AALBORG UNIVERSITET
STUDENTERRAPPORT

Title:
CUDAfy og Corecalc

Project period:
P10, forår 2016

Project Group:
nped11@student.aau.dk

Author:
Niels Brøndum Pedersen
Supervisor:
Bent Thomsen

Total Pages: - 28
Completion date: - 12-06-2016

Synopsis:

In this project, We report on research on making it easier for using a GPU, by using a spreadsheet program as a medium for coding to it. Before the coding of this functionality for a program. a test with matrix multiplication have conducted to see what GPU library best could work in the scope of this project. The libraries that have been testes are: CUDA, C++ AMP and CUDAfy. it was chosen to make a GPU function using CUDAfy, since it showed promise in the tests and also the open source spreadsheet program that was used, *Corecalc and Funcalc* was made in c# and CUDAfy was design to be used in that programming language. There have been conducted a test to see if this new GPU function showed that it coud be useful time wise and these tests showed if the expression that is being calculated is large enough.

The content of the report is free to use, yet a official publication (with source references) may only be made by agreement from the authors of the report.

1 Intro

Alt kode kan finde på denne GitHub profil Dugin13, her er et link til selve projektet [7].

I tidligere Projekt [5] blev der undersøgt på forskellen mellem en CPU, som bruges til hverdagen og er god til sekventielle beregningen og GPU, der for det meste bruges i grafiskes programmer til beregning af pixel. I tidligere projekt blev det fundet, at det ikke var nemt at programmer til en GPU, fordi metoderne til at kode til en GPU ikke har en god dokumentation og er kompliceret at lave. En artikel fundet i løbet af det projektet viste også, at meget af tiden bliver brugt på at overføre data til GPU [4].

Ud fra det tidligere projekt vil formålet med dette projekt, være at lave en simple programmering metode for at kunne bruge en GPU, uden man skal havde de store viden inden for dette, for at kunne det vil et regnearks program, der bliver brugt af mange og det er forholdsvis nemt at lave funktioner. Derfor kunne det være interessant, at lave en funktion til et regneark program, der vil flytte udregningen over på GPU. Regneark programment der bruges i dette projekt er *Corecalc and Funcalc*. *Corecalc and Funcalc* er et *open source* regnearks program lavet som et platform for at eksperimenter med teknologi og nye funktioner.

Før GPU funktion til *Corecalc and Funcalc* vil blive lavet, vil en test blive udarbejdet, der kigges 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 og 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, for at give et eksempel kunne være *Chestnut*[14]. Det første problem blev fundet da der blev arbejde på projektet var, at igennem *CUDA*, *CUDAfy* og *C++ AMP* ikke gav mulighed for at sende et funktion udtryk til GPU. Et andet problem der kom frem var, at når man arbejdere i et regneark program som en bruger, vil man som regel omskrive en funktion flere gange, eller kunne det være at funktion skal udregnes flere gange på grund af ændringer i arket. Det andet problem kunne blive løst ved at sende en indkodning af et udregnings udtryk der laves efter et abstrakt syntaks træ. 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*. Der er dog nogen der har kigget på ideen med at *compile* GPU kode på *run time*, såsom *Firepile* [9] der er en udvidelse til *Scala*. Et andet problem var at man ikke kunne finde en rigtig løsning på, hvordan et program håndtere forskellige mængder af tråde på *run time* til en GPU, problemet bliver løst på en meget simple måde, der også kan klare forskellige GPU *cards*.

I kapitel *Relateret Arbejde* 2 har arbejdet inden for det samme område som dette projekt vil.

I kapitel *Matrix Multiplatikon* 3 vil en beskrivelse af koden, som bliver brugt til at teste hvilken GPU bibliotek der virker bedst sammen med det regneark programmet *Corecalc and Funcalc*. Ved at fremstille et program der kan gøre en matrix multiplatikon for hvert bibliotek, der bliver kigget på.

I kapitel *Test af Matrix Multiplatikon* 4 kan udstyret på computeren, som er blevet testet på dokumenteres, samt resultaterne fra testene med matrix multiplatikon af de forskellige biblioteker.

GPU Funktion 5 giver en beskrivelse af hvordan funktion, der vil blive fremstillet til *Corecalc and Funcalc* vil virke i programmet for brugeren.

GPU-calculate til CoreCalc 6 beskriver klassen, der er lavet for at kunne udregne GPU i *Corecalc and Funcalc* ved at sende en indkodning til GPU af at udtryk, som GPU ville kunne bruge til at udregne, det ønskede resultat ud fra data det bliver sendt med.

For at finde ud af om funktion, der er fremstillet i dette projekt kan bruges i stedet for hvordan man normal vil udregne i et regneark, er en test blevet lavet af den, som kan ses i kapitel *Test af GPU-calculate* 7.

For at opsummere hvad der er blevet fundet i dette projekt, og kigget på om jeg har svaret på, om man kan bruge et regnearks program til at gøre det nemmere at kode til en GPU, er der en *Diskussion* 8. kapitel 9 er *Konklusion*.

Fremtidige Arbejde 10 kigger på, hvad der er af muligheder for at videre udvikle på funktion lavet til *Corecalc and Funcalc* og nogen teste ville være interessant at foretage.

2 Relateret Arbejde

Det tidligere projekt [5] blev der kigget på hvad forskellen mellem CPU og GPU, de forskellige metoder til at kode til en GPU. Metoden der blev brugt til at teste det med, var ved at fremstille den samme funktion som et *benchmark* i de forskelle metoder at udregne på, Funktion der blev brugt var *k-means clustering*. Resultatet blev at der er mange måder at programmerer til en GPU, men for at få noget ud af det, skal men havde erfaring og viden ingen for dette område. I projektet blev der kigget på *CUDA*, *AMP*, *F# Brahma*, *OpenCL* og *Harlan*.

Artiklen *THE GPU COMPUTING ERA*[8] giver et indblik over hvordan *NVIDIA* GPU'er har udviklers sig, samt hvordan man kan kode til dem gennem årende.

