AALBORG UNIVERSITET
STUDENTERRAPPORT

Title:
C++ AMP og funcalc

Project period:
P10, forår 2016

Project Group:
?

Authors:
Niels Brøndum Pedersen
Supervisor:
Bent Thomsen

Total Pages: - 0
Appendix: - 1 CD
Completion date: - 00-00-0000

Synopsis:

NOT MADE

1 Intro

I tidligere Projekt blev der kigget på, om GPU'ens regnekraft kunne bruges i flere programmer for at øge ydeevne. I dette projekt var det også fundet, at det ikke var nemt at programmer til en GPU, fordi metoderne til at kode til en GPU ikke har en godt dokumentation og er kompliceret at lave. Derfor vil jeg i dette projekt prøve at at lave en simple programmering metode for at kunne bruge en GPU's regnekraft GPU, uden man skal havde de store viden inden for det. Regneark bliver brugt af mange og er forholdsvis nemt at programmere, derfor kunne det være interessant, at lave en funktion til et regneark program for at udregne med en GPU. Regneark programment jeg der blev brugt i dette projekt er *FuncalcFuncalc* der er open source. Inden jeg går i gang med at lave GPU funktion vil jeg kigge på tre API for at programmere til en GPU (CUDA, CUDAfy og C++ AMP), for at se hvad der virker godt i forhold til matrix Multiplatikon for at give et godt overblik over hvad der skal bruges når udvidelsen til regnearket skal laves.

Andre har også prøvet at lave det nemmere at kode til en GPU (nogen ref her og eksempler). Det største problem jeg fandt mens jeg arbejde på projektet, var, at igennem CUDA, CUDAfy og C++ AMP ikke gav mulighed for at sende en funktion til GPU, jeg kunne heller ikke finde en smart måde at compile GPU kode på run time, og at når man koder i Excel, bliver det gjort på runtime af programmet der køre excel dokumentet. Måden jeg løste dette problem var ved at sende en opskrift der blev levet efter et abstrakt syntaks træ sammen med data der skal arbejdes på. Med den information vil funktion vide hvordan den skal udregne på den sendte data, dette giver mulighed for at "kode" til en GPU på run time uden at der skal compiles noget på run time. Et anden problem, dog mindre, var at jeg ikke kunne finde løsning på hvordan et program kunne håndtere at bruge forskellige tråde mængder på run time, dette problem jeg jeg kun løst på en meget simple måde der også kan klare forskellige GPU korts.

2 relateret arbejde

Til at hjælpe med at fremstille resultater der kunne bruges, er Papiret *Microbenchmarks in Java and C#*[2] blevet brugt til at hjælpe med at fremstille resultater der kan bruges til at kigge på tid.

3 Test af Matrix Multiplatikon

For at give en bedre indblik på hvor meget speed-up det giver at bruge GPU'en i forhold til CPU'en, er der fremstillet simpel programmer der går en matrix multikapion. Der er lavet to versioner for hver GPU library, der er blevet kigget på. Forskellen mellem versionerne er, at det ene gør brug af en dimension og den anden bruger to dimensioner for array. De library der er blevet tested er C++ AMP, CUDA, for C++ og C# med normal CPU udregning er også lavet for kunne give en grundlag for hvor meget speed-up man for.

Testene bliver gjort for at give et bedre indblik om GPU er bedre en CPU, der er nogen forskel på hvordan input til GPU ser ud i forhold til om speed-up. Derefter er

der blevet lavet et test program med C++ AMP der bliver kaldt fra C# kode, for at se hvordan dette kunne gøres og om det har den store effekt på udregnings tid med at bruge denne metode der er blevet fundet. Dette bliver gjort for fordi programmet Funcalc er skrevet i C#.

Testene er blevet fremstillet efter papiret (Microbenchmarks in Java and C#). Af de versioner af test papiret beskriver bruges der Mark3 version.

