta man i teorin borde kunna utföra denna beräkning snabbare på GPU.

Algoritmen utfördes med en nedre gräns på 2^{10} element samt en övre gräns på 2^{26} element där varje datastorlek kördes 10 gånger och ett medelvärde för följande operationer beräknades:

- Skrivning till beräkningsenhetens minne.
- Exekvering av beräkningsenheten.
- Läsning från beräkningsenheten minne.

I detta syftar beräkningsenheten på antingen en CPU eller en GPU.

4.1.2 Resultat

Resultaten för detta försök presenteras i tabellform i bilaga[A].

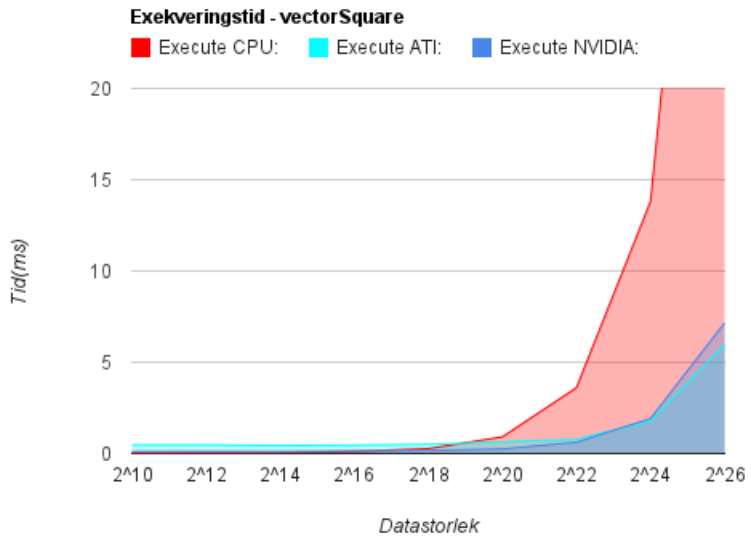
Resultaten visar att skrivning, exekvering och läsning av mindre datamängder till en CPU kunde utföras mycket snabbare än samma operationer till en GPU, men att allt eftersom att datamängderna utökades blev skillnaden mellan överföring till en CPU respektive en GPU mindre.

Vid en datastorlek på ca 2^{18} element var exekveringstiden lägre på en NVIDIA GPU jämfört med Intel Core i5 CPUn, skriv- och läsningstiderna gjorde dock fortfarande att den totala tiden på CPUn var lägre än den totala tiden för både ATI GPUn och NVIDIA GPUn.

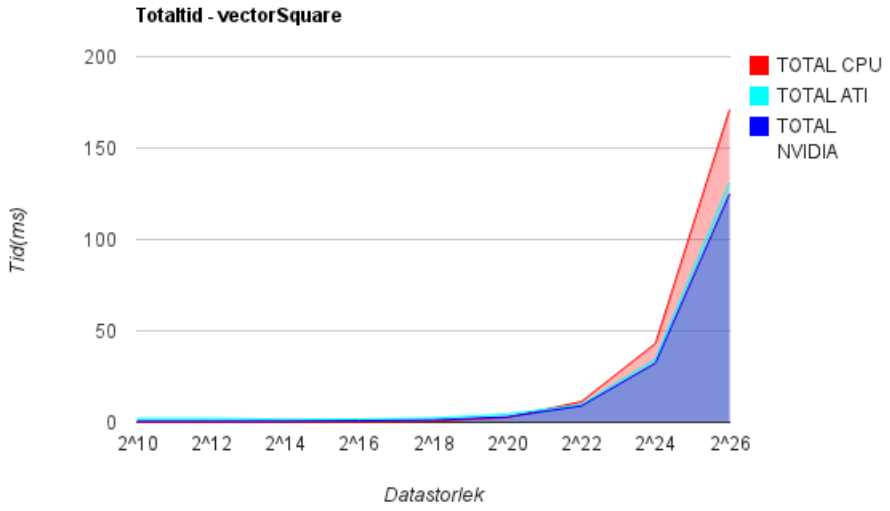
Vid 2^{20} element kan man enligt tidigare nämnd bilaga se att exekveringstiden med Intel Core i5 CPUn var längre än samma exekvering med ATI GPUn respektive NVIDIA GPUn men att den totala tiden även här, var lägre för CPUn.

Vid 2^{22} element ser man att den totala tiden för Intel Core i5 CPUn överstiger både ATI GPUn samt NVIDIA GPUn i snabbt stigande värden.