Artiklen *A Survey of CPU-GPU Heterogeneous Computing Techniques*[6] beskriver, hvad der er forgået inden for *Heterogene Computing* Teknikker i den videnskabelige ramme. Noget af det der er omtalt er partitonering af arbejdsbyrde for at udnytte CPU'er og GPU'er til at forbedre ydeevnen og/eller energieffektivitet. Der bliver også beskrevet *benchmark* der kan bruges til at evaluere Heterogene computersystemer.

For at gøre noget ved den dårlige beskrivelse af sprogene, der er ved GPU programmering har artiklen *GPU Concurrency: Weak Behaviours and Programming Assumptions*[1] undersøgt dem. Disse beskrivelsers af sprogene føre til at mange programmører bliver nød til at bruge antagelser ved programmering af software, hvor der gøres brug af GPU. Har denne artikel gennemført en undersøgelse af flere GPU, ved at bruge *litmus* tests. Beskrivelser i programmerings guider og de garantier som leverandørens dokumentation om hardware giver, vil blive undersøgt i omtalte artikel.

Det er ikke kun et problem at kode til en GPU, man skal også kunne vide hvordan information skal gemmes for at få det meste ud af en GPU, som bliver beskrevet i artiklen Artiklen *Adaptive Input-aware Compilation for Graphics Engines*[11].

Til at hjælpe med at fremstille resultater der kunne bruges igennem test af udregnings tid, er Papiret *Microbenchmarks in Java and C#*[13] blevet brugt til at hjælpe med at fremstille test programmer, der kunne give resultater angående den brugte tid på at løbe en funktion igennem.

Bogen *Spreadsheet Implementation Technology: Basics and Extensions*[12] er en samling af information inden for programmering af regneark, for at hjælpe programmører der skal i gang med at udvikle til regne ark.

Web bloggen *Collaboration and Open Source at AMD: LibreOffice* [3] beskriver et sammenarbejde mellem dem der har lavet *LibreOffice* og firmaet *AMD*, hvor formålet var at tillade at overføre udregninger i regneark programmet *LibreOffice* til GPU ved at bruge OpenCL. Denne kode blev fremstillet af *AMP* ansatte.

Spreadsheet optimisation on GPUs [2] er et bachelor projekt som kigger på om det er muligt at bruge GPU til at udregne parallelt i et regnearks program. Programmet som der vil blive udvide på i dette projekt er også *CoreCalc*.

Som dette projekt har planer om at prøve at fuldføre. Her har de brugt *LibreOffice* som regneark programmet til udvikle på og der bliver brugt *OpenCL* for programmering af GPU.

3 Matrix Multiplatikon

For at give et 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 igennem. Der er lavet to versioner for hver GPU biblioteker, der er blevet kigget på. Forskellen mellem versionerne er, at den ene gør brug af en dimension og den anden bruger to dimensioner for array. De biblioteker der er blevet tested er *C++ AMP*, *CUDA* og *CUDAfy*.

Testene bliver gjort for at give et bedre indblik på de forskellige valgte metoder til at kode til en GPU, om 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#.

Test metoden er blevet fremstillet efter artiklen *Microbenchmarks in Java and C#* [13]. Af de versioner af test artiklen beskriver bruges der *Mark4* version.

Koden der bliver gået igennem er *CUDAfy* med en dimension *array*. På figure 1 kan del et af koden for *Main* funktion ses. *testSize* er de forskellige størrelser der vil blive taget tid på, eksempelvis når den tester med værdien 20, vil den lave 2 matrixer der har størrelsen 20 X 20. Siden koden der bliver vist her, er for en dimension *array*, vil den lave *array* med *Size1d*, der er *Size* ganget med sig selv. værdierne der bliver lavet på linje 3 styrer hvor mange gange *Mark4*, variablen *n* er hvor mange gange der skal tages tid på den samme størrelses og *count* er hvor mange gange funktion skal gøres mens der tages tid.

For løkken der mellem på linje 5 og 22, bruges til at køre *testSize* og lave en Mark 4 test med matrixer med størrelsen beskrevet i *testSize*. Fra linje 7 til 16 vil de to *array* simpel blive lavet, disse *array* bruges til input for matrix multiplatikon. På linje 18 vil *Mark4* blive kaldt, den vil returner to tal empirisk middelværdi og standard afvigelsen, der vil blive lagt ned i et *array result*, der kommer til at holde resultaterne fra alle testende for dette program, samt størrelsen der blev testet.

```
int[] testSize = new int[] { 5, 10, 20, 50, 100, 200, 300,
    400, 500, 600, 700, 800, 900, 1000 };
            double[,] result = new double[testSize.Length,
                3];
            int i, n = 10, count = 100;

            for (i = 0; i < testSize.Length; i++)
            {
                int Size = testSize[i];
                int Size1d = Size * Size;
                int[] A = new int[Size1d];
                int[] B = new int[Size1d];
                int[] C = new int[Size1d];
                for (int x = 0; x < (Size1d); x++)
                {
                    A[x] = 2;
                    B[x] = 3;
                }
                Console.WriteLine(testSize[i] + " starting
                    ");
                double[] Mark4_time = Mark4(A, B, C, Size,
                     Size1d, n, count);
                result[i, 0] = testSize[i];
                result[i, 1] = Mark4_time[0];
                result[i, 2] = Mark4_time[1];
            }
```

Fig. 1: Første del af *Main* for CUDAfy matrix multiplatikon med en dimension *array*.

Det næste til der vil blive beskrevet er *Mark4* som kan ses på figure 2. Fra linje 6 til 12 bliver GPU information fundet og gemt, samt fundet ud hvor mange tråde der max kan være i en block. Denne information bliver lavet inden målingen starter, siden denne information kan laves starten af programmet og blive genbrugt. Fra linje 14 til linje 23 er hovede delen i *Mark4*, her vil *MA*, forkortelse for *matrix multiplikation Algoritme*, blive testet igennem og taget tid på ved at bruge *timer* klassen der kan ses påkode snippet3.

Funktion *MA* kan ses på figure 4. på linjerne 4 til 6 kan det ses at der bliver allokeret plads på GPU, på linje 9 og 10 bliver data flyttet over på GPU, fra linje 12 til 24 vil den finde ud af hvor tråde og blokke der skal bruges for at kunne gennemgå funktions input. På linje 27 vil GPU funktion blive kørt, hvorefter på linje 30 vil man flytte resultatet fra GPU tilbage til CPU og til sidst vil *array* der er blevet allokeret blive frigivet.

