KTH Computer Science
and Communication

Igenkänning av instanslösningar i OpenCL

En undersökning av en plattformsoberoende parallellprogrammeringsimplementation

CARL BJÖRKMAN, MICHEL CUPURDIJA

Examenrapport vid CSC
Kursnamn: DD143X dkand11
Handledare: Mads Dam
Examinator: Mads Dam

Referat

På senare år har parallell beräkning blivit allt mer viktigt. En vanlig PC idag har en sammanlagd FLOPS-kapacitet som överstiger kapaciteten en ensam CPU-kärna i PC:n kan erbjuda.

OpenCL är ett standardiserat ramverk byggt för portabel parallellberäkning på heterogena system så länge enheterna i systemen har drivrutiner.

I denna rapport undersöker vi ramverket genom att göra en implementation som utför uttömmande sökningar för att hitta lösningar till instanser av det NP-fullständiga problemet verbalaritmetik.

Resultaten visar att denna typ av beräkning är olämpad för processeringsenheter som inte har stöd för effektiv random access, men är utmärkt för de som har detta stöd.

Abstract

In recent years parallel computing has become increasingly important. In a regular PC today lies often a FLOPS-capacity surpassing the capacity of a single core in the CPU by far.

OpenCL is a standardized framework that allows the programmer to write code able to execute on any computational unit having supporting drivers.

This thesis examines the framework by an implementation doing exhaustive searches in order to find solutions to instances of the NP-complete problem of verbal arithmetic.

The results imply that this kind of computation is unsuited for processing units without efficient support for random access, but on the other hand showing excellent performance on the ones that do.

Innehåll

1 Introduktion 3
1.1 Syfte 3
1.2 Problemformulering 3

2 Bakgrund 5
2.1 Problemets bakgrund 5
2.2 Teoretisk bakgrund 5
2.3 Teknisk bakgrund 7
2.3.1 Tekniska begränsningar i CPU-utvecklingen och behovet av parallellism 7
2.3.2 Utvecklingen och idén bakom OpenCL 8

3 OpenCL 10
3.1 Ramverket 10
3.1.1 Programflödet i korthet 10
3.1.2 Plattformsmodell 11
3.1.3 Exekveringsmodell 11
3.1.4 Minnesmodell 11
3.1.5 Programmeringsmodeller 12
3.1.6 Synkronisering 12
3.1.7 Minnesobjekt 12
3.2 OpenCL C 13

4 Hypotes och metod 14

5 Resultat 17
5.1 Implementationssvårighet 17
5.2 Prestanda vid olika konfigurationer 17

6 Diskussion och slutsats 21

Litteraturförteckning 24

Bilagor 25

A Fullständig implementation av verbalaritmetik-lösaren **26**

A.1 C++ 26

A.2 OpenCL 29

B Testkod **39**

B.1 Testkod-genererare 39

B.2 Testkod-analyserare 40

B.3 Testkörare 43

Akronymer

CPU	Central processing unit
DSP	Digital signal processor
FLOPS	Floating point operations per second
GPGPU	General purpose computing on graphics processing units
GPU	Graphical processing unit
PC	Personal computer
PRAM	Parallel Random Access Machine

Översättningar

Arbetsföremål	Work-item
Arbetsgrupp	Work-group
Arbetsparallell	Work parallel
Indexrymd	Index-space
ND-intervall	ND-range
OpenCL-enhet	OpenCL-device
Uppgiftsparallell	Task parallel
Värd	Host

Förord

Vi har delat upp arbetet enligt följande: Carl Björkman gjorde research om OpenCL, samt gjorde den faktiska implementationen av OpenCL-lösaren. Michel Cupurdija utförde tester och konstruerade C++-lösaren. Resten gjorde båda gemensamt.

Kapitel 1

Introduktion

I en värld där den möjliga klockfrekvensförbättringen av PC CPU:er är avtagande har det blivit populärt att producera fler kärnor för enheterna istället. Enligt Moores lag ska antalet transistorer i en dator fördubblas vartannat år. Lagen har visat sig stämma, men den säger ingenting om prestanda. Det finns fall där 45-procentiga ökningar i antalet processor-transistorer har lett till en ökning av blott 10-20 procent i processorkraft[1]. Dessa typer av resultat har gjort det alltmer lockande att satsa på parallell beräkning.

OpenCL är ett nytt ramverk utvecklat av Apple och standardiserat av Khronos Group[2]. Ramverket erbjuder programmeraren möjligheter att utföra beräkningar på en dators CPU, GPU och/eller DSP. Då FLOPS-kapaciteten hos dessa enheter sammanlagt överstiger kapaciteten hos en ensam kärna i en CPU, kan man vänta sig prestandavinster när man utnyttjar alla förfogade enheter med hjälp av OpenCL vid tillräckligt stora instanser av parallelliserbara problem.

1.1 Syfte

I denna uppsats undersöker vi hur mycket prestanda man kan vinna vid beräkningsintensiva probleminstanser och hur mycket tid som går åt till att göra implementationen i detta ramverk istället för en sekventiell implementation i språket C++. Som testproblem använder vi det matematiska spelet verbalaritmetik. Detta problem har visats vara NP-fullständigt[8] och instanser av problemet blir därav väldigt beräkningsintensiva att lösa med en uttömmande sökning. Uttömmande sökningar är väldigt enkla att parallellisera, vilket i vårt fall gör dem behändiga att använda för utvärdering. Mer om detta i kap 2.1 och kap 4.

1.2 Problemformulering

En vanlig programmerare som inte har erfarenhet av ramverket OpenCL kan finna det svårt att avgöra hur mycket programmeraren kan förbättra sitt programs prestanda genom att nyttja OpenCL. Om programmeraren bestämmer sig för att

försöka använda ramverket kan detta innebära en stor tidsinvestering. Skulle det visa sig att programmet inte når förväntat prestandaökning har programmeraren investerat tiden förgäves, vilket i vissa fall kan innebära ekonomiska konsekvenser.

I denna uppsats reder vi ut när det kan tänkas vara lämpligt att använda detta ramverk OpenCL och hur mycket prestanda man kan vinna vid beräkningsintensiva probleminstanser av rimlig storlek.

Kapitel 2

Bakgrund

I detta kapitel tar vi upp en kort, datalogisk analys av innebörden av flertal beräkningsenheter istället för en samt en teknisk sammanfattning av processeringsenheternas evolution .

2.1 Problemets bakgrund

Inom komplexitetsteorin delar man in problem i olika klasser beroende på vilka egenskaper dess lösningsalgoritmer har. Vi har studerat ett problem som ligger i klassen NP. Det innebär att man kan verifiera lösningen i polynomisk tid. Problemet är även *NP-svårt* vilket innebär att man kan reducera ett annat NP-svårt problem till det. Dessa två egenskaper gör att problemet klassificeras att vara *NP-fullständigt.*

Med undantag för små instanser tar NP-fullständiga problem lång tid att hitta lösningar till. Förutsatt att $P \neq NP$ tar det exponentiell tid att *känna igen* (hitta) en lösning till en instans. Många praktiska problem faller dock naturligt i denna kategori. Därav är det av stort intresse att på olika sätt få dessa lösningsigenkännanden att ta mindre tid. Det finns approximativa metoder, empiriska metoder med mera. Dessa är ofta komplicerade. Man kan låta pröva alla olika tänkbara lösningar och se om de passar, en så kallad *uttömmande sökning.* Inte helt oväntat tar detta väldigt lång tid. Men man kan reducera tiden om man kan dela upp de tänkbara lösningarna och låta en beräkningsenhet bearbeta vardera intervall av tänkbara lösningar.

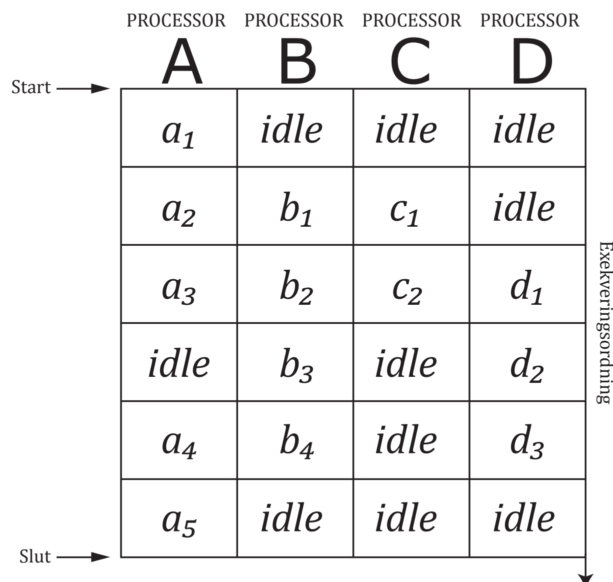
Vi har konstruerat en algoritm som utför parallelliserade, utömmande sökningar för att hitta instanslösningar till ett NP-fullständigt problem. Det är i huvudsak en praktisk undersökning. Det kommer att ge insikt i hur pass moget OpenCL är för att utnyttja teorin i nuläget.

2.2 Teoretisk bakgrund

När man analyserar parallella algoritmer brukar man använda en model som kallas *PRAM*, Parallel Random Access Machine. Denna modell är en teoretisk, ”ideal”

maskin som ignorerar problem relaterade till synkronisering och kommunikation. PRAM-modellen är oberoende av arkitektur. En annan viktig aspekt av PRAM är att man kan bidra med ett *problem-beroende antal processorer*, varav varje processor har ett lokalt minne samt tillgång till ett globalt minne (ett enda för hela maskinen). Skrivning och läsning till minnet medför ingen fördröjning i modellen. Alla dessa egenskaper och abstraktioner gör det lättare att analysera de logiska, algoritmiska och matematiska problemen med parallellisering.

Vi introducerar begreppet *tidssteg* för en algoritm och definierar det som ett asymptotiskt uttryck för det största antalet instruktioner som *måste* utföras i följd i algoritmen. Antalet tidssteg för ett problem blir alltså längden av den minimala *längsta kedjan av instruktioner* som beror av något tidigare resultat.



Figur 2.1. Exempel på en synkon, parallell algoritm fördelad på fyra processorer med någon specifik input.

I figur 2.1 illustreras en synkron, parallell algoritm, som för tydlighetens skull är för en specifik input (istället för generell input). Antag nu att denna algoritm är så uppdelad i parallella sekvenser som är möjligt för den givna inputen. T.ex. instruktionerna a_2, b_1 och c_1 beror av att instruktionen a_1 är utförd. b_3 beror av a_3, b_2, c_2 och d_1 etc. D.v.s inga fler processorer skulle kunna få exekveringen att slutföras i färre steg. Då är höjden på denna tabell antal tidssteg algoritmen tar för just den inputen eftersom att om det fanns en kortare kedja av instruktioner som kunde utföra samma sak skulle algoritmen inte vara så långt uppdelar i parallella sekvenser som möjligt för den givna inputen. Observera att det generella förhållandet mellan antal tidssteg och input alltid är givet som ett asymptotiskt uttryck.