3.1 Resultater

CPU ikke lavet i nu

CUDA CUDA 1D MA in C++ mean, sdev
size: 5 time: 0.542 , 0.242
size: 10 time: 0.477 , 0.014
size: 20 time: 0.498 , 0.007
size: 50 time: 0.609 , 0.008
size: 100 time: 0.984 , 0.021
size: 200 time: 3.063 , 0.035
size: 300 time: 8.957 , 0.114
size: 400 time: 20.526 , 0.106
size: 500 time: 38.194 , 0.027
size: 600 time: 63.997 , 0.027
size: 700 time: 100.162 , 0.020
size: 800 time: 147.458 , 0.135
size: 900 time: 209.182 , 0.025
size: 1000 time: 286.169 , 0.046

CUDAfy CUDAfy 1D MA in C Sharp mean , sdev
size: 5 time: 0,533 , 0,147
size: 10 time: 0,478 , 0,019
size: 20 time: 0,487 , 0,010
size: 50 time: 0,543 , 0,012
size: 100 time: 0,607 , 0,012
size: 200 time: 0,87 , 0,013
size: 300 time: 1,661 , 0,041
size: 400 time: 3,617 , 0,047
size: 500 time: 6,359 , 0,107
size: 600 time: 8,456 , 0,096
size: 700 time: 14,109 , 0,050
size: 800 time: 19,492 , 0,035
size: 900 time: 24,987 , 0,036
size: 1000 time: 37,275 , 0,029

CUDAfy 2D MA in C Sharp mean , sdev
size: 5 time: 0,499 , 0,033
size: 10 time: 0,492 , 0,010
size: 20 time: 0,499 , 0,005
size: 50 time: 0,56 , 0,014
size: 100 time: 0,74 , 0,014
size: 200 time: 1,594 , 0,013
size: 300 time: 3,658 , 0,116
size: 400 time: 6,251 , 0,213
size: 500 time: 10,728 , 0,062
size: 600 time: 15,122 , 0,049
size: 700 time: 22,367 , 0,051
size: 800 time: 28,626 , 0,039
size: 900 time: 41,546 , 0,025
size: 1000 time: 50,958 , 0,011

C++ AMP AMP 1D MA in C++ mean, sdev
size: 5 time: 0.826 , 0.373
size: 10 time: 0.7 , 0.020
size: 20 time: 0.694 , 0.013
size: 50 time: 0.714 , 0.034
size: 100 time: 0.819 , 0.017
size: 200 time: 1.266 , 0.021
size: 300 time: 1.999 , 0.07
size: 400 time: 3.56 , 0.038
size: 500 time: 6.042 , 0.039
size: 600 time: 9.649 , 0.016
size: 700 time: 14.629 , 0.017
size: 800 time: 21.16 , 0.012
size: 900 time: 30.643 , 0.027
size: 1000 time: 42.58 , 0.044

C++ AMP og C# AMP 1D MA in C Sharp mean , sdev
size: 5 time: 0,637 , 0,342
size: 10 time: 0,556 , 0,022
size: 20 time: 0,56 , 0,0205
size: 50 time: 0,552 , 0,011
size: 100 time: 0,656 , 0,026
size: 200 time: 1,076 , 0,032
size: 300 time: 1,931 , 0,046
size: 400 time: 3,478 , 0,017
size: 500 time: 6,009 , 0,018
size: 600 time: 9,534 , 0,021

size: 700 time: 14,509 , 0,023
size: 800 time: 21,107 , 0,014
size: 900 time: 30,587 , 0,020
size: 1000 time: 42,488 , 0,039

AMP 2D MA in C Sharp mean , sdev
size: 5 time: 0,611 , 0,180
size: 10 time: 0,562 , 0,014
size: 20 time: 0,581 , 0,012
size: 50 time: 0,685 , 0,007
size: 100 time: 1,099 , 0,008
size: 200 time: 3,061 , 0,232
size: 300 time: 7,895 , 0,027
size: 400 time: 14,574 , 0,080
size: 500 time: 21,529 , 0,182
size: 600 time: 29,924 , 0,685
size: 700 time: 39,074 , 0,964
size: 800 time: 54,605 , 0,070
size: 900 time: 69,917 , 0,117
size: 1000 time: 90,851 , 0,098

3.2 Efter Tanker

Det største problem jeg har haft med med CUDA og CUDAfy er, at man selv skal finde ud af hvor mange blokke og antal tråde pr. blok. Hvilket godt kan give nogen problemer når man arbejder med et program der skal arbejde med varierende input. Efter hvad jeg ar fundet på nettet angående problem med hvor mange blokke og antal tråde pr. blok man skal bruge har jeg for det meste fundet at man skal teste sig frem alt efter hvad der virker godt på det hardware man tester på.