På kode snittet 5 kan koden der bruges på GPU ses. Måden at en tråd finder dens ide på, er ved at gøre brug af *thread.threadIdx*. Men dette er ikke nok fordi *thread.threadIdx* giver dens id i den block den nu befinder sig i, Derfor skal der også bruges *thread.blockIdx* der giver id på den block tråden befinder sig i.

For at gøre det mere overskueligt at kigge på resultaterne vil *Main* part 2 lave en tekst fil med resultaterne i og gemme filen, kode snippet 6 viser denne del.

```
1 public static double[] Mark4(int[] A, int[] B, int[] C,
      int Size, int Size1d, int n, int count)
2         {
3             double dummy = 0.0;
4             double st = 0.0, sst = 0.0;
5
6             CudafyModule km = CudafyTranslator.Cudafy();
7
8             GPGPU gpu = CudafyHost.GetDevice(CudafyModes.
                  Target, CudafyModes.DeviceId);
9             gpu.LoadModule(km);
10
11            GPGPUProperties GPU_prop = gpu.
                  GetDeviceProperties();
12            int max_threadsPerBlock = GPU_prop.
                  MaxThreadsPerBlock;
13
14            for (int j = 0; j < n; j++)
15            {
16                Timer t = new Timer();
17                for (int i = 0; i < count; i++)
18                    dummy += MA(A, B, C, Size, Size1d, gpu
                          , max_threadsPerBlock);
19                double time = t.Check() / count;
20                st += time;
21                sst += time * time;
22            }
23            double mean = st / n, sdev = Math.Sqrt((sst -
                  mean * mean * n) / (n - 1));
24            return new double[2] { mean, sdev };
25        }
```

Fig. 2: *Mark4* klassen der bliver brugt til at teste med.

```
1 // timer class taken from the paper: Microbenchmarks in
     Java and C# by Peter Sestoft (sestoft@itu.dk) IT
     University of Copenhagen, Denmark
2     // plan on using Mark4 for tests
3     class Timer
4     {
5         private readonly System.Diagnostics.Stopwatch
              stopwatch = new System.Diagnostics.Stopwatch()
              ;
6         public Timer() { Play(); }
7         public double Check() { return stopwatch.
              ElapsedMilliseconds; }
8         public void Pause() { stopwatch.Stop(); }
9         public void Play() { stopwatch.Start(); }
10    }
```

Fig. 3: Timer klassen taget fra artiklen *Microbenchmarks in Java and C#* [13].

```
1 public static int MA(int[] A, int[] B, int[] C, int Size,
      int Size1d, GPGPU gpu, int max_threadsPerBlock)
2         {
3             // allocate the memory on the GPU
4             int[] GPU_A = gpu.Allocate<int>(A);
5             int[] GPU_B = gpu.Allocate<int>(B);
6             int[] GPU_C = gpu.Allocate<int>(C);
7
8             // copy the arrays 'a' and 'b' to the GPU
9             gpu.CopyToDevice(A, GPU_A);
10            gpu.CopyToDevice(B, GPU_B);
11
12            int threadsPerBlock = 0;
13            int blocksPerGrid = 0;
14
15            if (Size1d < max_threadsPerBlock)
16            {
17                threadsPerBlock = Size1d;
18                blocksPerGrid = 1;
19            }
20            else
21            {
22                threadsPerBlock = max_threadsPerBlock;
23                blocksPerGrid = (Size1d /
                      max_threadsPerBlock) + 1;
24            }
25
26            // launch GPU_MA
27            gpu.Launch(threadsPerBlock, blocksPerGrid).
                  GPU_MA(GPU_A, GPU_B, GPU_C, Size, Size1d);
28
29            // copy the array 'c' back from the GPU to the
                  CPU
30            gpu.CopyFromDevice(GPU_C, C);
31
32            gpu.Free(GPU_A);
33            gpu.Free(GPU_B);
34            gpu.Free(GPU_C);
35            return 1;
36        }
```

Fig. 4: *matrix multiplikation Algoritme* funktion.

```
[Cudafy]
        public static void GPU_MA(GThread thread, int[]
            GPU_A, int[] GPU_B, int[] GPU_C, int Size, int
             Size1d)
        {
            int i = thread.threadIdx.x + thread.blockDim.x
                 * thread.blockIdx.x;

            if (i < Size1d)
            {
                GPU_C[i] = 0;
                int x = i / Size;
                int y = i % Size;
                for (int z = 0; z < Size; z++)
                {
                    GPU_C[i] += GPU_A[(x * Size) + z] *
                        GPU_B[(z * Size) + y];
                }
            }
        }
```

Fig. 5: GPU funktion til *matrix multiplikation Algoritme*.

```
1 string lines = "CUDAfy 1D MA in C Sharp   mean   , sdev \r\n
    ";
2             for (i = 0; i < testSize.Length; i++)
3             {
4                 lines = lines + "size: " + result[i, 0] +
                      " time: " + result[i, 1] + " " +
                      result[i, 2] + "\r\n";
5             }
6
7             // Write the string to a file.
8             string path = @"c:\result\
                  CUDAfy_1D_MA_in_C_Sharp.txt";
9             System.IO.StreamWriter file;
10            if (!System.IO.File.Exists(path))
11            {
12                file = System.IO.File.CreateText(path);
13
14            }
15            else
16            {
17                file = new System.IO.StreamWriter(path);
18            }
19            file.WriteLine(lines);
20            file.Close();
```

Fig. 6: Anden del af *Main* for *CUDAfy* matrix multiplatikon med en dimension *array*.

4 Test af Matrix Multiplatikon

I denne afsnit vil resultaterne af testende på en computer med udstyret blive vist:

- Styresystem: Windows 10 Pro N
- Processor: intel core i5-4690K
- Hukommelse (RAM): 12 GB
- Skærmkort: NVIDIA GeForce GTX 960

4.1 Resultater fra Programmerne

Resultater vil blive delt op efter hvad tiden er blevet målt i, C# og C++. Dette er gjrot på grund af, at C# har muligheden for at måle i tid ved at bruge *System.Diagnostics.Stopwatch*.