Den övre gränsen, enligt tidigare nämnd bilaga, visar en totaltidsskillnad på hela 39 ms mellan CPUn och den långsammare ATI GPUn.



Figur 4.1: En grafisk representation av exekveringstiderna för vectorSquare-algoritmen i bilaga[A]



Figur 4.2: En grafisk representation av totaltiderna för vectorSquare-algoritmen i bilaga[A]

4.2 Testförsök 2: dotProduct

Algoritmen 'dotProduct' är en lite mer komplicerad algoritm som beräknar dotprodukten av två vektorer med fyra element i vardera. För de som inte är bekanta är den matematiska formeln, om man använder vektorer av längd fyra:

$$(a, b, c, d) \cdot (s, t, u, v) = as + bt + cu + dv \tag{4.1}$$

Indata: Två vektorer av samma storlek.
Utdata: En resultatvektor som tar fyra element från vardera vektor i indata i taget, och räknar ut deras dotprodukt.

4.2.1 Utförande

En fortfarande simpel algoritm, men som är lite med komplicerad än vektorkvadrering är uträkningen av dotprodukt. Vad som gör den mer komplicerad är att man för varje resultat gör fler beräkningar.

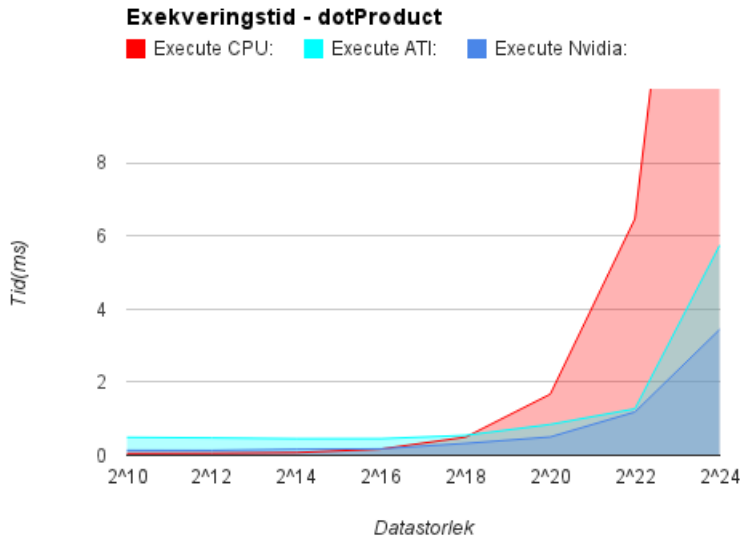
Denna algoritm kördes på CPU och GPU med antalet dotprodukter uträknade mellan 2^{10} och 2^{24}. Varje datastorlek kördes 10 gånger och ett medelvärde räknades ut för samma operationer som i föregående test: överföring till minne, exekvering och läsning från minne.

4.2.2 Resultat

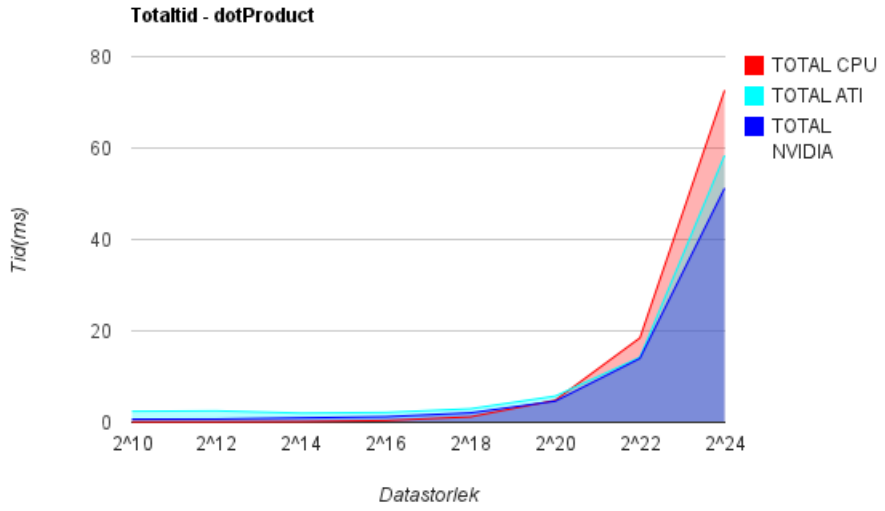
Precis som för förra algoritmen så tog skrivningen till minnet mycket längre tid för körning på GPU än CPU för små datamängder. På samma sätt minskade den tidsmässiga skillnaden för större datamängder. Vad det gäller exekveringen och läsningen är mönstret också väldigt liknande.