Vi kan nu presentera följande sats för PRAM-modellen: [14]

Brent's sats. *En synkron, parallell algoritm A som tar t tidssteg och består av x instruktioner kan implementeras på p processorer i tid $O(x/p + t)$.*

Bevis. Låt x_i vara antalet instruktioner som utförs i det i:te exekveringssteget av A. Om man använder p processorer kan detta simuleras i tiden $O(x_i/p + 1)$. Därför kan A simuleras i $\sum_{i=1}^{t}(x_i/p + 1) = O((\sum_{i=1}^{t} x_i)/p + t) = O(x/p + t)$. □

Anledningen till varför man med p processorer kan simulera varje exekveringssteg av A i tiden $O(x_i/p + 1)$ är för att om t.ex. $O(x_k/p) = O(n)$ i steget k så har man bara tillräckligt med processorer för att i steg k utföra en $O(n)$:te-del av antalet instruktioner som skulle kunnat utföras i detta steg om man hade tillräckligt med processorer, d.v.s. $O(n)$ gånger fler processorer. Om däremot $x_k/p < 1$ så har man mättat antalet instruktioner man kan utföra i steg k med antal processorer, då det går åtminstone åt 1 steg till detta utförande av instruktioner.

Vad satsen egentligen säger är att den övre gränsen för en algoritm är antingen begränsad av antal tillgängliga processorer eller hur stor den största delen av algoritmen som inte kan parallelliseras är (d.v.s. hur många tidssteg algoritmen har).

Om t.ex. en algoritm går i $O(n)$ på en processor och har $\log n$ tidssteg är denna sagd att gå i $O(n/(n/\log n) + \log n) = O(\log n)$ på $n/\log n$ antal processorer enligt ovanstående sats.

Med ett NP-fullständigt problem kan vi med en uttömmande sökning förvänta oss en tidskomplexitet på $O(a^n)$ i det seriella fallet, där a är någon konstant. I det parallella fallet blir detta med PRAM-modellen $O(a^n/p + t)$ där t är antal tidssteg en instans av problemet tar. Så om vi har $O(a^n)$ antal processorer får vi en tidskomplexitet på $O(a^n/a^n + t) = O(t)$. Vi kommer använda detta faktum i vår hypotes.

2.3 Teknisk bakgrund

Följande sektion kommer att omfatta en kort översikt av hårdvaruutvecklingen och vad som gjort parallellism mer aktuellt på senare tid.

2.3.1 Tekniska begränsningar i CPU-utvecklingen och behovet av parallellism

Det är allmänt känt att klockfrekvensen hos en dator är en av de mest påvisande faktorerna för hur snabbt den kan utföra beräkningar. Det har således funnits ett stort intresse hos lekmän för utvecklingen av datorers klockfrekvenser.

Även om Moores lag angående antalet transistorer fortfarande håller, som tidigare nämnt, håller dock inte CPU:ernas klockfrekvens samma förbättringskurs.

De mest signifikanta fysiska anledningarna till detta är energiåtgång, värmeutveckling och strömläckage[3].

De två förstnämnda anledningarna hänger ihop. Mycket energi går åt till att stänga av och sätta på alla miljardtals transistorer varav denna process avger värme. Om man gör fler strömtillförselväxlingar per tidsenhet ökar också energibehovet respektive värmeutvecklingen.

Eftersom energiförbrukningen och värmeutvecklingen utgör två av de mest signifikanta flaskhalsarna faller det då naturligt att tala om CPU-effektivitet i termer av *flyttalsberäkningar per tidsenhet per effektenhet*.

I IEEE-publiceringen ”Optimizing Power Using Transformations” ([4]) lades det fram modeller som signifikant kunde reducera energiåtgången och behålla samma genomströmning. Denna publicering var den första som påvisade fördelar hos multiprocessering med avseende på energibesparing. [5]

En av de arktiekturella förändringarna som gav upphov till störst reduktion i energiåtgång innebar att på bekostnad av en ökad fördröjning i processering av en faktor 2 kunde man få ner spänningen med en faktor 1.7. En sido-effekt av förändringarna innebar dock att kapacitansen ökade med faktorn 2.2. [5]

Sambandet mellan effekt, kapacitans, spänning och klockfrekvens i en CPU lyder:

$$P = CV^2 f$$

Där P är genomsnittlig effekt, C är kapacitans, V är spänning och f är klockfrekvens. [4]

Om vi nu har två CPU:er med frekvensen $f/2$ vardera kommer vi få en spänning på $1/1.7V = 0.6V$ och en kapacitans på $2.2C$. Effekten blir då $2.2C * (0.6V)^2 * 1/2f = 2.2C * 0.36V^2 * 0.5f = 0.396P$. Alltså en klar besparing av strömtillförsel.

Det är alltså *CPU-effektivare* att ha fler kärnor med lägre klockfrekvens än en enda med en högre klockfrekvens. Detta möjliggör att man kan öka den summerade klockfrekvensen av samtliga CPU:er till en frekvens som hade bidragit till orimligt hög energiförbrukning och värmeutveckling i en ensam, enkelkärnig CPU.

Förbättringarna vi talar om är dock med avseende på den sammanlagda FLOPS-kapaciteten i CPU:n. Det innebär inte att exekveringen av ett program automatiskt kommer att utföras snabbare. För att förbättra exekveringstiden för ett program måste vi skriva programmet så att det kan utföra dess beräkningar parallellt på de olika kärnorna i CPU:n.

2.3.2 Utvecklingen och idén bakom OpenCL

Processortillverkarnas produktion av seriella CPU:er blev allt mer ansträngd under det tidiga 2000-talet och Intels beslut att avbryta utvecklingen av deras *Tejas*- och *Jayhawk*-processorer markerar något av slutet för denna era[6].

De allra första parallella beräkningsenheterna kan man återfinna redan på 1960-talet[7] och de har sedan dess haft en signifikant betydelse för superdatorer. För persondatorerna var det dock först i mitten av 2000-talet då parallella beräkningsenheter blev populära.

Datorspelsmarknaden hade också kraftigt drivit utvecklingen av grafikkort. Därav besitter moderna grafikkorts *GPU*:er (Graphical Processing Unit) en beräknings-

kapacitet jämförbar med motsvarande moderna CPU:er. 2006 lanserade ATI (numera AMD) ett interface kallat *"Close to Metal"* som tillät programmerarna att använda GPU:ns beräkningskapacitet till mer än bara grafiska beräkningar[9]. Denna typ av beräkning kallas *GPGPU* (General Purpose computing on Graphics Processing Units). Ett år efter att AMDs "Close to Metal" blev tillgänglig gjorde NVIDIA ett motsvarande GPGPU-interface för deras grafikkort tillgängligt kallat *"CUDA"* (Compute Unified Device Architecthure)[10].

Frånvaron av kompatibilitet mellan dessa två plattformar motiverade Apple att utveckla ett *plattformsoberoende*, *arkitekturoberoende* och *processeringsenhetstypoberoende* ramverk. Ramverket tillät programmeraren att utföra beräkningar på en enhet utan att behöva ta hänsyn till vilken plattform, vilken arkitektur eller ens vilken typ av processeringsenhet koden skall exekveras på. Detta ramverk är OpenCL.

När Apple var klara med utvecklingen presenterade de ramverket för *Khronos Group* i Juni, 2008. Khronos Group bildade en specialgrupp som sedan arbetade i fem månader på att standardisera OpenCL[11]. En allmänt tillgänglig specifikation blev klar i December 2008[12].

Kapitel 3

OpenCL

Följande kapitel behandlar själva ramverket OpenCL samt signifikanta komponenter. Det mesta i detta kapitel är en tolkning av specifikationen Khronos group har gett ut för OpenCL 1.1 [13]

3.1 Ramverket

Enligt den officiella specifikationen är OpenCLs målgrupp ”expertprogrammerare som vill skriva portabel, men effektiv kod”. Detta återspeglar sig i OpenCLs arkitektur. Den exponerar låg-nivå-interfaces till komponenter i en dator, samtidigt som den abstraherat det så att den färdiga koden behåller sitt oberoende.

Ramverket är till stora delar en abstraktion av heterogena beräkningsplattformar. Tanken är att programmeraren inte ska behöva tänka på exakt *vad* det är som bearbetar data, utan snarare *hur* den bearbetas.

Man kan dela upp OpenCLs abstraktioner i fyra modeller: En *plattformsmodell* – högnivå-beskrivning av det heterogena system man arbetar på, en *exekveringsmodell* – representation av exekveringsflödet på den heterogena plattformen, en *minnesmodell* – beskrivning av de olika minnestyper OpenCL arbetar med och hur de interagerar samt en *programmeringsmodell* – högnivå-abstraktion av hur programmeraren väljer att lösa parallelliseringen av problemet. Vi beskriver dessa modeller mer i detalj senare i detta kapitel.

3.1.1 Programflödet i korthet

Programmeraren har möjligheter att ”diagnostisera” plattformen programmet körs på genom att göra anrop till ramverket och få tillbaka information om t.ex. vilka beräkningsprocessorer som finns, vilken typ de har, hur många beräkningsenheter de har, vilken klockfrekvens de arbetar i med mera. Programmeraren kan sedan välja vilka beräkningsprocessorer som skall användas och låta kompilera ett program skrivet i språket *OpenCL C* (vi återkommer till detta språk senare) för denna processor. Sedan väljer man en *kernel* ur detta program. En kernel är en metod som

är markerad med en *qualifier* så att den sedan kan köas för ett anrop i värden (det kan finnas mer än en kernel per program). Programmeraren skapar en *kommandokö* till processorn, sätter argument till samtliga kernels och lägger dessa kernels i kommandokön. Efter detta kommer OpenCL att göra instanser av dessa kernels. När OpenCL är klar kan programmeraren hämta in resultaten.

3.1.2 Plattformsmodell

Plattformsmodellen består av en *värd* samt ett antal *OpenCL-enheter* på ett antal *plattformar*. Värdens uppgift är att skicka *kommandon* till de olika OpenCL-enheterna som i sin tur exekverar dem. Observera att i praktiken är det troligt att man enbart kommer att exekvera kommandon på en enda plattform (förmodligen en vanlig dator). Men man kan i teorin ha en värd som skickar ut kommandon till olika OpenCL-enheter på olika plattformar i ett kluster av beräkningsmaskiner. En OpenCL-enhet är i praktiken t.ex. en CPU eller en GPU i en dator, men det kan också vara något ovanligare som t.ex. en DSP, så länge det finns drivrutiner som stödjer ramverket.