Ved C++ kan man dog ikke måle tid i sekunder, efter hvad jeg har undersøgt. Man kan bruge biblioteket *time.h*, problemet med dette biblioteket er, at det måler i *clock ticks*, der er en tidsenhed af en konstant, med en system specifik længde. Der findes en macro *CLOCKS_ PER_ SEC*, som burde gøre arbejde, men med mine test kan kan jeg ikke se, hvordan det passer sammen med, hvad jeg får med mine C# målinger. For at få nogen målinger der passer forholdsvis med hinanden, er der prøvet at gøre så *AMP* i C++ tid er på samme decimal, som *AMP* når det bliver brugt igennem C#.

I figure 7 kan en tabel ses over målingerne fra C# ses og I figure 8 kan målingerne fra C++ ses. *C++ AMP* med to dimensioner array skete der en stack overflow da array blev på størrelse 300 x 300 eller over. Dette kun godt lade sig gøre for C# version fordi, C# bliver nød til at lave to dimension array om til en dimension for at kunne sende dem til C++ koden. Hvorefter at *AMP* vil allokerer dem som to dimensioner igen på GPU.

4.2 resultat af testen

Ud fra testen virker det som om at *CUDAfy* er et godt valg til at kode i, når det skal gøres i C# . Derfor vil GPU funktion der laves i dette projekt bruge *CUDAfy* til at lave GPU delen med.

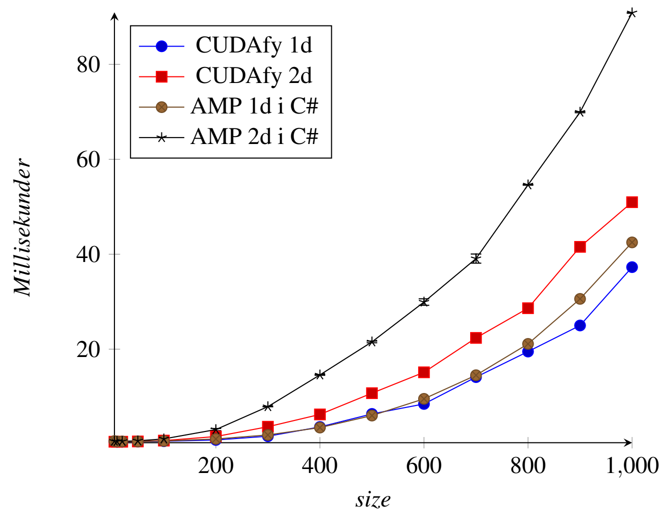

Fig. 7: alle målinger der er lavet i C#

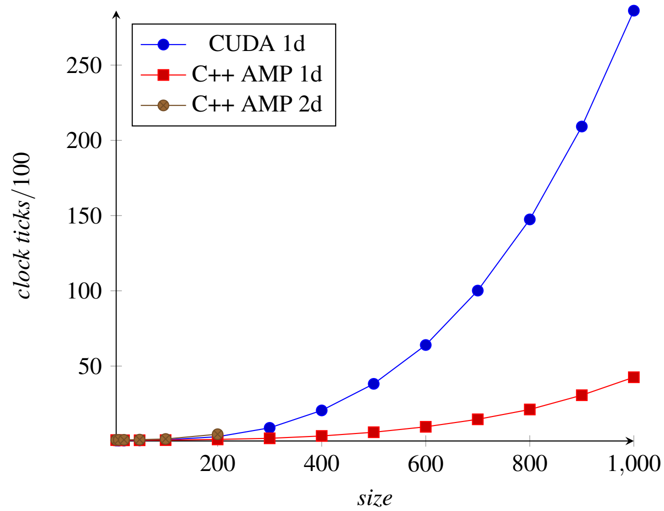

Fig. 8: samlet C++

5 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 man 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 *CoreCalc* udtryk/funktion, hvor talende i udtrykket peger til hvilken kolonne i den data den skal indsætte på dette placering i udregningen.

På figure 9 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 et udtryk (1+2), som beskriver til *GPU* at kolonne 1 skal plusset(+) 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 også det markeret område har lige så mange rækker som input har.

For at lave GPU funktion, vil klassen *Function* i *Corecalc and Funcalc* blive modificeret for at kunne kalde GPU funktion. I figure 10 kan et klasse diagram ses over klasserne i *Corecalc and Funcalc*. pilen der er peger på *Function* klassen.

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. 9: et billede af *CoreCalc* hvor *GPU* bliver brugt.

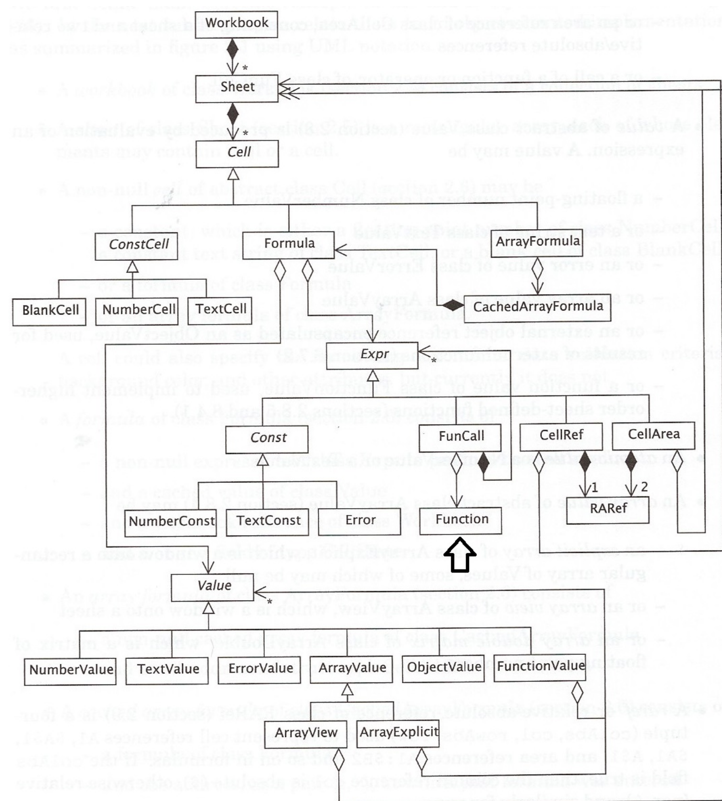

Fig. 10: Grafs over klasserne i *Corecalc and Funcalc*, taget fra bogen *Spreadsheet optimisation on GPUs* [?] side 30. pilen peger på klassen der vil blive lavet i, for at kunne bruge GPU funktion.