Vid uträkningen av totalt 2^{18} resultat kan man se att exekveringen börjar gå snabbare för NVIDIA GPUn. Detta blir även fallet för ATI GPUn vid 2^{20} resultat. Om vi tar en titt på den totala körningstiden så börjar NVIDIA GPUn bli snabbare vid 2^{20} resultat och AMD GPUn vid 2^{22}.

Allt detta kan hittas i bilaga[A]. En grafisk representation av exekveringstiderna samt totaltiderna visas nedan.



Figur 4.3: En grafisk representation av exekveringstiderna för dotProduct-algoritmen i bilaga[A]



Figur 4.4: En grafisk representation av totaltiderna för dotProduct-algoritmen i bilaga[A]

4.3 Huvudförsök: matrixMultiplication

Det finns många sätt att bättre utnyttja GPUns struktur än i de tidigare testförsöken. Vad de två testförsöken har gemensamt är att de båda använder sig av endimensionellt data i form av vektorer. Dessutom är tidskomplexiteten linjär för båda. En algoritm som inte är lika triviell och som dessutom utnyttjar GPUns förutsättningar på ett mycket bättre sätt är matrismultiplikation. Här hanterar vi tvådimensionellt data i form av matriser.

Vad matrismultiplikationen gör är att den för varje rad i den första matrisen räknar ut dotprodukten med denna rad, och en kolumn i den andra matrisen. I position (a, b) i svarsmatrisen kommer svaret från dotprodukten av rad a i första matrisen och kolumn b i den andra att stå. För att möjliggöra detta måste alltså bredden av den första överrensstämma med höjden av den andra. En matris av storlek NM som multipliceras med en matris av storlek MP kommer ge en svarsmatris av storlek NP. Om man undersöker tidskomplexiteten av detta kommer NMP operationer, vilket ger den $O(n^3)$.

4.3.1 Utförande

Denna algoritm är en algoritm som definitivt kan effektiviseras med hjälp av parallellisering. Då en CPU inom typisk programmering måste bearbeta ett element

i taget samt gå igenom tre stycken for-loopar för att få fram resultatet, kan man istället genom GPGPU direkt beräkna en relativt stor mängd element med hjälp av endast en inre loop.

De två yttersta looparna skulle normalt sätt använda den innersta loopen för att räkna ut värdet av dotprodukten för två vektorer, position för position i svarsmatrisen. Inom GPGPU kan man ignorera de två yttersta looparna och skriva kerneln på ett sådant sätt att den räknar ut så många positioner i svarsmatrisen som möjligt, samtidigt.

Kerneln som skrivits inom detta projekt ser ut på följande sätt:

```
__kernel void matrixMultiplication(     __global float* matrix_A,
                                        __global float* matrix_B,
                                        __global float* matrix_res,
                                        int matrix_A_width,
                                        int matrix_B_height){

        int index_one = get_global_id(0);
        int index_two = get_global_id(1);
        float current_res = 0.0;
        for ( int i = 0; i < matrix_A_width; i++ ){
                current_res =
                current_res+
                matrix_A[i + index_two * matrix_A_width]*
                matrix_B[i * matrix_B_height + index_one];
        }
        matrix_res[index_one + matrix_A_width * index_two] = current_res;
}
```

Vad denna *kernel* gör är att den som argument, tar emot två matriser, `matrix_A` och `matrix_B`, samt en tredje matris, `matrix_res`, som ska innehålla resultatet. *Kerneln* tar även emot argumenten `matrix_A_width` och `matrix_B_height` som används för att definiera hur stor matrisen ska bli.

Det första som händer i *kerneln* efter att argumenten har tagits emot är `index_one` = `get_global_id(0)` samt `index_two` = `get_global_id(1)`. Genom att göra detta kan man direkt få ut positionerna som ska användas inom den tvådimensionella rymden som matriserna utgör. Varje position i denna rymd kommer att skapa ett *work-item*.

Efter detta sätts en temporär variabel, `current_res` = `0`, och följs sedan av en for-loop som adderar varje multiplikation till denna, och därmed räknar ut dotprodukten. Varje *work-item* kommer att köra en for-loop. Dessa svarselement sparas sedan i `matrix_res` och finns därefter tillgängliga när det körande programmet

hämtar ut dessa med hjälp av kommandot `clEnqueueReadBuffer`.