3.1.3 Exekveringsmodell

Exekveringsmodellen består av *kernels* som exekverar på OpenCL-enheter och ett *värdprogram* som exekverar på *värden* och genererar dessa kernels. Värden är en enhet som centralt styr fördelningen av beräkningarna. När en kernel skickas från värden till en OpenCL-enhet skapas en *indexrymd* (som vi beskriver mer i detalj nedan). Instanser av denna kernel, kallade *arbetsföremål*, exekveras sedan för varje punkt i denna indexrymd.

Arbetsföremål är indelade i grupper kallade *arbetsgrupper*. Poängen med dessa är att man kan synkronisera lokalt i en arbetsgrupp genom att ha en funktion, kallad *barriär*, alla arbetsföremål i arbetsgruppen måste ha anropat innan något av arbetsföremålen får exekvera vidare.

Indexrymden i OpenCL kallas *ND-intervall*. Ett ND-intervall har N dimensioner, $N = \{1, 2, 3\}$. Varje arbetsföremål i ND-intervallet har ett *lokalt ID*, ett *globalt ID* och ett *arbetsgrupp-ID*. Om $D = 2$ är förhållandet mellan dessa ID:n följande:

$$(g_x, g_y) = (w_x * S_x + s_x + F_x, w_y * S_y + F_y)$$

där g_i är det globala ID:t, s_i är det lokala ID:t, w_i är arbetsgrupp-ID:t, S_i är storleken på arbetsgrupperna (den måste vara homogen för sin dimension) och F_i är en offset som definierar var ND-intervallet börjar. Förhållandet om $N = 1$ eller $N = 3$ är analogt.

3.1.4 Minnesmodell

I minnesmodellen definierar man fyra olika typer av minnen vars egenskaper skiljer sig i vad värden respektive ett arbetsföremål kan och inte kan göra. De fyra typerna

	Global	Konstant	Lokal	Privat
Värd	Dynamisk allokering Läs/skriv	Dynamisk allokering Läs/skriv	Dynamisk allokering -	Ingen allokering -
Kernel	Ingen allokering Läs/skriv	Statisk allokering Läs	Statisk allokering Läs/skriv	Statisk allokering Läs/skriv

Tabell 3.1. Minnesmodellerna i OpenCL

är *globalt minne*, *konstant minne*, *lokalt minne* samt *privat minne*. Deras egenskaper kan sammanfattas i följande tabell:

3.1.5 Programmeringsmodeller

Det finns två programmeringsmodeller OpenCL stödjer: en *dataparallell* och en *arbetsparallell* modell. OpenCL stödjer också kombinationer av dem.

I en *dataparallell* modell ser man programflödet som att ett visst antal instruktioner ska appliceras på viss data lagrat i något minne. I en strikt dataparallell model ska det egentligen vara en 1-till-1-förhållande mellan varje grupp minne resp. varje uppsättning instruktioner. Men i OpenCL är det inte ett krav. Denna modell var den huvudsakliga modellen man utgick ifrån när man utvecklade ramverket.

I en *arbetsparallell* modell ser man en uppsättning instruktioner som är oberoende av sin position i indexrymden. Varje arbetsföremål har en egen uppgift som inte behöver likna de andra arbetsföremålens uppgifter. Det blir därför i denna modell svårare att avgöra hur lång tid respektive arbetsföremål kommer att bearbetas och därav svårare att ”balansera” dem tidsmässigt.

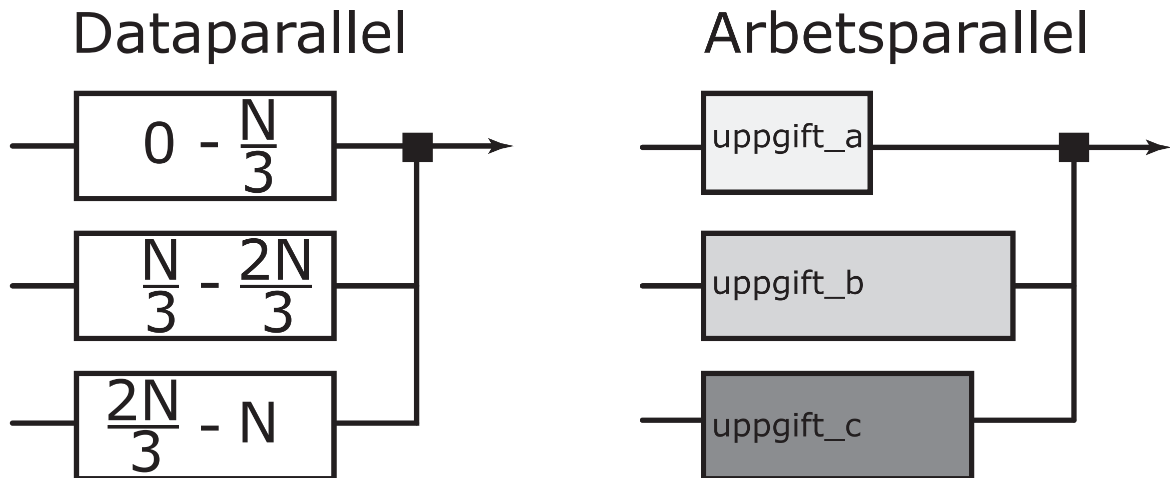
Se figur 3.1 för illustration.

3.1.6 Synkronisering

Synkronisering i OpenCL kan ske inom en arbetsgrupp med en *barriär*, som tidigare nämnt. Man kan också synkronisera *kommandon* i en kommando-kö genom att ha en kommando-kö-barriär eller att invänta *events* från OpenCL. Varje anrop till OpenCL-API:et returnerar ett event man kan invänta.

3.1.7 Minnesobjekt

Minnesobjekt i OpenCL är en abstrakt representation för portioner av minne värden och dess kernels har tillgång till. Minnesobjekt kan delas upp i två kategorier:



Figur 3.1. Illustration av den data- respektive arbetsparallella programmeringsmodellen

bufferobjekt och *imageobjekt*. Skillnaden är att ett *bufferobjekt* har minne lagrat sekvensiellt och är åtkomligt med vanliga pekare medan *imageobjekt* är lagrade i ett format som är "obskyrt" för värden resp kerneln. Ett imageobjekt kommer man istället åt med inbyggda funktioner i OpenCL C.

3.2 OpenCL C

När man skriver kernels som skall exekveras på OpenCL-enheterna skriver man i ett språk kallat *OpenCL C*. OpenCL C är baserat på C-språkets specifikation (ISO/IEC 9899:1999) men med ett antal modifikationer och begränsningar.

Bland modifikationerna finner man mycket lånat ifrån GLSL, OpenGL Shading Language. Bland annat stöd för vektorer där man kan välja komponenter ur en vekor med punktoperatorn som alternativ till index offset. Det finns många inbyggda typer som vec$n$ där $n \in \{2, 3, 4\}$ och bool. Ett annat tillägg är *qualifiers*. Dessa anges dels i samband med parametertyper och dels beskriver de vilken minnestyp respektive parameter är (se minnesmodeller i kap 3.1) och dels för att visa om en metod är en kernel eller en vanlig metod åtkomlig av andra kernels. Språket har ett stort antal inbyggda funktioner. Däribland arbetsföremål-relaterade funktioner (för att få information om en instans olika ID:n) och en omfattande samling matematiska funktioner.

Begränsningarna omfattar saknat stöd för explicita casts (konvertering måste ske via anrop till konstruktorer som konverterar en typ till en annan) undantaget konvertering mellan ints och floats. Det mesta av C:s standardbibliotek är oanvändbart. Funktionspekare är otillåtna. Bit-fields är otillåtna. Rekursion är otillåtet.

Kapitel 4

Hypotes och metod

Vårt syfte är att använda OpenCL för att hitta lösningar till instanser av svåra problem.

Vi konstaterar först att det finns signifikanta, strategiska metoder man kan använda för att reducera körtiden för program. Den *mest* signifikanta metoden är att använda algoritmer som är så effektiva som möjligt. Observera att det finns avancerade tillvägagångssätt för att hitta lösningar till instanser av NP-fullständiga problem, men för att underlätta parallellisering kommer vi att utföra uttömmande sökningar. De andra signifikanta metoderna för att reducera körtiden är att se om man kan förenkla problemet givet dess definition (det kan man dock sällan med NP-svåra problem) samt eventuella ”råoptimeringar” (som t.ex. att göra en stor allokering i början istället för flera små allokeringar under körningen).

Har man utfört dessa metoder för körtidsreduktion och fortfarande vill förbättra sin prestanda kan man överväga att utnyttja mer än en processeringsenhet (om maskinen har mer än en). Vi kommer att undersöka om man kan reducera körtiden i OpenCL med ett NP-fullständigt attrapproblem. Vårt val av problem är, som tidigare nämnt, *verbalaritmetik*. Vi ger en kort definition av problemet:

Verbalaritmetik:
Givet ett ord A, ett ord B och ett ord C. Finns det någon injektiv funktion f : (bokstav $\rightarrow$ siffra) så att $f(A) + f(B) = f(C)$ där $f(A_0)$, $f(B_0)$, $f(C_0) \neq 0$?

Vi analyserar tidskomplexiteten för detta problem. Låt n vara antalet unika bokstäver i instansen, m den högsta ordlängden och b vara den matematiska basen. För varje tilldelning av en siffra till en unik bokstav måste det ske en tilldelning av resterande unika bokstäver innan en evaluering av huruvida tilldelningarna ger en lösning är möjlig. Evalueringen går i $O(m^2)$ då man i värsta fall har enbart unika bokstäver i respektive ord och för varje bokstav måste man i värsta fall beräkna vad b^m är. Om vi räknar multiplikation som enhetskostnad tar det $O(m)$ att räkna ut b^m eftersom $b^m = \prod_{i=1}^{m} b$. Alltså tar evalueringen $O(m) * O(m) = O(m^2)$ i tid.

Vi får då att

$$t(n) \leq \begin{cases} m^2 & n = 0 \\ b * t(n-1) & n \geq 1 \end{cases}$$

vilket ger

$$t(n) = O(b^n m^2)$$

Vi har dock en förenkling i vår problemdefinition. Eftersom det gäller att funktionen f ska vara *injektiv* är det värdelöst att t.ex. evaluera en tilldelning till siffran s om det redan gjorts en tilldelning till siffran s. Detta påverkar tidskomplexiteten. Vi kan se det som ett rekursionsträd som avsmalnar med en gren per rekursionsnivå och djupet på trädet blir antalet unika bokstäver. Vi får då att tidskomplexiteten blir:

$$t(n,b) \leq \begin{cases} m^2 & n * b = 0 \\ b * t(n-1, b-1) & n, b \geq 1 \end{cases}$$

och då $\prod_{i=b}^{b-n+1} i = \frac{\prod_{i=b}^{b-n+1} i * \prod_{i=b-n}^{1} i}{\prod_{i=b-n}^{1} i} = \frac{\prod_{i=b}^{1} i}{\prod_{i=b-n}^{1} i} = \frac{b!}{(b-n)!}$ ger detta

$$t(n,b) \leq \begin{cases} m^2 & n = 0 \\ \frac{b!m^2}{(b-n)!} & 0 < n \leq b \\ b!m^2 & n > b \end{cases}$$

alltså betydligt bättre än tidskomplexiteten för problemet utan hänsyn till nämnd förenkling. Observera att om $n > b$ har instansen ingen lösning då det inte finns tillräckligt med unika siffror för att matcha de unika bokstäverna.