6 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 [10]. *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 lavet til at 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, har jeg også tilført noget kode i klassen *Function* i *Corecalc*.

6.1 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, at det tager en god potion tid at hente denne information. Konstruktøren og de globale variabler kan ses i figure 11, *tempResult* bliver brugt i *makeFuncHelper* til at styre hvad for nogen midlertidig variable er i brug.

```
1  class GPU_func
2      {
3          private CudafyModule km;
4          private GPGPU gpu;
5          private GPGPUProperties GPU_prop;
6          private List<int> tempResult;
7
8          public GPU_func()
9          {
10             km = CudafyTranslator.Cudafy();
11
12             gpu = CudafyHost.GetDevice(CudafyModes.Target,
                   CudafyModes.DeviceId);
13             gpu.LoadModule(km);
14
15             GPU_prop = gpu.GetDeviceProperties();
16         }
```

Fig. 11: konstruktøren for *GPU_func* samt de globale værdier der bliver brugt i klassen

6.2 makeFunc

makeFunc og *makeFuncHelper* funktionerne bruges til at fremstille en indkodning af hvordan hvordan GPU'en skal udregne, ud fra at abstrakt syntaks træ af et udtryk.

Denne liste har X antal a fire tal som er en enkle udregning (+,-,*,/) med to værdier, X er hvor mange udregning er skal gøres i alt, for at komme til resultatet som udtrykket vil give.

Tallene i den enkle udregning har en bestemt mening, 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.

makeFuncHelper kan ses i fem dele. Første del kan ses på figure 12 hvor den finder ud af om input variabler er en funktion eller en værdi. Det skal pointeres at *makeFuncHelper* er en rekursiv funktion, der er lavet for at gå igennem det abstrakt syntaks træ nemt.

```
1 private List<List<int>> makeFuncHelper(FunCall input, bool
      root)
2         {
3             List<List<int>> temp = new List<List<int>>();
4             int locationOne = 0, locationTwo = 0; // used
                  to hold the temp locating of the result
5             // find where to place output
6
7             bool oneIsFunc = (input.es[0] is FunCall);
8             bool oneIsNumber = (input.es[0] is NumberConst
                  );
9             bool twoIsFunc = (input.es[1] is FunCall);
10            bool twoIsNumber = (input.es[1] is NumberConst
                  );
```

Fig. 12: Første del af *makeFuncHelper* som kigger på variabler af input.

Anden del kan ses på figure 13, denne del arbejder med inputs variablerne. Fra linje 1 til 13 er for første værdi, hvis denne værdi er en funktion, vil den kalde sig selv med funktion som input, Hvis det er en værdi, vil den tage værdien og ligge den ned i *locationOne*, som er værdien der holder styr på, hvor *GPUFunc* skal tage information fra for hver udregning trin. Det samme sker så for anden inputs variablerne fra linje 14 til 27.

Den tredje kigger på hvilket form for udregning/operation der bliver gjort i input, dette kan ses i figure 14

Fjerde del der kan ses på figure 15 stater med at lave en liste *result*, der bliver brugt til holde indkodning af input i dette point i det abstraktet syntaks træ.

Hvis der har været brugt midlertidig variable for dens input variabler, vil den fjerne dem fra listen fra brugte midlertidig variable liste *tempResult*. Dette er gjort for at spare på pladsen på GPU ved at genbruge midlertidig variable placeringer, hvor det er muligt.

```
if (oneIsFunc)
{
    temp.AddRange(makeFuncHelper(input.es[0]
        as FunCall,false));
    locationOne = temp[temp.Count - 1][3];
}
else if(oneIsNumber)
{
    locationOne = (int)Value.ToDoubleOrNan((
        input.es[0] as NumberConst).value);
}
else
{
    // some kind of error...
}
if (twoIsFunc)
{
    temp.AddRange(makeFuncHelper(input.es[1]
        as FunCall, false));
    locationTwo = temp[temp.Count - 1][3];

}
else if (twoIsNumber)
{
    locationTwo = (int)Value.ToDoubleOrNan((
        input.es[1] as NumberConst).value);
}
else
{
    // some kind of error...
}
```

Fig. 13: Anden del af *makeFuncHelper* der laver arbejdet med inputs variabler.

Den femte og sidste del 16 i *makeFuncHelper* gemmer hvad de tidligere dele har fundet ind i listen *result* og returner *temp* der er en liste af lister, efter at have lagt *result* på den.

For at give et eksempel på hvorden *makeFunc* gør, 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. Eg bliver *(1+2)*(3+4)* til en liste der kan se sådan ud

- 1,1,2,-1
- 3,1,4,-2

```
1               int functionValue = 0; ;
2               string function = input.function.name.ToString
                    ();
3               switch (function)
4               {
5                   case "+":
6                       functionValue = 1;
7                       break;
8                   case "-":
9                       functionValue = 2;
10                      break;
11                  case "*":
12                      functionValue = 3;
13                      break;
14                  case "/":
15                      functionValue = 4;
16                      break;
17                  default:
18                      // some kind of error...
19                      break;
20              }
```

Fig. 14: Tredje del af *makeFuncHelper* finder ud af hvad for en operation der sker i input.

- -1,3,-2,0

6.3 findnumberOfTempResult

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

6.4 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 på GPU når indkodning af udtrykket bliver læst igennem, grundet dette bliver gjort her, er at antal af midlertidig resultater der skal gemmes i en funktion kan variere alt efter hvor kompliceret den er og efter hvad der er fundet på nettet, giver CUDAfy ikke mulighed for at lave et *array* på run time på GPU tråde lokale hukommelse.