Algoritmen utfördes med kvadratiska matriser med en nedre gräns på $2^0 \cdot 2^0$ element samt en övre gräns på $2^{11} \cdot 2^{11}$ element där varje datastorlek $(2^0 \cdot 2^0, 2^1 \cdot 2^1, ..., 2^{11} \cdot 2^{11})$ kördes 5 gånger. Detta för att medelvärden skulle kunna beräknas fram för de olika operationerna:

- Skrivning till minnet som tillhör beräkningsenheten.
- Exekvering av beräkningsenheten.
- Läsning från minnet som tillhör beräkningsenheten.
- Totaltid körtid.

Beräkningsenheten syftar då antingen på en CPU eller en av två GPUer, beroende på vilken *device* som användes.

Algoritmen utfördes med en `local_work-group_size` på $16 \cdot 16$ element. Notera även att `global_work-group_size` är datastorleken som användes vid respektive försök.

4.3.2 Antagande

Att göra ett perfekt antagande av tiden som det tar att exekvera denna algoritm på CPU respektive GPU är väldigt svårt. Man kan däremot börja med att göra ett naivt antagande där man enbart tar hänsyn till klockhastigheter och antalet kärnor. Även om dessa antaganden skiljer sig väldigt mycket från de riktiga resultaten borde den procentuella tidsskillnaden mellan GPU och CPU fortfarande vara rimlig.

Klockhastigheterna för både GPU och CPU är angivna i Hz, vilket anger hur många klockcykler som körs per sekund. Om man har X antal kärnor kan man approximera att det går att köra X cykler samtidigt. Matrismultiplikation använder sig av två aritmetiska operationer, addition och multiplikation. På Intels Sandy bridge-processorer har addition en latens på 1 klockcykel och multiplikation en typisk latens på 3 klockcykler[15].

Den naiva uppskattningen kan då bestämmas med formeln:

$$Tid(s) = \frac{antal\ klockcykler\ som\ behövs}{antal\ kärnor \cdot klockhastighet} \tag{4.2}$$

En bättre ansats är istället en som även tar hänsyn till skriv- och läshastigheter. För att göra detta är det viktigt att ta reda på vilken latens skrivningar och läsningar har till minnet, helst angivna i antal cykler. Då kan tiden approximeras bättre med formeln:

$$Tid(s) = \frac{antal\ klockcykler\ som\ behövs}{antalet\ kärnor \cdot klockhastighet} + \frac{minnesklockcykler\ som\ behövs}{antalet\ kärnor \cdot minneshastigheten} \tag{4.3}$$

CPU
Intel Core i5 2500k använder sig av 4 entrådiga kärnor. Klockhastigheten på dessa är 3.3 GHz. Detta borde innebära att den kan göra 3.3 miljarder cykler per sekund per kärna. Den typiska tiden det tar att komma åt L3-cache är 8.18 ns. Om detta räknas om till klockcykler i hastigheten 3.3 GHz motsvarar detta ca 27 cykler.

För en matrixmultiplikation av materiser med storlek MN och NP kommer kommer tiden att approximeras med formeln:

$$Tid(s) = \frac{27(MN + NP + MP) + (1 + 3)MNP}{4 \cdot 3.3GHz} \tag{4.4}$$

Där siffran 1 är antalet cykler som krävs för addition och siffran 3 för multiplikation.

GPU
De två grafikkort som används har följande klockhastigheter, minneshastigheter och antal kärnor:
GTX 560 TI: 900 MHz, 4.2 GHz, 384 kärnor.
Radeon HD5870: 850 MHz, 1.2 GHz, 320 kärnor.

Att komma åt minnet på NVIDIAs grafikkort tar enligt NVIDIA CUDA Programming Guide[16] 400 - 600 klockcykler. Låt oss använda värdet 500 anta att detta är fallet även för ATI. Om vi använder oss av samma latens för multiplikation och addition på GPUerna som för CPU får vi formlerna:

NVIDIA GEFORCE GTX 560 TI:

$$Tid(s) = \frac{(1 + 3)MNP}{384 \cdot 0.9GHz} + \frac{500(MN + NP + MP)}{384 \cdot 4.2GHz} \tag{4.5}$$