Det faller naturligt att göra en uttömmande sökning med rekursion. Dessvärre stödjer inte OpenCL C rekursion. Vi måste därför implementera ”syntetisk rekursion” med hjälp av en omfattande loop samt en stack som håller nuvarande avbildning för respektive in-värde. Tanken är att vi kan parallellisera problemet genom att låta de första k (fixt värde) antal avbildningarna i f vara statiska för resp arbetsföremål. T.ex. om $k = 2$ kommer vi att ha b^2 antal arbetsföremål. Om då $b = 10$ får vi 100 arbetsföremål varav den första har att $f(X_0) = 0, f(X_1) = 0$, andra kommer att ha $f(X_0) = 0, f(X_1) = 1$, tionde arbetsföremålet kommer ha $f(X_0) = 1, f(X_1) = 0$ och så vidare, där $X = A \cup B \cup C$. Observera att om $n > b$ kommer det första arbetsföremålet på en gång konstatera att den inte kan ge en lösning då definitionen av verbalaritmetik kräver att f är injektiv.

En annan restriktion, som egentligen inte kommer från OpenCL, är något som kallas ”NVIDIA Watchdog”. Detta är en timeout som inte låter någon process ockupera ett NVIDIA grafikkort med beräkningar i mer än 5 sekunder. Detta gäller alltså enbart om man använder OpenCL på NVIDIA-grafikkort. Men det ökar kravet på att beräkningarna måste vara väl fördelade på de olika arbetsföremålen snarare än att låta ett enskilt arbetsföremål beräkna mycket.

Vår hypotes är att en probleminstans av tillräckligt hög storlek kommer att kunna lösas snabbare med OpenCL på en dator med ett tillräckligt kraftfullt grafikkort

än en vanlig C++-implementation kan lösa samma instans. Enligt PRAM-modellen (se kapitel 2.2) har vi i detta problem att $t = \theta(m^2)$, då evalueringen är det enda som inte går att parallellisera. Om vi har $O(b!)$ antal processorer (vilket vi kan enligt resonemang i PRAM-modellen) får vi då en tidskomplexitet på $O(m^2)$.

Vi implementerar en lösare i ren C++ och en i C++ med OpenCL. Vi låter sedan genera testfall, kör dessa med olika konfigurationer och låter analysera resultaten.

Kapitel 5

Resultat

5.1 Implementationssvårighet

Vi hade båda förkunskaper i C/C++, men ingen alls i OpenCL. Det tog oss ca 8 timmar att utföra implementationen i C++ och få den buggfri. Vi spenderade ca 20 timmar på att läsa på om OpenCL inför implementationen och själva implementationen tog ca 80 timmar. Det finns ingen debugger till OpenCL och det blev därav mycket svårt att undersöka programmets beteende under körtid. Detta blev speciellt problematiskt när man började köra fler arbetsföremål än ett åt gången. Det gick dock att göra primitiva spårutskrifter till minne man sedan kunde läsa av för att kunna få en uppfattning om det ungefärliga programflödet.

5.2 Prestanda vid olika konfigurationer

Samtliga tester utfördes på en Mac Pro (MA970*/A) med två 2.8 GHz (E5462) Quad-Core Intel Xeon ”Harpertown”-CPU:er och ett NVIDIA GeForce GTX 285 grafikkort. CPU:erna har 8 kärnor sammanlagt och 12 MB L2-cache per CPU (24 MB sammanlagt). GPU:n har 30 kärnor och ingen cache.

Vi körde fem olika tester med bas 10, 11, 12, 13 resp. 14. Probleminstanserna var genererade enligt följande: Vi lät generera ett slumptal tal A_{10} och ett slumptal B_{10} och sätter ett tredje tal C_{10} till $A + B$. Sedan låter vi probleminstansen vara $A_b + B_b = C_b$, där b är testets bas. Varje test omfattade 100 instanser. I ett test lät vi varje instans lösas med följande olika konfigurationer:

C++ Seriell beräkning på CPU. Implemention i ren C++

GPU2 Parallell beräkning på GPU med 2 statiska funktionsavbilder för respektive arbetsföremål

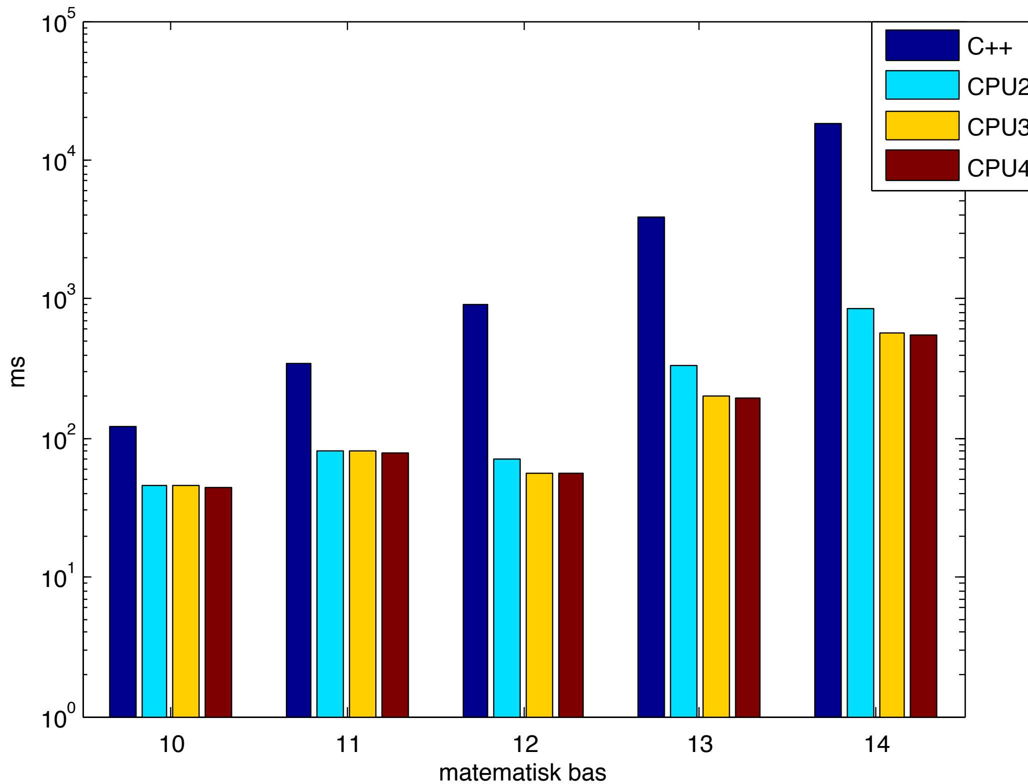
GPU3 Parallell beräkning på GPU med 3 statiska funktionsavbilder för respektive arbetsföremål

GPU4 Parallell beräkning på GPU med 4 statiska funktionsavbilder för respektive arbetsföremål

CPU2 Parallell beräkning på CPU med 2 statiska funktionsavbilder för respektive arbetsföremål

CPU3 Parallell beräkning på CPU med 3 statiska funktionsavbilder för respektive arbetsföremål

CPU4 Parallell beräkning på CPU med 4 statiska funktionsavbilder för respektive arbetsföremål



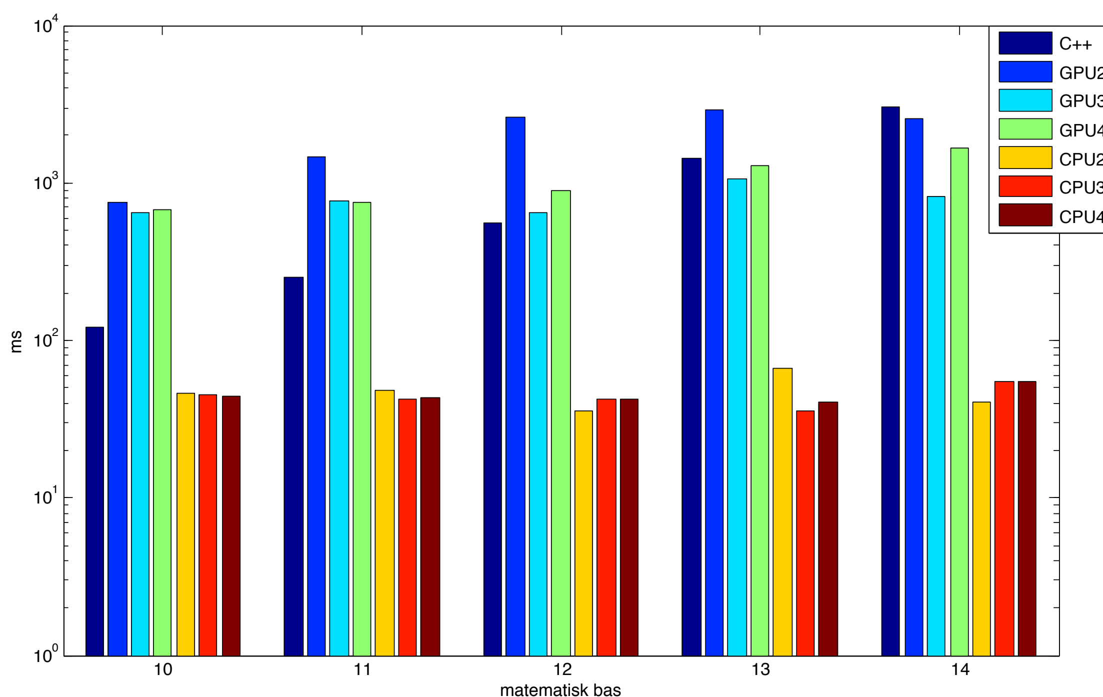
Figur 5.1. Genomsnittshastighet för respektive konfiguration i förhållande till bas.

Figur 5.1 visar genomsnittliga körtiden för respektive konfiguration i respektive bas. Vi har inte tagit med data för GPU-körningar då dessa fått timeout vid ett flertal tillfällen och får då orättfärdigt lägre genomsnittskörtid om man tar hänsyn till samtliga instanser. Vi noterar att **C++** är långsammast oavsett bas, men att avståndet ökar med basen. Vi noterar även att de parallella körningarna var genomsnittligt snabbare i bas 12 än i bas 11.