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

```
1                List<int> result = new List<int>();
2
3                if (oneIsFunc)
4                {
5                    tempResult.RemoveAt(tempResult.
                         FindLastIndex(x => x == locationOne));
6                }
7                if(twoIsFunc)
8                {
9                    tempResult.RemoveAt(tempResult.
                         FindLastIndex(x => x == locationTwo));
10               }
11
12               int outputPlace = 0;
13               if(!root)
14               {
15                   int x = -1;
16                   while (outputPlace == 0)
17                   {
18                       if(!tempResult.Contains(x))
19                       {
20                           outputPlace = x;
21                       }
22                       else
23                       {
24                           x--;
25                       }
26                   }
27               }
28               tempResult.Add(outputPlace);
```

Fig. 15: Fjerde del af *makeFuncHelper* fjerner brugt midlertidig variabler placeringer for genbrug og finde ud af hvor inputs resultat skal placeres.

```
1                result.Add(locationOne);
2                result.Add(functionValue);
3                result.Add(locationTwo);
4                result.Add(outputPlace);
5
6                temp.Add(result);
7
8                return temp;
```

Fig. 16: Femte del af *makeFuncHelper* sætter alt sammen og sender hvad den har fundet ud af tilbage.

Derefter vil alle *array* få allokeret plads på GPU. De *array* der bliver sent til GPU, vil være dem med information om input data og indkodning af udtrykket, *array* der er blevet allokeret for de midlertidig resultater vil ikke blive send til GPU, da dette *array* ikke holder information fra CPU, som GPU skal bruge.

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.

6.5 GPUFunc

GPUFunc er funktion der vil blive kørt på GPU. Den har tre ting den gør for hver punkt i indkodningen. 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. Koden har meget til fælles med *makeFuncHelper*, på grund af den læser indkodningen og udregner alt efter, hvad indkodningen beskriver i sted for at lave en indkodningen.

6.6 kode i Corecalc Function klasse

Koden der lavet i *Function* klassen i *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. De linjer der er sat ind kan ses på figure 17.

```
1                GPU = new GPU_func();
2                .
3                .
4                .
5                new Function("GPU", GPUFunction()); //
                     GPUFUNC
```

Fig. 17: Ting der er lavet i *Function* konstruktøren.

Den sidste del af koden i denne klasse ligger i *GPUFunction*, kan ses i figure 18, koden her er samlelede mellem *Corecalc* og *GPU_ func*. Kodens 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 indkodning. Med indkodningen 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.

```
1     private static Applier GPUFunction()
2     {
3         return
4           delegate(Sheet sheet, Expr[] es, int col, int
                row)
5           {
6               if (es.Length == 2)
7               {
8                   Value v0 = es[0].Eval(sheet, col, row);
9
10                  if (v0 is ErrorValue) return v0;
11                  ArrayValue v0arr = v0 as ArrayValue;
12                  if (v0arr != null)
13                  {
14                      int rows = v0arr.Rows;
15                      double[,] input = ArrayValue.
                            ToDoubleArray2D(v0arr);
16                      int[,] function = GPU.makeFunc(es[1]
                             as Corecalc.FunCall);
17                      double[] output = GPU.calculate(
                            input, function);
18
19                      Value[,] result = new Value[1, rows
                            ];
20
21                      for (int r = 0; r < rows; r++)
22                          result[0,r] = NumberValue.Make(
                                output[r]);
23
24                      return new ArrayExplicit(result);
25                  }
26                  else
27                      return ErrorValue.argTypeError;
28              }
29              else
30              {
31                  return ErrorValue.argCountError;
32              }
33          };
34    }
```

Fig. 18: *GPUFunction* der bliver brugt *Function* konstruktøren.

7 Test af GPU-calculate

Der er lavet 4 test inden for *Funcalc*. Den første test er kolonne *A* fyldt med konstanter, tallet 2, og kolonne *B* fyldt med konstanter, tallet 3. I kolonne *C* er der hvor udregningen

vil ske. Kolonnerne har max 100 rækker, derfor er der 1000 tal eller udregninger i hver kolonne. Denne test starter med *A1*B1* og for hver *x* bliver der liget et ekstra *+A1*B1* på udregningen. Funktion er sådan ud:

- f(1) = (A1*B1)
- f(x) = (A1*B1) + f(x-1)

I anden test er kolonnerne *A* og *B* fyldt med tallet 2 og kolonnerne *C* og *D* fyldt med tallet 3. Kolonne *E* bliver brugt til at holde udregningerne i, hvor denne test starter med Denne test starter med *A1*B1*C1*D1* og for hver *x* bliver der liget et ekstra *+A1*B1*C1*D1* på udregningen. Funktion er sådan ud:

- f(1) = (A1*B1*C1*D1)
- f(x) = (A1*B1*C1*D1) + f(x-1)

Tredje test er kolonnerne *A*, *B* og *C* fyldt med tallet 2 og kolonnerne *D*, *E* og *F* fyldt med tallet 3. Kolonne *E* bliver brugt til at holde udregningerne i, hvor denne test starter med denne test starter med *A1*B1*C1*D1*E1*F1* og for hver *x* bliver der liget et ekstra *+A1*B1*C1*D1*E1*F1* på udregningen. Funktion er sådan ud:

- f(1) = (A1*B1*C1*D1*E1*F1)
- f(x) = (A1*B1*C1*D1*E1*F1) + f(x-1)

For den fjerde test blev der kigget på om størrelsen af data kunne gøre at GPU kunne blive bedre. kolonnerne *A* til *S* blev flydt med konstanter fra start og kolonne *T* brues til at holde regnestykket Funktion er sådan ud:

- f(1) = A
- f(2) = B*f(n-1)
- f(3) = C*f(n-1)
- f(4) = D*f(n-1)
- f(5) = E*f(n-1)
- f(6) = F*f(n-1)
- f(7) = G*f(n-1)
- f(8) = H*f(n-1)
- f(9) = I*f(n-1)
- f(10) = J*f(n-1)
- f(11) = K*f(n-1)
- f(12) = L*f(n-1)
- f(13) = M*f(n-1)
- f(14) = N*f(n-1)
- f(15) = O*f(n-1)
- f(16) = P*f(n-1)
- f(17) = Q*f(n-1)
- f(18) = R*f(n-1)
- f(19) = S*f(n-1)

7.1 Resultater

I tabel 19 kan resultaterne fra første test ses, i tabel 20 kan resultaterne fra anden test ses, tabel 21 viser resultaterne fra tredje test og Tabel 22 viser resultanterne fra den fjerde test.