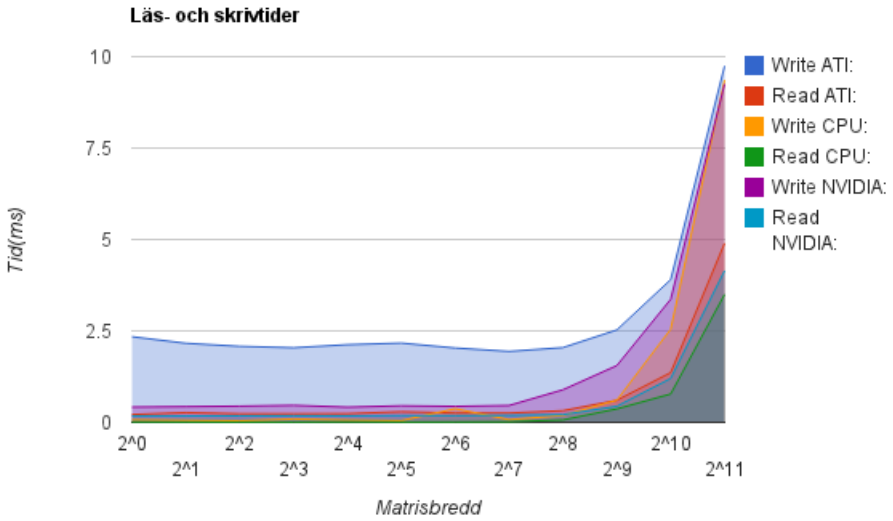
ATI RADEON HD5870:

$$Tid(s) = \frac{(1 + 3)MNP}{320 \cdot 0.85GHz} + \frac{500(MN + NP + MP)}{320 \cdot 1.2GHz} \tag{4.6}$$

Antaganden för både CPU och GPU är bara giltiga då M*N*P > antal kärnor eftersom vi inte kan dela upp en beräkning på flera kärnor.

4.3.3 Resultat

Resultaten i bilaga [B] visar att läsningen utan tvivel är snabbast på CPUn för alla försök. Efter denna kommer de två grafikkorten i ordningen NVIDIA följt av ATI. På samma sätt är CPU snabbast även för skrivningen, i princip alla fallen och följs av grafikkorten i samma ordning.

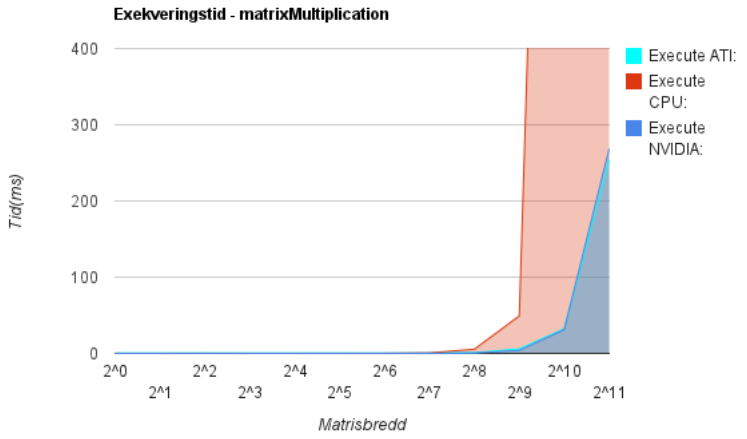


Figur 4.5: En grafisk representation av läs- och skrivtiderna för matrixMultiplication-algoritmen i bilaga[B]

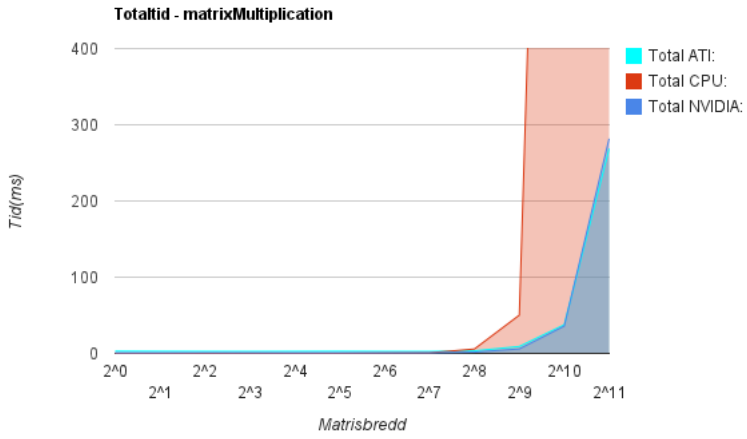
Precis som vid de två testförsöken är mindre datastorlekar mycket snabbare på CPU, men då datastorleken ökar, minskar den totala skillnaden, och vid en viss brytpunkt blir GPU snabbare.
Vid datastorleken $2^7 \cdot 2^7$ finns den första brytpunkten då de två GPU enheterna blir snabbare än CPUn. ATI-kortet är här ca 0.19ms snabbare än CPUn. NVIDIA-kortet är ca 0.47ms snabbare.