Figur 5.2 visar genomsnittshastigheten för olika konfigurationer i respektive bas. I denna figur har vi enbart använt oss av data från instanser ingen GPU-körning fått timeout på. Vi noterar att **C++** är den enda konfigurationen som ökar entydligt exponentiellt i förhållande till basen. Vi noterar att de parallella körningarna på CPU är snabbare än de seriella vid högre baser (när de väl hittar en lösning innan timeout). Vi noterar att de parallella körningarna på CPU alltid är överlägset snabbast och att de till synes inte ökar i förhållande till basen.

Figur 5.3 visar antalet gånger respektive konfiguration har haft lägst körtid i någon instans. Vi noterar att **C++** och **GP4** aldrig är snabbast. Vi noterar att noterar att en parallell körning på GPU faktiskt kan vara snabbast. Vi kan inte utläsa något förklarligt mönster bland **CPU2**, **CPU3** och **CPU4**.

Figur 5.4 visar den högsta tiden uppmätt för någon instans för respektive kon-



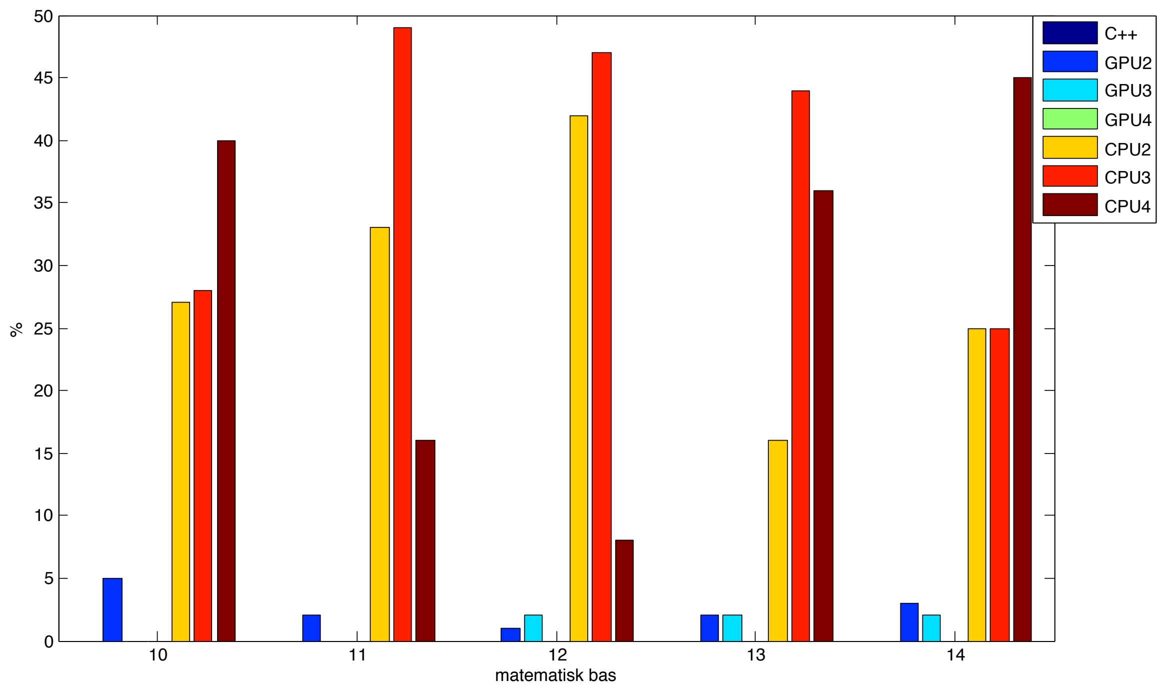
Figur 5.2. Genomsnitthastighet för respektive konfiguration i förhållande till bas. Enbart instanser som alla konfigurationer klarade att lösa innan en timeout är medräknade.

figuration. Vi noterar att **C++** ökar exponentiellt. OpenCL-implementationerna ökar ”nästan” exponentiellt. Vi noterar att hoppen från en icke-primtal-bas till en primtal-bas utgör de mest signifikanta ökningarna i tid för OpenCL-implementationerna.

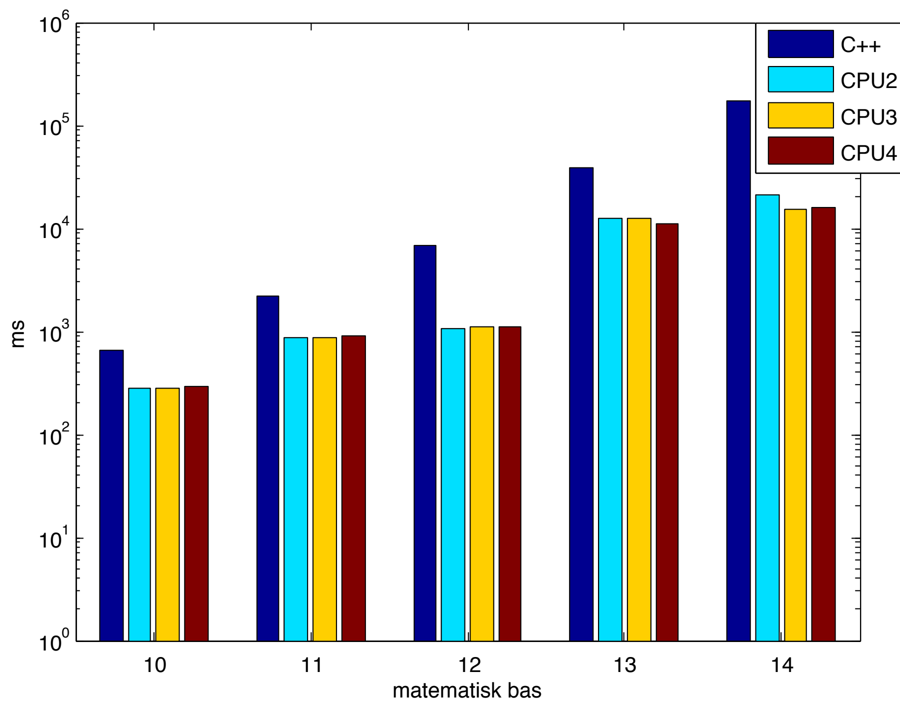
Tabell 5.1 visar hur stor andel av instanserna som fick timeout (p.g.a. NVIDIA watchdog) för respektive konfiguration i respektive bas. Vi noterar att högre bas generellt ger högre timout-rate och att lägre antal statiska funktionsavbildningar också generellt ger högre timeout-rate. Sistnämnda egenskap är dock inte entydlig för **GPU3** och **GPU4** vid bas 13 respektive 14.

	GPU2	GPU3	GPU4
Bas 10	0	0	0
Bas 11	4	2	4
Bas 12	15	2	1
Bas 13	38	13	10
Bas 14	53	28	37

Tabell 5.1. Timeout-rate för GPU-körningar i procent.



Figur 5.3. Antal gånger respektive konfiguration var snabbast i någon instans.



Figur 5.4. Högsta tiden uppmätt för någon instans för respektive konfiguration.

Kapitel 6

Diskussion och slutsats

FLOPS-kapaciteten i ett modern grafikkort överstiger i många fall kapaciteten de vanliga CPU:er erbjuder. Man skulle kunna tänka sig att nästan allt som går att parallellisera kan få någon slags prestandaökning om den utnyttjar grafikkortet till fullo. I vårt fall hade vi ett problem som var enkelt att parallellisera och, ur ett teoretiskt perspektiv, vinner stort på en parallellisering.

När man beskådar våra resultat ser man snabbt att det inte entydligt är så i verkligheten. GPU-körningarna var i många fall långsamma och dessutom opålitliga på grund av timeout-risken. Att körningarna var långsamma kan inte bero på OpenCL eftersom de parallelliserade CPU-körningarna var betydligt snabbare än den seriella implementationen i samtliga fall.

Gapet i hastighet mellan den seriella körningen och de parallella CPU-körningarna blir allt mer påtaglig ju högre bas vi har (se figur 5.1). Man kan ana att en större uppdelning av problemet (fler statiska avbildningar) lönar sig mer ju större instans som beräknas. Detta fenomen har vi ingen direkt förklaring till. Det är inte omöjligt att det finns en matematisk anledning. Detta är dock ingenting vi kan undersöka närmare då vi har begränsad tid.

Vi hade heller inte tid att pröva större baser, men man kan tänka sig att trenden kan fortsätta ytterligare efter bas 14. Detta påvisar att man kan tjäna mycket på att parallellisera stora instanser för CPU. Frågan uppstår då huruvida man även kan tjäna på att parallellisera för GPU, vilket var vår originalhypotes.

Man kan försiktigt analysera vad vår data i figur 5.2 påvisar. Trots att de ”tyngsta” instanserna är bortskalade i denna data kan man fortfarande se att de seriella körningarna uppvisar en tidsmässigt exponentiell trend i förhållande till basen. GPU-körningarna verkar dock mer tids-resistenta till bas-stegningen i dessa storlekar av probleminstanser. Man kan då inte utesluta att GPU-körningarna faktiskt skulle kunna vara snabbare än de seriella med en tillräckligt hög bas om man kunde förhindra time-out. Observera att det är omöjligt att få ut någon prestandavinst ur GPU-körningar om man inte går runt detta problem, då antalet timeouts snabbt växer med instansens storlek (se tabell 5.1).

I figur 5.3 kan man se att GPU-körningarna i vissa fall faktiskt var snabbare

än de parallella CPU-körningarna. Det är dock förmodligen enbart för att det finns så många GPU-kärnor tillgängliga. Vissa instanser kan sedan ha lösningar som involverar de statiska avbildningar någon GPU-kärna fått tilldelad redan i början av sökningen samt att resten av avbildningarna består av låga värden och kommer därför att ha evaluerats tidigt.

Den mest troliga förklaringen till varför GPU:n underpresterar CPU:n är *avsaknaden av cache* hos GPU:n. Vad GPU:er är bra på är sekvensiell inläsning av data. Ett grafikkort har oftast mycket hög bandbredd mellan minne och GPU-kärna[15] vilket gör att den lämpar sig för t.ex. addition av mycket stora vektorer. Vad den däremot *inte* är bra på är *random access*. Vår lösare använder tyvärr nästan uteslutande random access vilket kan förklara den relativt dåliga prestandan.

Problempartitionerna (arbetsföremålen) i den parallelliserade algoritmen använder dock lite data (den har linjär minneskomplexitet) och har därav *bra lokalitet*. Detta gör att bearbetning med CPU är väldigt effektiv då den väl kan utnyttja CPU:ns cache. Detta implicerar att uttömmande sökningar med fördel kan parallelliseras och köras på CPU:er, dock inte cache-lösa GPU:er.