For at løse dette problem for mig, har jeg lavet en generist metode der finder ud af hvor mange blokke der skal bruges, hvis det hele ikke kan gøres på en blok, denne metode er dog ikke perfekt.

En anden ting jeg er kommet på er plads mangle på GPU'en, min GPU kunne kun klare at gange matrixer der har max størrelse på ???. En løsning på dette problem kunne være at begynde at bruge billede hukommelse på GPU'en til readonly data. Hvilke skulle være muligt med CUDA og skulle ikke være muligt med CUDAfy.

En mindre fejl der begyndte at vise sig var at, når men kalder GPU funktion flere gange blev resultatet plusse med 3 hvis man ikke først sætter det til 0 inden man udregner. Denne fejl begundte at vise sig efter 3 udringer efter hinanden.

4 GPU Funktion

I dette projekt havde jeg planer om at lave en ekstra funktion til *CoreCalc* for at kunne "kode" til GPU igennem regnearket. Funktion har jeg kaldt *GPU*, man skal markere de

felter men vil bruge som output, ligesom funktion *TRANSPOSE*. *GPU* har 2 input, det første er et array, eller et data sæt, der marker det data der skal udregnes på, og det andet er et udtryk/funktion, hvor talende i udtrykket peger til hvilken kolonne i den data den skal indsætte på dette placering i udregningen.

På 1 kan der ses et billede af *CoreCalc* hvor *GPU* er i brug. A1 til B6 er data sættet der vil blive brugt som input til *GPU*. I C6 kan *GPU* funktion beskrivelsen blive set, den tager A1 til B6 som input array (A1:B6) og en funktion (1+2), som beskriver til *GPU* at kolonne 1 skal plusse samme med kolonne 2. Bemærk at C1 til C6 er markeret mens at funktion bliver skrevet, dette bliver gjort fordi at *GPU* output også er at array, så derved viser du *GPU* funktion hvor dens output skal være. Bemærk at det markeret område har lige så mange rækker som input har.

På 2 kan det ses at resultaterne fra *GPU* er lavet og placeret i alle de markeret kolonner fra 1.

C6	A	B	C	D
1	1	7		
2	2	8		
3	3	9		
4	4	10		
5	5	11		
▶ 6	6	12	=GPU(A1:B6 , 1+2)	
7				

Fig. 1: et billede af *CoreCalc* hvor *GPU* bliver brugt.

G16	A	B	C
1	1	7	8
2	2	8	10
3	3	9	12
4	4	10	14
5	5	11	16
6	6	12	18
7			

Fig. 2: et billede af *CoreCalc* hvor *GPU* har lavet dens udregning.

5 GPU-calculate til CoreCalc

I dette projekt vil CUDAfy blive brugt, til at øge regnekræften i open source programmet *Funcalc* der kan findes på hjemmesiden [1]. *Funcalc* en udvidelse til Corecalc, der er en implementering af et regneark funktionalitet lavet i sproget C#, det er lavet som et forskning prototype som ikke er ment for kunne blive brugt i stedet for de officielse versioner, såsom Microsoft Excel.

Klassen *GPU_ func* i *GPU_ calculate* mappen er hvor det meste arbejde ligger fra dette projekt. For at kunne bruge det jeg har fremstillet, er der også tilført noget kode i klassen *Function*.

5.1 GPU_ func

Klassen *GPU_ func* er der seks funktioner og en konstruktør.

konstruktøren bliver brugt til at hente information om GPU'en der bruges til at bestemme om en blok er nok, hvis ikke hvor mange blokke skal der så bruges. Grunden til at information bliver hentet når man laver klassen er for minimere tiden funktion skal bruge på udregning, da jeg har observeret tager en god potion tid at hente denne information.