Vid $2^8 \cdot 2^8$ element finns den andra brytpunkten som gör att totaltiden för ATI GPUn samt NVIDIA GPUn är snabbare än CPUn, detta med ca 2.41ms respektive ca 4.12ms.

Då datastorleken på matrisen blir större än $2^9 \cdot 2^9$ kan man se att exekveringstiden för CPU stiger extremt. Exekveringstiden för $2^9 \cdot 2^9$ ligger på ca 50 ms, men då storleket ökar till $2^{10} \cdot 2^{10}$ tar exekveringen nästan 2 sekunder för att inte nämna vid storleken $2^{11} \cdot 2^{11}$ då det nästan tar en halv minut. Det här mönstret sker inte i närheten lika extremt på GPU då dessa storlekar får tiderna 6 ms, 32 ms och 254 ms på ATI-kortet respektive 4 ms, 31 ms och 268 ms på NVIDIA-kortet.



Figur 4.6: En grafisk representation av exekveringstiderna för matrixMultiplication-algoritmen i bilaga[B]



Figur 4.7: En grafisk representation av totaltiderna för matrixMultiplication-algoritmen i bilaga[B]

I föregående delkapitel presenterades tre antaganden för hur de olika tiderna bör se ut vid eventuella körningar inom denna algoritm. Följande tabell har sedan framställts:

Bredd*	Ant.** CPU	Tid CPU	Ant. NVIDIA	Tid NVIDIA	Ant. ATI	Tid ATI
2^4	0.003	0.0530938	0.0003	0.12124	0.001	0.396935
2^5	0.016	0.0733004	0.001	0.1299	0.004	0.403121
2^6	0.105	0.122992	0.007	0.1299	0.02	0.399162
2^7	0.736	0.643518	0.039	0.168457	0.095	0.435782
2^8	5.486	5.52093	0.255	0.553724	0.503	0.998246
2^9	42.28	49.2486	1.797	3.88565	2.998	5.99374
2^{10}	331.8	1947.59	13.4	31.2497	19.89	32.3806
2^{11}	2629	29871	103.3	268.439	142.7	253.99

*Bredd är den matrisbredd som används (höjden är densamma).
**Ant. står för antagande.
Alla tider angivna i millisekunder.

Kapitel 5

Diskussion

I detta kapitel kan man läsa om diskussioner relaterade till de två testförsöken samt huvudförsöket. Detta kapitel kommer även att behandla de förhållanden som algoritmerna utförts under samt eventuella felkällor.

Alla enheter som har använts har kört under samma förhållanden vad det gäller *local worksize* i huvudförsöket. Denna *local worksize* sattes till $16 \cdot 16$ eftersom testförsöken visade att ATI-kortet, som hade den minsta *workgroup*-storleken, kunde hantera 256 *workitems* samtidigt. Genom att använda denna *worksize* på alla enheter blir resultaten mer eller mindre rättvisa.

Enligt OpenCL kan både Intel-processorn och NVIDIA-kortet hantera 1024 *workitems* samtidigt. Detta är förståligt för NVIDIA-kortet, men för Intel-processorn antas detta ha någonting med virtuella kärnor eller någon form av hårdvaruacceleration att göra. NVIDIA-kortet får lite bättre resultat om man kör med en *local worksize* på $32 \cdot 32$, men däremot blir resultaten för Intel-processorn mycket sämre.

I resultatet av huvudalgoritmen noterade vi att körtiden för Intel-processorn ökade drastiskt efter matrisstorleken $2^9 \cdot 2^9$. Vi var till en början osäkra på vad detta kunde bero på, och började därför att fundera på vilka fördelar förutom klockhastigheten som CPU har över GPU, samt vilken fördel som vi kan ha förlorat. Efter lite undersökning av datastorlekar insåg vi att storleken som tre matriser av storlek $2^9 \cdot 2^9$ fyllda med floats tar på minnet är:

$$4Byte \cdot 2^9 \cdot 2^9 \cdot 3 = 3.146MByte \tag{5.1}$$

där 4 byte är storleken av en float. Storleken på minnet för nästa matrisstorlek blir på samma sätt:

$$4Byte \cdot 2^{10} \cdot 2^{10} \cdot 3 = 12.58MByte \tag{5.2}$$

Frågan är då vilken fördel som CPUn inte längre kan nyttja. Svaret är cache-minnet. I hårdvaruspecifikationen ser man att processorn har 6 MB cacheminne. Därför får den mindre datastorleken plats i cacheminnet, men när vi kör uträkningar av den större börjar vi få cachemissar vilket tar relativt lång tid att fixa.