Ett fenomen man kan observera i figur 5.1 och figur 5.4 är att prestandan nästintill är opåverkad för de parallela CPU-körningarna i stegningen från bas 11 till bas 12. Det är svårt att avgöra de exakta anledningarna till detta. Vi har två teorier. Den ena är att, då bas 11 och 13 båda är primtal, krävs det fler operationer i evalueringen (då en dator räknar i två-potenser) än talet 12 som består av två två-potenser. Den andra teorin är att mängden minne arbetsföremålen använder i bas 12 bidrar till mindre *false-sharing* än de råkar göra i de andra baserna. False sharing är ett fenomen som uppstår när dataparallella beräkningar utförs på ett flertal processorer som har cache-minnen. Om den ena processorn har någon data i en cache-rad som egentligen är associerad med den andra processorns beräkningar och då invaliderar den raden så måste den andra processorn uppdatera sin cache-rad med korresponderande data helt i onödan. De seriella körningarna får inte lika tydlig förminskad prestandaförlust som de parallella CPU-körningarna får mellan dessa baser, vilket motvisar vår första teori.

Eftersom vi hade begränsad tid hade vi ingen möjlighet att utnyttja portabiliteten OpenCL erbjuder i våra prototyper. Som nämnt i kapitel 3.1.1 kan man låta värden undersöka den maskin den skall beräkna med för att bestämma konfiguration på körningen. Då räcker det inte med att undersöka antalet tillgängliga beräkningsenheter samt klockfrekvensen. Utan man måste även analysera huruvida processeringsenheten är bra på random access eller sekvensiell beräkning eller både och. Om problemet lämpar sig kan även tänka sig att man kan distribuera beräkningarna över både CPU och GPU. Svårigheten ligger då i att balansera arbetsbördan då CPU:n kan vara snabbare än GPU:n och vice versa.

Det finns grafikkort som är speciellt utvecklade för GPGPU. T.ex. NVIDIAs ”Tesla”-kort. Denna serie kort är i huvud taget inte gjorda för att ge video-output till bildskärmar, utan de har en arkitektur gjord för data-beräkningar. De har dessutom

en data-parallell cache[16]. Man kan tänka sig att den typen av problem vi behandlat lämpar sig bra för Tesla och kort med liknande arkitektur.

Uttömmande sökningar lämpar sig utmärkt att implementera i OpenCL förutsatt att processeringsenheten/processeringsenheterna dessa beräknas på kan utföra random access effektivt. Det tar dock väldigt lång tid att implementera något i OpenCL om man saknar tidigare erfarenhet med ramverket. Därav är det enbart lämpligt om man skall utveckla prestandakritiska programkomponenter eller något som kommer att användas länge och frekvent (t.ex. ett kod-bibliotek).

I framtiden kan man förvänta sig betydligt fler kärnor i datorer för hemmabruk. Intel gav nyligen ut en experimentell CPU med 48 kärnor[17]. Man kan då förvänta sig att implementationer i ett standardiserat ramverk som OpenCL verkligen kan löna sig i längden. Förutsatt att Moore's lag håller och trenden med fler kärnor håller ser framtiden ljus ut för OpenCL och dess likar.

Litteraturförteckning

[1] Intel's 90nm Pentium M 755: Dothan Investigated, URL: http://www.anandtech.com/show/1399/3, 2011-02-28

[2] OpenCL, URL: http://www.khronos.org/opencl/, 2011-02-28

[3] The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software, Herb Sutter, URL: http://www.gotw.ca/publications/concurrency-ddj.htm, 2011-03-09

[4] Chandrakasan, A.P.; Potkonjak, M.; Mehra, R.; Rabaey, J.; Brodersen, "Optimizing power using transformations", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol.14, no.1, pp.12-31, R.W., Jan 1995

[5] Munshi, Aafab; Gaster, Benedict; Mattson, Timothy; Ginsburg, Dan: "OpenCL Programming Guide", Addison-Wesley Professional, section 1.2, Jun 2011, ISBN: 978-0-321-74964-2

[6] "Intel Halts Development Of 2 New Microprocessors", URL: http://www.nytimes.com/2004/05/08/business/08chip.html, 2011-03-11

[7] Gary Anthes, "The Power of Parallelism", URL: http://www.computerworld.com/s/article/65878/The_Power_of_Parallelism, 2011-03-11

[8] David Eppstein, "On the NP-completeness of Cryptarithms", Computer Science Department, Columbia University, URL: http://www.ics.uci.edu/~eppstein/pubs/Epp-SN-87.pdf, 2000-06-8

[9] "A Brief History of General Purpose (GPGPU) Computing", URL: http://www.amd.com/us/products/technologies/stream-technology/opencl/pages/gpgpu-history.aspx, 2011-03-18

[10] "CUDA for GPU Computing", URL: http://news.developer.nvidia.com/2007/02/cuda_for_gpu_co.html, 2007-02-16

[11] "Khronos Launches Heterogeneous Computing Initiative", URL: http://www.khronos.org/news/press/releases/khronos_launches_heterogeneous_computing_initiative/, 2008-06-16

[12] ”The Khronos Group Releases OpenCL 1.0 Specification”, URL: http://www.khronos.org/news/press/releases/the_khronos_group_releases_opencl_1.0_specification/, 2008-12-09

[13] ”The OpenCL Specification Version 1.1”, Document revision 36, Khronos OpenCL Working Group, 2010-09-05

[14] Douglas Andrew Chin, ”Complexity Issues in General Purpose Parallel Computing”, Oxford University, 1991

[15] Ian Buck, ”Chapter 32. Taking the Plunge into GPU Computing”, Stanford University, URL: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter32.html 2005

[16] ”NVIDIA Tesla GPU Computing Solutions for HPC”, NVIDIA, URL: http://www.nvidia.com/docs/IO/43395/tesla_product_overview_dec.pdf, 2007

[17] Agam Shah, ”Intel to ship samples of experimental 48-core chip”, URL: http://www.computerworld.com/s/article/9174981/Intel_to_ship_samples_of_experimental_48_core_chip, 2010-04-07

Bilaga A

Fullständig implementation av verbalaritmetik-lösaren

A.1 C++

main.cpp:

```
#include <iostream>
#include <string>
#include <vector>
#include <cmath>
#include <ctime>
#include <sys/time.h>
#include <sstream>
#include <set>

using namespace std;

int complie_number(vector<int>& binds, vector<int>& problem, int base){
        int wordlength = problem.size();
        int number = 0;
    float base_float = (float) base;
        for(int j=0;j<wordlength;j++){
                number += binds[problem[wordlength-j-1]]*(int)(pow(
                    base_float,j) + .5f);
        }
        return number;
}

vector<int> encode(string& word, string& chars){
    vector<int> result;
    for(int i=0;i<word.length();i++){
        for(int j=0;j<chars.length();j++){
            if(word[i] == chars[j]){
                result.push_back(j);
                break;
            }
        }
```

```
    }
    return result;
}

char intToChar(int ch){
    if(ch < 10) return (char)ch+48;
    return (char) ch + 55;
}

string intToStr(int num, int base, int length){
    string result(length,'␣');
    for(int i=0,j=num;i<length;i++){
        int power = (int) round(pow((float)base,(length-i-1)));
        int number = (int)(j / power);
        result[i] = intToChar(number);
        j -= number*power;
    }
    return result;
}

void print_problem(vector<string>& problem){
        int arglen = 0;
        for(int i=0;i<3;i++){ if(problem[i].length() > arglen) arglen =
            problem[i].length();}
        cout << "␣␣";
        if(problem[0].length() < arglen) cout << "␣";
        cout << problem[0] << endl << "+␣";
        if(problem[1].length() < arglen) cout << "␣";
        cout << problem[1] << endl;
        for(int i=0;i<arglen+2;i++) cout << "-";
        cout << endl << "␣␣" << problem[2] << endl << endl;
}

bool solve_r(vector<int>& binds, int level, vector< vector<int> >&
    problem, bool *used_digit, int base){
        if(level < binds.size()){
                for(int i=0;i<=base-1;i++){
                        if(used_digit[i]) continue;
                        used_digit[i] = true;
            binds[level] = i;
                        if(solve_r(binds,level+1,problem, used_digit,
                            base)) return true;
                        used_digit[i] = false;
                }
                return false;
        }

        //validate
        int number[3] = {0,0,0};
        for(int i=0;i<3;i++){
                number[i] = complie_number(binds, problem[i],base);
        }
```

```
        if(binds[problem[0][0]]*binds[problem[1][0]]*binds[problem
            [2][0]] == 0) return false;

        if(number[0] + number[1] == number[2]) return true;

        return false;
}

vector<int> solve(vector<string>& problem, int base){
        string chars;
        set<char> exists;

        for(int i=0;i<3;i++)
                for(int j=0;j<problem[i].length();j++){
                        if(exists.find(problem[i][j]) == exists.end()){
                                exists.insert(problem[i][j]);
                                chars.push_back(problem[i][j]);
                        }
                }

        bool used_digits[base];
        for(int i=0;i<base;i++) used_digits[i] = false;

        vector<int> binds(chars.size());
    vector< vector<int> > encodedProblem;

    for(int i=0;i<3;i++) encodedProblem.push_back(encode(problem[i],
        chars));

        if(solve_r(binds, 0, encodedProblem, used_digits,base)){
        vector<int> result;
        for(int i=0;i<3;i++)
            result.push_back(complie_number(binds,encodedProblem[i],
                base));
        return result;
    }
    return vector<int>();
}

int main (int argc, char * const argv[]) {
        if(argc < 4){
                cout << "Wrong␣number␣of␣arguments." << endl;
                return 0;
        }

    volatile int base = 10;

    for(int i=4;i<argc;i+=2){

        if(string(argv[i]) == "-b"){
            base = atoi(argv[i+1]);
        }
    }
```

```
    vector<string> problem;
    for(int i=1;i<=3;i++) problem.push_back(argv[i]);
    cout << endl << "Solver: C++" << endl;
    cout << endl;
        cout << "Problem:" << endl << endl;
        print_problem(problem);
        timeval tic, toc;
    gettimeofday(&tic, NULL);
    vector<int> result = solve(problem,base);
    gettimeofday(&toc, NULL);
        if(result.size() == 0)
                cout << "No resolution." << endl;
        else{

                vector<string> solution;
                for(int i=0;i<3;i++){
                        stringstream ss;
                        ss << intToStr(result[i],base,problem[i].length
                            ());
                        solution.push_back("" + ss.str());
                }
                cout << "Resolution:" << endl << endl;
                print_problem(solution);

        }

        cout << "It took " << ((toc.tv_sec-tic.tv_sec)*1000 + (toc.
            tv_usec-tic.tv_usec)/1000) << " milliseconds to find a 
            conclution." << endl << endl;
        return 0;
}
```