5.2 makeFunc

makeFunc og *makeFuncHelper* funktionerne bruges til at fremstille en opskrift med hvordan hvordan GPU'en skal udregne. Denne liste har *X* antal a fire fire tal som er en enkle udregning (+,-,*,/). Tal første og tredje tal er hvad variabler der skal gøres noget med, det andet tal er for at bestemme hvilken udregning der skal gøres (+,-,*,/) og det fjerde og sidste bliver brugt til at bestemme om resultatet skal lige ligges i en midlertidig variable eller om den skal ligges i resultat listen.

For at give et eksemple kan vi tage regnestykket *(1+2)*(3+4)*, det her ses om en kommando for GPU funktion hvor tallene bliver brugt til at bestemme hvilken kolonne, den skal tag værdien fra. Den kunne komme til at se sådan ud: (1,1,2,-1),(3,1,4,-2),(-1,3,-2,0). 1+2 er blevet lavet om til (1,1,2,-1), 3+4 er blevet om til (3,1,4,-2) og ()*() er lavet til (-1,3,-2,0). Grunden til at der står minus -1 og -2 ved udregningen for 1+2 og 3+4, er at minus værdier bliver brugt til at beskrive midlertidig variabler og 0 for output.

5.3 findnumberOfTempResult

Dette er en simple funktion der går gennem en opskrift og finder det antal af midlertidig resultater der skal bruges i opskriften.

5.4 calculate

Når man skal bruge en GPU gennem CUDA og CUDAfy skal man gøre forskellige ting, disse ting bliver gjort her. Variablerne *numberOfTempResult*, *SizeOfInput*, *AmountOfNumbers* og *numberOfFunctions* bliver lavet til at starte med. *SizeOfInput* er hvor mange koloner der er med i input, *AmountOfNumbers* er hvor mange rækker der med i input.

Der bliver lavet to array *output* og *tempResult*. *tempResult* Bliver brugt til holde midlertidig resultater for GPU regne stykke, grundet dette bliver gjort her, er at mængden af midlertidig i en funktion kan variere og efter hvad jeg har kunne finde på nettet giver CUDAfy ikke mulighed for at lave et array på run time.

Det første der bliver udregnet er, hvor mange block og tråde der skal bruges, alt efter hvad hardware kan holde til.

Derefter vil array få allokeret plads på GPU, Hvorefter vil de der har nødvendig data blive send over. Derefter vil selve GPU funktion blive kaldt. Derefter vil output data blive hente tilbage fra GPU og til sidst vil den hukommelse der er brugt på array blive frigivet.

5.5 GPUFunc

GPUFunc er funktion der vil blive kørt på GPU. Den har tre ting der gør for hver punkt i opskriften. Først vil den finde de variabler den skal bruge, midlertidig eller fra input, så vil den gøre noget med de to variabler den har fundet, +,-,*,/, og til sidst vil den finde ud af hvor output skal lægges hen.

5.6 kode i Corecalc Function klasse

Koden der findes i *Function* for *Corecalc*, der har bundet GPU koden sammen til resten af *Corecalc*. Der er tre forskellige steder at kode er blevet sat ind, selve *Function* har fået en private *GPU_func GPU* der bliver initierets når *Function* bliver. For at GPU funktion skal kunne blive kaldt, er den blevet sat ind i tabellen af funktioner.

Den sidste del af koden i denne klasse ligger i *GPUFunction*, koden her er samlelede mellem *Corecalc* og *GPU_ func*. Koden opgave er tage funktions kaldet input og lave det til information som *GPU_ func* klassen vil kunne bruge til at udregne med, det gør den ved at hente det beskrevet array, lave det om til et double array, derefter tager den det andet input og laver om til en opskrift. Med opskriften og double array kan den kalde *calculate* der giver et double array tilbage med resultaterne, til sidst vil den fremstille et *Corecalc value* array, hvor resultatet vil blive kopieret over i og derefter vil blive sent tilbage.

6 Test af GPU-calculate

6.1 Resultater

References

1. IT University of Copenhagen. Corecalc and funcalc spreadsheet technology in c sharp. `http://www.itu.dk/people/sestoft/funcalc/`.
2. Peter Sestoft (sestoft@itu.dk). Microbenchmarks in java and c sharp, 9 2015.