Vid testkörningar noterade vi också att vi på grafikkortet inte kunde köra storlekar på $2^{13} \cdot 2^{13}$. Minnesåtgången som det här tar får plats på båda grafikkortens minne som är på 1GB vardera. Nästa storlek får däremot inte plats. För att lösa detta skulle man behöva dela upp matriserna för att sedan utföra beräkningarna på mindre matriser med hjälp av flera överföringar till och från enhetens minne.

Om vi nu tänker tillbaka på våra tidigare antaganden för olika datastorlekar med hänsyn till cacheminnet visar det sig att de är ganska rimliga.

Man ser i tabellen som finns i kap. 4.3.3 att de antaganden som gjorts för CPUn ser rimliga ut för datastorlekar i storleksordningen 2^6 till 2^9, detta antar vi bero på att vi i antagandena använde oss av latensen 8.18ns vilket enligt tidigare nämnda referens[12] är genomsnittslatensen för en L3 cache. Denna latens borde alltså vara rimlig.

Att antagandet inte ser bra ut för större datastorlekar beror, precis som vi har nämnt tidigare, på att man överstiger cachestorleken och därför måste börja ta hänsyn till fler latenser, som exempelvis läsning från primärminne.

För GPU ser man att antaganden från 2^8 ända upp till 2^{11} är i samma storleksordning som de egentliga tiderna. Eftersom allt data får plats på grafikkortens minnen behövdes aldrig några extra inläsningar till minnet som för de sista värdena hos CPU.

För de mindre tiderna, både för GPU och för CPU är tiderna längre än våra antaganden. Det här beror förmodligen på att man i OpenCL behöver köra någon form av startprocess, oavsett datamängd, som tar en viss tid, och mot de mindre tiderna blir dominant.

5.1 Felkällor

Efter flertal försök insåg vi att resultaten kunde variera från körning till körning och började därför att tänka på eventuella felkällor och hur man kunde lösa dessa.

Fragmenterat minne Att man innan eventuella försök utförde en omstart för att rensa primärminnet samt cacheminnet gjorde att resultaten väldigt ofta påverkades positivt.

Bakgrundsprogram Andra program körandes i bakgrunden använder konstant en del av beräkningskapaciteten. Dessa stängdes därför ner för att få mindre varierande resultat. Detta gäller då även exempelvis autostart program efter eventuella omstarter av datorer.

Konstant körning Vi noterade att exempelvis ATI-kortet inte använde sin fulla beräkningskapacitet om enheten inte behövde arbeta ett antal tidscykler först. Vi antar att detta är någon form av funktion som finns inprogrammerad i kortet för att inte belasta det mer än nödvändigt. För att lösa detta och få ut data med full kapacitet valde vi därför att automatisera testprocessen. Den automatiserade testprocessen beräknade varje datastorlek ett antal gånger direkt för att sedan gå vidare till nästa datastorlek och slutligen beräkna medelvärdet.

Kapitel 6

Slutsats

Under projektets gång har vi lärt oss mer och mer om GPGPU, speciellt om OpenCL. Vi har gjort upptäckter och fått resultat längst vägen som har inspirerat oss till att på egen hand fortsätta studera GPGPU. Dessa resultat har samtidigt besvarat frågeställningen som presenterades i inledningen.

Låt oss ta en titt på huvudfrågan:
Kan man med hjälp av en icke-triviell parallell algoritm visa att GPU är mer lämpad att utföra vissa beräkningar än CPU?

Våra resultat tyder på att detta faktiskt är möjligt. Vid stora mängder parallella beräkningar har vi, med hjälp av de tre algoritmerna, visat att GPUn faktiskt kan utprestera CPUn. Varför detta är möjligt är tack vare GPUns arkitektur, som tillåter den att köra mycket mer parallellt än CPUn, men med lägre hastigheter. Redan i bakgrunden nämnde vi saker som tyder på att detta borde vara fallet.

Om vi ser tillbaka på projektets gång inser vi hur mycket vi egentligen har lärt oss. Vi har lärt oss att använda en teknik som från början var helt främmande för oss, samt tagit del av fördelarna och nackdelarna som denna teknik har.