A.2 OpenCL

main.cpp:

```
#include <iostream>
#include <sstream>
#include <fstream>
#include <stdexcept>
#include <sys/time.h>
#include <vector>
#include <set>
#include <cmath>
#ifdef __APPLE__
#include <OpenCL/OpenCL.h>
#endif
#ifdef WIN32
//include OpenCL for windows
#endif
using namespace std;
```

```
cl_platform_id getPlatformId(){
        cl_platform_id id;
        clGetPlatformIDs(1, &id, NULL);
        return id;
}

cl_device_id getDeviceId(cl_uint which_device, cl_device_type
    which_type){
        cl_platform_id id = getPlatformId();
        cl_uint number_of_devices;
        clGetDeviceIDs(id, which_type, 0, NULL, &number_of_devices);
        if(which_device > number_of_devices || which_device < 1) throw
            out_of_range("Invalid␣device␣number!");
        cl_device_id device_id_array[number_of_devices];
        clGetDeviceIDs(id, which_type, number_of_devices,
            device_id_array, NULL);

        return device_id_array[which_device - 1];
}

void printErrorCode(cl_int errCode){
    switch (errCode) {
        case CL_INVALID_PROGRAM:
            cout << "CL_INVALID_PROGRAM" << endl;
            break;
        case CL_INVALID_VALUE:
            cout << "CL_INVALID_VALUE" << endl;
            break;
        case CL_INVALID_DEVICE:
            cout << "CL_INVALID_DEVICE" << endl;
            break;
        case CL_INVALID_BUILD_OPTIONS:
            cout << "CL_INVALID_BUILD_OPTIONS" << endl;
            break;
        case CL_INVALID_OPERATION:
            cout << "CL_INVALID_OPERATION" << endl;
            break;
        case CL_COMPILER_NOT_AVAILABLE:
            cout << "CL_COMPILER_NOT_AVAILABLE" << endl;
            break;
        case CL_BUILD_PROGRAM_FAILURE:
            cout << "CL_BUILD_PROGRAM_FAILURE" << endl;
            break;
        case CL_OUT_OF_RESOURCES:
            cout << "CL_OUT_OF_RESOURCES" << endl;
            break;
        case CL_OUT_OF_HOST_MEMORY:
            cout << "CL_OUT_OF_HOST_MEMORY" << endl;
            break;
        default:
            cout << "Unknown␣error" << endl;
            break;
    }
```

```
}


cl_context createContext(cl_device_type processorType){
        cl_int err_code;
        cl_context_properties ctx_properties[] = {CL_CONTEXT_PLATFORM,(
            cl_context_properties)getPlatformId(),0};
        cl_context ctx = clCreateContextFromType(ctx_properties,
            processorType, NULL, NULL, &err_code);
        if(err_code != CL_SUCCESS) throw runtime_error("Could not
            create context");
        return ctx;
}

cl_command_queue createQueue(cl_context ctx, cl_device_id device_id){
        cl_command_queue queue = NULL;
        queue = clCreateCommandQueue(ctx, device_id, 0, NULL);
        if(queue == NULL) throw runtime_error("Could not create queue")
            ;
        return queue;
}

cl_program createProgram(cl_context ctx, cl_device_id device_id, const
    char* file_name, int base, int uniqueNumbers){

        cl_program program = NULL;
        cl_int err_code;

    ifstream file_stream(file_name);
        string program_source, line;
        if(!file_stream.is_open()) throw runtime_error("Could not open
            kernel file");
    stringstream defines;
    defines << "#define BASE " << base << '\n';
    defines << "#define UNIQUE_NUMBERS " << uniqueNumbers << '\n';
    program_source.append(defines.str());
    while(!file_stream.eof()){
        getline(file_stream,line);
        program_source.append(line + '\n');
    }
        const char* program_source_charpointer = program_source.c_str()
            ;
        program = clCreateProgramWithSource(ctx, 1, &
            program_source_charpointer, NULL, NULL);
        if(program == NULL) throw runtime_error("Could not create
            program");
        err_code = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
        if(err_code != CL_SUCCESS){
                char build_log[1 << 14];
                clGetProgramBuildInfo(program, device_id,
                    CL_PROGRAM_BUILD_LOG, sizeof(build_log), build_log,
                     NULL);
                cerr << "Error in kernel: ";
        printErrorCode(err_code);
```

```
        cout << endl << build_log << endl;
                throw runtime_error("Kernel␣error");
        }

        return program;
}

cl_kernel createKernel(cl_program program, const char* kernel_name){
        cl_kernel kernel = NULL;
        kernel = clCreateKernel(program, kernel_name, NULL);
        if(kernel == NULL) throw runtime_error("Could␣not␣create␣kernel
            ");
        return kernel;
}

vector<int> encode(string& word, string& chars){
    vector<int> result;
    for(int i=0;i<word.length();i++){
        for(int j=0;j<chars.length();j++){
            if(word[i] == chars[j]){
                result.push_back(j);
                break;
            }
        }
    }
    return result;
}

char intToChar(int ch){
    if(ch < 10) return (char)ch+48;
    return (char) ch + 55;
}

void print_problem(vector<string>& problem){
        int arglen = 0;
        for(int i=0;i<3;i++){ if(problem[i].length() > arglen) arglen =
             problem[i].length();}
        cout << "␣␣";
        if(problem[0].length() < arglen) cout << "␣";
        cout << problem[0] << endl << "+␣";
        if(problem[1].length() < arglen) cout << "␣";
        cout << problem[1] << endl;
        for(int i=0;i<arglen+2;i++) cout << "-";
        cout << endl << "␣␣" << problem[2] << endl << endl;
}


int main (int argc, char * const argv[]) {

    if(argc < 4){
                cout << "Wrong␣number␣of␣arguments." << endl;
                return 0;
        }
```

```
volatile size_t array_size = 100;
cl_device_type processorType = CL_DEVICE_TYPE_GPU;
int base = 10;
int partitionMagnitude = 2;

for(int i=4;i<argc;i+=2){
    if(string(argv[i]) == "-a"){
        partitionMagnitude = atoi(argv[i+1]);
    } else if(string(argv[i]) == "-p"){
        if(string(argv[i+1]) == "cpu")
            processorType = CL_DEVICE_TYPE_CPU;
        else if(string(argv[i+1]) == "gpu")
            processorType = CL_DEVICE_TYPE_GPU;
    } else if(string(argv[i]) == "-b"){
        base = atoi(argv[i+1]);
    }
}

float base_float = (float)base;
float pm_float = (float)partitionMagnitude;
float powParMag = pow(base_float,pm_float); // base^pow
array_size = powParMag-((int)powParMag) == 0 ? (int)powParMag : (
    int)powParMag + 1;

cout << endl;
cout << "Solver: OpenCL" << endl;
cout << "procesor type: ";
if(processorType == CL_DEVICE_TYPE_CPU)cout << "CPU";
else if(processorType == CL_DEVICE_TYPE_GPU) cout << "GPU";
cout<< endl;
cout << "array size: " << array_size << endl;
cout << "base: " << base << endl;
cout << endl;

vector<string> problem;
for(int i=1;i<=3;i++) problem.push_back(argv[i]);

    cout << "Problem:" << endl << endl;
    print_problem(problem);

    cl_mem memory_objects[4];
size_t global_work_size_array[] = {array_size};

size_t local_work_size_array[] = {1};


// Process puzzle
int wordLength[3] = {problem[0].length(),problem[1].length(),
    problem[2].length()};
int totalLength = wordLength[0]+ wordLength[1]+ wordLength[2];
int words[totalLength];

string chars;
```

```
    set<char> exists;

    for(int i=0;i<3;i++)
            for(int j=0;j<problem[i].length();j++){
                    if(exists.find(problem[i][j]) == exists.end()){
                            exists.insert(problem[i][j]);
                            chars.push_back(problem[i][j]);
                    }
            }

for(int i=0,o=0;i<3;i++){
    vector<int> translated = encode(problem[i], chars);
    for(int j=0;j<wordLength[i];j++,o++){
        words[o] = translated[j];
    }
}


int uniqueNumbers = chars.length();
int solution[uniqueNumbers];


    // Create OpenCL stuff

    cl_context ctx = createContext(processorType);
    cl_device_id device_id = getDeviceId(1, processorType);
    cl_command_queue queue = createQueue(ctx, device_id);
    cl_program program = createProgram(ctx, device_id, "
        solve_kernel.cl",base,uniqueNumbers);
    cl_kernel kernel = createKernel(program, "solve");
cl_kernel resetKernel = createKernel(program, "resetNumbers");




for(int i=0;i<uniqueNumbers;i++) solution[i] = 0;

    memory_objects[0] = clCreateBuffer(ctx, CL_MEM_READ_ONLY |
        CL_MEM_COPY_HOST_PTR , sizeof(int)*totalLength, words, NULL
        );// words
    memory_objects[1] = clCreateBuffer(ctx, CL_MEM_READ_ONLY |
        CL_MEM_USE_HOST_PTR , sizeof(int)*3, wordLength, NULL); //
        wordlength
memory_objects[2] = clCreateBuffer(ctx, CL_MEM_READ_WRITE |
    CL_MEM_USE_HOST_PTR , sizeof(int)*uniqueNumbers, solution, NULL
    ); // solution
memory_objects[3] = clCreateBuffer(ctx, CL_MEM_READ_WRITE |
    CL_MEM_ALLOC_HOST_PTR , sizeof(int), NULL, NULL); //
    solution_found


for(int i=0;i<4;i++)if(memory_objects[i] == NULL) throw
    runtime_error("Could not create memory objects");
```

```
cl_int err_code = CL_SUCCESS;
err_code |= clSetKernelArg(kernel, 0, sizeof(int), &
    partitionMagnitude);
     err_code |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &
         memory_objects[0]);
     err_code |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &
         memory_objects[1]);
err_code |= clSetKernelArg(kernel, 3, sizeof(cl_mem), &
    memory_objects[2]);
err_code |= clSetKernelArg(kernel, 4, sizeof(cl_mem), &
    memory_objects[3]);

err_code |= clSetKernelArg(resetKernel, 0, sizeof(cl_mem), &
    memory_objects[3]);
     if(err_code != CL_SUCCESS) throw runtime_error("Could not set 
         kernel arguments");

     bool run_in_opencl = true;
bool timedOut = false;

timeval tic, toc;

gettimeofday(&tic, NULL);

     if(run_in_opencl){
     err_code = clEnqueueNDRangeKernel(queue, resetKernel, 1, NULL,
         local_work_size_array, local_work_size_array, 0, NULL, NULL
         );
     clFinish(queue);
     err_code = clEnqueueNDRangeKernel(queue, kernel, 1, NULL,
         global_work_size_array, local_work_size_array, 0, NULL,
         NULL);
     if(err_code != CL_SUCCESS) throw runtime_error("Could not 
         enqueue kernel");
              cl_int runResult = clFinish(queue);
     if(runResult != CL_SUCCESS) cout << "Something went wrong in 
         runtime: ";
     switch (runResult) {
          case CL_INVALID_COMMAND_QUEUE:
               cout << "Invalid command queue (timeout might have been
                    reached)" << endl;
               timedOut = true;
               break;
          case CL_OUT_OF_RESOURCES:
               cout << "Out of resources" << endl;
               break;
          case CL_OUT_OF_HOST_MEMORY:
               cout << "Out of host memory" << endl;
               break;
          case CL_SUCCESS:
               cout << "Execution went fine" << endl;
               break;
          default:
               cout << "unknown" << endl;
```

```
                break;
            }
            } else {

            }

        gettimeofday(&toc, NULL);


            cout << "Calculation took " << ((toc.tv_sec-tic.tv_sec)*1000 +
                (toc.tv_usec-tic.tv_usec)/1000) << " milliseconds" <<endl<<
                endl;

        int solution_found;
        clEnqueueReadBuffer(queue, memory_objects[3], true, 0, sizeof(int),
             &solution_found, 0, NULL, NULL);

        if(!solution_found)
                cout << "No resolution." << endl;
            else{

                vector<string> solution_str;
                for(int i=0,o=0;i<3;i++){
                        string s(wordLength[i],' ');
                    for(int j=0;j<wordLength[i];j++)
                        s[j] = intToChar(solution[words[o+j]]);
                        solution_str.push_back("" + s);
                    o+=wordLength[i];
                }
                cout << "Resolution:" << endl << endl;
                print_problem(solution_str);

            }

            // Release
        for(int i=0;i<4;i++) clReleaseMemObject(memory_objects[i]);
        clReleaseKernel(resetKernel);
            clReleaseKernel(kernel);
            clReleaseProgram(program);
            clReleaseCommandQueue(queue);
            clReleaseContext(ctx);


            cout << "All good" << endl << endl;
}
```