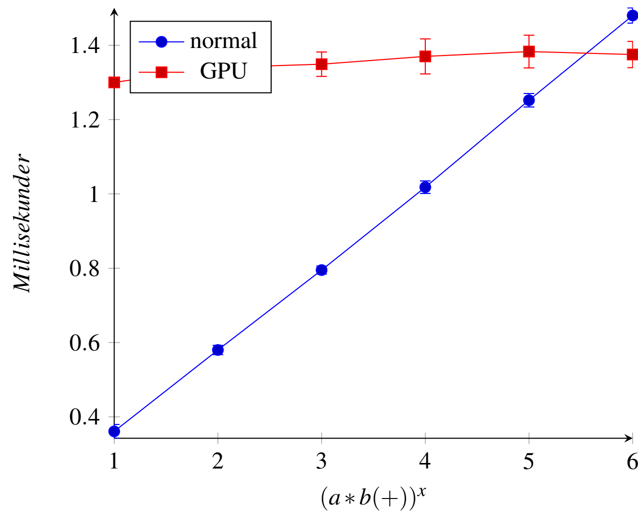

Fig. 19: Testen af Funcalc hvor kolonerne er fyldt ud, 1000 tal i vær kolonne, hvor *A=2*,*B=3* og C er hvor funktioner ligger. x står for hvor mange udredninger der er af (a1*b1(+)) og for GPU (1*2(+)) .

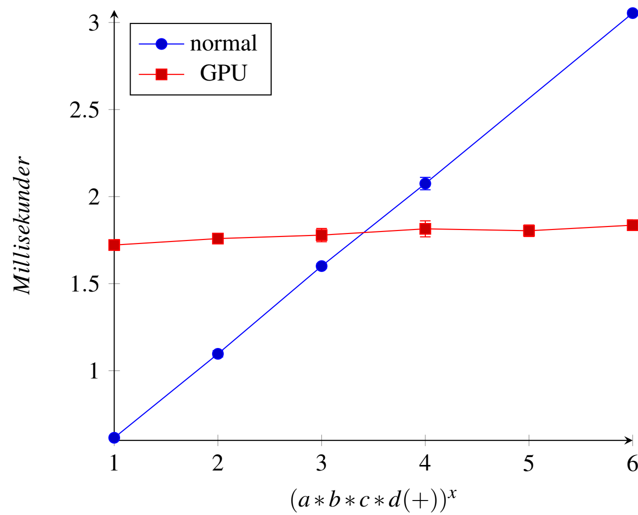

Fig. 20: Testen af Funcalc hvor kolonerne er fyldt ud, 1000 tal i vær kolonne, hvor *A=B=2*,*C=D=3* og E er hvor funktioner ligger. x står for hvor mange udredninger der er af (a1*b1*c1*d1(+)) og for GPU (1*2*3*4(+)).

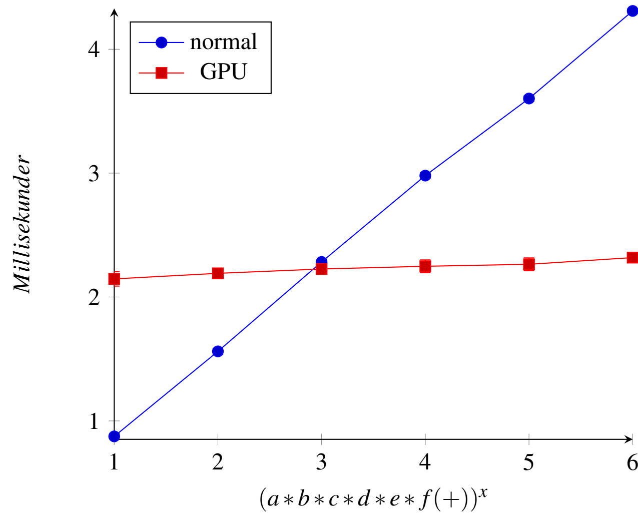

Fig. 21: testen af Funcalc hvor kolonerne er fyldt ud, 1000 tal i vær kolonne, hvor *A=B=C=2,D=E=F=3* og C er hvor funktioner ligger. x står for hvor mange af udredningen (a*b*c*d*e*f(+)) og for GPU (a*b*c*d*e*f(+)) .

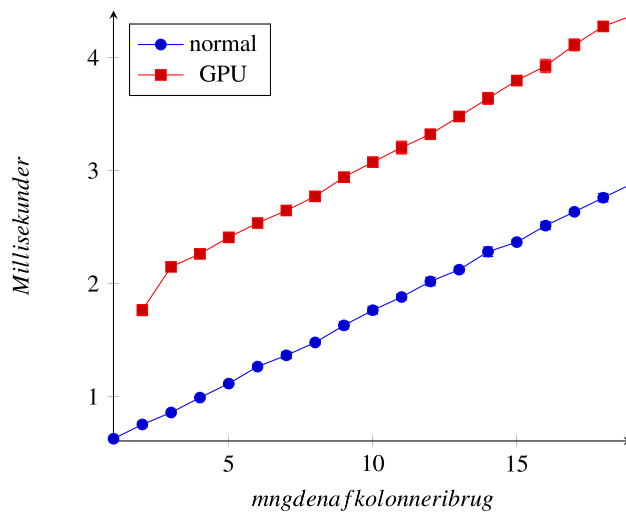

Fig. 22: Test af om mængde data der bliver arbejde på, har indflydelse på om GPU bliver bedre at arbejde på.

8 Diskussion

For at se på mulighederne at kode til en *GPU* igennem sproget C# , som er det sprog *Corecalc* er skrevet i, er der blevet lavet nogle forskellige teste, der gav nogle interessante resultater. Det kan ses på *C++ AMP* og *CUDA* i tabellen 8 at *C++ AMP* virker til at være meget bedre en *CUDA*, dette tilfælde kunne skylde at *C++ AMP* måske har en smule optimering af data til tråde, fordi i *C++ AMP* skal der ikke findes ud af hvor mange tråde og blokke der skal bruges til at køre en funktion, men i sted giver man et *array* man vil havde gået igennem på GPU. hvorimod i *CUDA* skal man selv programmerer, hvor mange tråde og blokke der skal bruges. En anden årsag til at *CUDA* havde dårlige målinger, vil være dårlig optimeret kode.