Vi tror att GPGPU är en teknik för framtiden, då Moores lag förmodligen inte kan gälla länge till. CPU-tekniken har gått från att hela tiden göra snabbare processorer till att fokusera mer på parallellisering. Även om CPUer parallelliseras mer och mer tror vi inte att dom kommer överstiga den nivå av parallellisering som motsvarande GPUer har vid den tidpunkten.

Kapitel 7

Ordlista och förkortningar

SIMD - Single Instruction Multiple Data - En term som inom datorarkitektur innebär att man med hjälp av en instruktion utför en operation på en mängd minnesobjekt samtidigt. Detta används vanligtvis inom parallella beräkningsenheter.

SISD - Single Instruction Single Data - En term som inom datorarkitektur innebär att man med hjälp av en instruktion utför en operation på ett minnesobjekt.

SDK - Software Development Kit - Utvecklingsverktyg som används specifikt för att kunna utveckla på en tillverkares hårdvara.

Litteraturförteckning

[1] *What's the difference between a CPU and a GPU?*
Kevin Krewell, NVIDIA Corporation
http://blogs.nvidia.com/2009/12/whats-the-difference-between-a-cpu-and-a-gpu/
16:e december 2009

[2] *Mapping computational concepts to GPUs*
Mark Harris, NVIDIA Corporation
http://dl.acm.org/citation.cfm?doid=1198555.1198768
2005

[3] *What is GPU Computing?*
NVIDIA Corporation
http://www.nvidia.com/object/GPU_Computing.html

[4] *Moore's Law Inspires Intel Innovation*
Intel Corporation
http://www.intel.com/content/www/us/en/silicon-innovations/moores-law-technology.html

[5] *GPGPU Programming*
GPGPU.org
http://gpgpu.org/developer

[6] *CUDA - Parallel Programming made easy*
NVIDIA Corporation
http://www.nvidia.com/object/cuda_home_new.html

[7] *NVIDIA - OpenCL*
NVIDIA Corporation
http://developer.nvidia.com/opencl

[8] *The OpenCL Specification*
Khronos OpenCL Working Group
http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf
6:e januari 2011

[9] *NVIDIA Geforce GTX 560 TI Specifications*
NVIDIA Corporation
http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-560ti/specifications

[10] *ATI RADEON HD5870 Specifications*
Advanced Micro Devices, Inc.
http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5870/Pages/ati-radeon-hd-5870-overview.aspx#2

[11] *Intel Core i5 2500k Specification*
Intel Corporation
http://ark.intel.com/products/52210

[12] *Jämförelse av cache-latenser på moderna CPUer*
AnandTech, Inc.
http://www.anandtech.com/show/4955/the-bulldozer-review-amd-fx8150-tested/6

[13] *NVIDIA GPU Computing SDK*
NVIDIA Corporation
http://developer.nvidia.com/gpu-computing-sdk

[14] *AMD APP SDK*
Advanced Micro Devices, Inc
http://developer.amd.com/sdks/AMDAPPSDK/downloads/Pages/default.aspx

[15] *Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPU*
Agner Fog, Copenhagen University College of Engineering
http://www.agner.org/optimize/instruction_tables.pdf
Senast uppdaterad: 2012-02-2

[16] *NVIDIA CUDA Programming Guide*
NVIDIA Corporation
http://www.clear.rice.edu/comp422/resources/cuda/NVIDIA_CUDA_ProgrammingGuide_2.3.pdf
Version 2.3
7:e jan 2009

Bildkällor:

[17] *ATI Stream Technology Logo*
Advanced Micro Devices, Inc.
http://developer.amd.com/gpu_assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf

[18] *NVIDIA CUDA Logo*
NVIDIA Corporation
http://www.nvidia.com/content/GTC/documents/1031_GTC09.pdf

[19] *Khronos OpenCL Logo*
Khronos OpenCL Working Group
http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[20] *Khronos OpenCL image of a platform*
Khronos OpenCL Working Group
http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

Figurer

2.1 a) ATI Stream[17]
b) NVIDIA CUDA[18]
c) OpenCL[19] 5
2.2 Grafisk framställning, platform[19] 7
2.3 OpenCL - Körschema 16

4.1 Exekveringstider, vectorSquare 22
4.2 Totaltider, vectorSquare 23
4.3 Exekveringstider, dotProduct 24
4.4 Totaltider, dotProduct 25
4.5 Läs- och skrivtider, matrixMultiplication 29
4.6 Exekveringstider, matrixMultiplication 31
4.7 Totaltider, matrixMultiplication 31