solve_kernel.cl:

```
kernel void resetNumbers(global int *solution_found_ptr
                        ){
    *solution_found_ptr = 0;
}

bool checkSolution(global const int *words,
```

```
                    global const int *wordLengths,
                    private int *stack
                    ){
    int sums[3];
    int offset = 0;
    for(int i=0;i<3;i++){
        sums[i] = 0;
        if(stack[words[offset]] == 0) return false;
        for(int j=0;j<wordLengths[i];j++){
            sums[i] += stack[words[offset + wordLengths[i] -j -1]]*
                round(pown((float)BASE,j));
        }
        offset += wordLengths[i];
    }
    if(sums[0] + sums[1] == sums[2]){
        return true;
    }
    return false;
}

kernel void solve(global const int partitionMagnitude,
                  global const int *words,
                  global const int *wordLengths,
                  global int *solution_ptr,
                  global int *solution_found_ptr
                  ){
    if(*solution_found_ptr == 1){
        return;
    }
    int gid = get_global_id(0);

    private int stack[UNIQUE_NUMBERS];
    private int used_numbers[BASE];

    for(int i=0;i<BASE;i++){
        used_numbers[i] = 0;
    }
    for(int i=0;i<UNIQUE_NUMBERS;i++){
        stack[i] = 0;
    }
    for(int i=0,j=gid;i<partitionMagnitude;i++){
        int power = (int) round(pow((float)BASE,(partitionMagnitude-i
            -1)));
        stack[i] = (int)(j / power);
        if(used_numbers[stack[i]] == 1) return;
        used_numbers[stack[i]] = 1;
        j -= stack[i]*power;
    }

    int depth = partitionMagnitude;
    bool recurse_down = true;


    while(true){
```

```
        if(*solution_found_ptr == 1){
            return;
        }

        if(depth == UNIQUE_NUMBERS){

            if(checkSolution(words,wordLengths,stack)){
                for(int i=0;i<UNIQUE_NUMBERS;i++){
                    solution_ptr[i] = stack[i];
                }
                *solution_found_ptr = 1;
                break; // solution found!
            }
            recurse_down = false;
            depth--;
            continue; // solution was not valid, recurse up
        }


        /** recursive section **/


        if(!recurse_down){
            used_numbers[stack[depth]] = 0;
            stack[depth]++;
        }

        while(stack[depth] < BASE && used_numbers[stack[depth]] == 1){
            stack[depth]++;
        }


        if(stack[depth] == BASE){ // no free number was found
            if(depth == partitionMagnitude){
                break; // all solutions exhausted
            }
            recurse_down = false;
            stack[depth] = 0; // reset this element
            depth--;
            continue; // recurse up
        }

        used_numbers[stack[depth]] = 1; //allocate this number
        recurse_down = true;
        depth++;
        // recurse down
    }
}
```

Bilaga B

Testkod

B.1 Testkod-genererare

Generate.java:

```
import java.util.*;
import java.io.*;

/**
 *
 * @author Carl Björkmam, Michel Cupurdija
 */
public class Generate
{

    public static char intToChar(int ch){
        if(ch < 10) return (char)(ch+48);
        return (char) (ch + 55);
    }

    public static String intToStr(int num, int base){
        String result = "";
        int length = (int)Math.round((Math.log10(num)+.5f));
        for(int i=0,j=num;i<length;i++){
            int power = (int) Math.round(Math.pow((float)base,(length-i
                -1)));
            int number = (int)(j / power);
            result += intToChar(number);
            j -= number*power;
        }
        return result;
    }

    public static void main(String[] args){
        int base = 10;
        if(args.length > 0) base = Integer.parseInt(args[0]);
        int maxNumbers = 5;
        int batchSize = 100;
```

```
        try{
            FileWriter fw = new FileWriter(new File("out.txt"),false);
            Random rand = new Random();
            int maxRand = (int)Math.pow(10,maxNumbers);
            for(int i=0;i<batchSize;){

                int a = rand.nextInt(maxRand);
                int b = rand.nextInt(maxRand);
                int c = a + b;
                String a_str = intToStr(a,base);
                String b_str = intToStr(b,base);
                String c_str = intToStr(c,base);
                if(a_str.charAt(0) == '0' ||
                   b_str.charAt(0) == '0' ||
                   c_str.charAt(0) == '0') continue;
                i++;
                fw.write(""+intToStr(a,base)+" "+intToStr(b,base)+" "+
                    intToStr(c,base)+" -b "+base+"\n");
            }
            fw.close();
        } catch(Throwable e){
            System.err.println("error!");
        }
    }
}
```

B.2 Testkod-analyserare

AnalyzeSpeed.java:

```
import java.util.*;
import java.io.*;

/**
 *
 * @author Carl Björkmam, Michel Cupurdija
 */
public class AnalyzeSpeed
{
    public static void main(String[] args){
        if(args.length < 1){ System.out.println("too few args"); return
            ;}
        boolean scale = false;
        if(args.length == 2 && args[1].equals("-s")) scale=true;
        try{
            Scanner sc = new Scanner(new File(args[0]));
            int[] stats = new int[7];
            int[] count = new int[7];
            int[] row = new int[7];
            int goodRows = 0;
            while(sc.hasNextLine()){
                sc.nextLine(); // line nr
```

```
                sc.nextLine(); // arguments
                boolean timeOut = false;
                for(int i=0;i<7;i++){
                    String line = sc.nextLine().split(" ")[1];
                    if(!line.equals("-")){
                        count[i]++;
                        row[i] = Integer.parseInt(line);
                    } else{
                        timeOut = true;
                    }
                }
                sc.nextLine(); // empty row
                if(!scale || !timeOut){
                    goodRows++;
                    for(int i=0;i<7;i++)
                        stats[i] += row[i];
                }
            }
            System.out.println("Result:");
            String[] types = {"C++ ","GP2 ","GP3 ","GP4 ","CP2 ","CP3 "
                ,"GP4 "};
            for(int i=0;i<7;i++){
                if(scale) stats[i] /= goodRows;
                else stats[i] /= count[i];
                System.out.println(types[i]+stats[i]+" ms "+(100-count[
                    i])+" misses");
            }
        } catch(Throwable e){
            System.out.println("error");
        }
    }
}
```

AnalyzeFastest.java:

```
import java.util.*;
import java.io.*;

/**
 *
 * @author Carl Björkmam, Michel Cupurdija
 */
public class AnalyzeFastest
{
    public static void main(String[] args){
        if(args.length < 1){ System.out.println("too few args"); return
            ;}
        try{
            Scanner sc = new Scanner(new File(args[0]));
            int[] stats = new int[7];
            int[] row = new int[7];
            int goodRows = 0;
            while(sc.hasNextLine()){
                sc.nextLine(); // line nr
                sc.nextLine(); // arguments
```

```
                int fastest = 0;
                for(int i=0;i<7;i++){
                    String line = sc.nextLine().split(" ")[1];
                    if(!line.equals("-")){
                        row[i] = Integer.parseInt(line);
                        if(row[i] < row[fastest]) fastest = i;
                    }
                }
                sc.nextLine(); // empty row
                stats[fastest]++;
            }
            System.out.println("Result:");
            String[] types = {"C++ ","GP2 ","GP3 ","GP4 ","CP2 ","CP3 "
                ,"GP4 "};
            for(int i=0;i<7;i++){
                System.out.println(types[i]+stats[i]+" times");
            }
        } catch(Throwable e){
            System.out.println("error");
        }
    }
}
```

AnalyzeSlowest.java:

```
import java.util.*;
import java.io.*;

/**
 *
 * @author Carl Björkmam, Michel Cupurdija
 */
public class AnalyzeSlowest
{
    public static void main(String[] args){
        if(args.length < 1){ System.out.println("too few args"); return
            ;}
        try{
            Scanner sc = new Scanner(new File(args[0]));
            int[] stats = new int[7];
            while(sc.hasNextLine()){
                sc.nextLine(); // line nr
                sc.nextLine(); // arguments
                for(int i=0;i<7;i++){
                    String line = sc.nextLine().split(" ")[1];
                    if(1 <= i && i <= 3) continue;
                    int res = Integer.parseInt(line);
                    if(res > stats[i]) stats[i] = res;
                }
                sc.nextLine(); // empty row
            }
            System.out.println("Result:");
            String[] types = {"C++ ","","","","CP2 ","CP3 ","CP4 "};
            for(int i=0;i<7;i++){
                if(1<=i&&i<=3)continue;
```

```
                System.out.println(types[i]+stats[i]+" ms");
            }
        } catch(Throwable e){
            System.out.println("error");
        }
    }
}
```

B.3 Testkörare

bench.sh:

```
#!/bin/sh

exec<"out.txt"
val=1
while read line
do
echo Line $val
echo $line
echo "C++ \c"
./verbalarithmetic $line
echo "GP2 \c"
./clverbalarithmetic $line
echo "GP3 \c"
./clverbalarithmetic $line -a 3
echo "GP4 \c"
./clverbalarithmetic $line -a 4
echo "CP2 \c"
./clverbalarithmetic $line -p cpu
echo "CP3 \c"
./clverbalarithmetic $line -p cpu -a 3
echo "CP4 \c"
./clverbalarithmetic $line -p cpu -a 4
echo ""
val=`expr $val + 1`;
done
```