En overraskende ting kan observeres i *CUDAfy* 1d og 2d kan noteteres, at det er hurtigere, hvis håndterede data i 1d *array* i sted for 2d *array*, når matrixen begynder at blive størrelse på 10 eller over. Hvilket er det modsatte af hvad artiklen *Adaptive Input-aware Compilation for Graphics Engines*[11] beskriver.

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 man skal bruge. Hvilket godt kan give nogen problemer, når man arbejder med et program, der skal arbejde med varierende input. Efter hvad jeg har 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. Noget man også kunne kigge på er om artiklen *Adaptive Input-aware Compilation for Graphics Engines*[11] kunne bruges til at øge hvor effektiv GPU funktion er. For at løse problemet med varierede tråde mænge, har jeg fremstillet en generist metode, der finder ud af, hvor mange blokke der skal bruges, ved at hente information om GPU igennem funktioner som *CUDA* og *CUDAfy* stiller til rådighed. Denne information indeholder eksempelvis hvor mange tråde der maksimal kan være i en block på GPU. Hvor ved hvis en enkel block ikke kan holde det antal af tråde der skal bruges, vil denne metode finde antallet af blokke, der er nødvendig for at kunne holde den mængden af tråde der skal bruges i udregningen. Denne metode er dog ikke perfekt, siden der kunne laves noget optimering af hvordan data bliver gemt til GPU.

Igennem resultaterne af testene der kan ses i afsnit 4.1 for GPU funktion og den simple måde at lave funktioner til *Corecalc*, det kan ses fra testene at jo mere der skal udregnes jo bedre er det at gøre det på GPU, i stedet for at gøre det på den normal måde. Det kan ses på figure 22 at størrelsen af data ikke har den store indflydelse på om GPU bliver bedre at bruge. Dog kan det ses ud fra testene der er figure 19 og 20, GPU næsten er en vandret linje, hvorimod at den metoden at stille denne form for udregningen op, bliver til en linje med en stigning på cirka 45%. Derfor ville funktion, hvor man skal genbruge data meget, såsom matrix multiplikation virke effektiv på en GPU. Noget anden der kan tages fra testen med varierende størrelser af data mængde er, at igennem *Corecalc and Funcalc* er det ikke muligt at ramme for hvad GPU maksimal kan holde af data i et ark. Men da test af matrix multiplikation gav fejlen med ikke mere fri data på GPU, skal man tænke over at 3 *array* på størrelse på over 1023 x 1023 bliver allokeret på GPU og igennem *Corecalc and Funcalc* ark er det kun muligt af lave et *array* på 1000 x 19, siden det sidste kolonne skal bruges på at holde udregningen. Så ud fra testen ser man at, hvis udtrykket er stort, kan det betale sig at flytte udregningen over

på GPU. En løsningen på dette problem, hvis det skulle opstå, kunne være, at begynde på brug af billede hukommelse på GPU'en til readonly data. Hvilke skulle være muligt med *CUDA*, men desværre ikke skulle være muligt med *CUDAfy* på dette tidspunkt.

9 Konklusion

Formålet med dette projekt var, at prøve at gøre det det nemmere at kode til GPU, ved at bruge et regneark program til at kode til den. GPU funktion i *Corecalc* blevet lavet, men denne funktion er dog ikke blevet bruger testet, derved kan der ikke siges noget om det er blevet nemmere at kode til en GPU. Måden man skriver en funktion i *funcalc* funktions ark kunne være sværere for personer der har erfaring med regnearks programmer, men ikke har erfaring med at programmerer. Her kunne måden at skrive et udtryk man vil havde udregnet være nemmere for dem.

Testene af GPU funktion sammen med hvordan man normalt vil lave den samme form for udregning, viser at GPU funktion kan være hurtigere at udregne. Dog skal udregning være tilpas stor og have godt med data at arbejde med, for at kunne være et brugbar alternativ.

10 Fremtidige Arbejde

Første ting kunne være at lave lidt om på GPU funktion, så den kunne tage værdier der er den samme for alle udregninger, på dette tidspunkt kan GPU funktion kun arbejde med et *array* der består af input. Men hvis GPU funktion gav lov til at sende et ekstra *array* med globale værdier alle tråde skal bruge, kunne det hjælpe på overførsel tid og spare på pladsen på GPU. Dette kunne gøres ved at tillade, at når man giver lov til at markeret en celle ved bruge af GPU funktion, der sådan bliver til den global værdi, måden de globale værdier kunne pointet til i indkodning, ville være at bruge de positive tal der ikke bliver bugt for at beskrive input data. Eksempelvis kunne man skrive en GPU funktion sådan: $= GPU(A1 : B10, (1+2) * C1)$. her vil den lave et *array* over *A1:B10* hvor hver tråd har den data i og sende et *array* med *C1* som alle tråde vil tilgå, for at få fadt på denne værdi.

GPU funktinon kunne blive videreudvikle, så den kan håndterer *if()* og *rand()*. Måden den kunne klare problemet med *if()*, er ved at tilføre ekstra værdier ind ind i indkodningen. Eksempelvis kunne man se *if()* som en split i en udregning, vil den nye værdi indkodning holde styr på hvilken gren man befinder sig på. Derudover skal der også lavet så funktion på GPU, som er i stand til at udregne et boolean udtryk. Udvidelsen for at læse boolean udtryk burde ikke være det store problem, men udvidelsen for at håndtere *if()* er muligvis ikke liget til, siden GPU funktion skal kunne håndtere at hoppe i indkodning. *CUDAFY* ser ikke ud til at have en tilfældigheds funktion i sig, så derved kan *rand()* ikke gøres på GPU siden, man kunne lave nogen tilfældige tal og sende med til GPU hvis den skal et tilfældigt tal, dette ville dog sænke hurtigheden for GPU, siden at testene viser at, der helst skal være flere udregninger en mængde af data man sender, ville det ikke være hurtigere at gøre udregningerne på GPU. Dog ser det ud til at *CUDA* har en form for tilfældigheds generator med navn *cuRAND*, denne funktion kunne måske blive til rådighed for *CUDAfy* i fremtiden.

Et forslag ville være at GPU funktion kan tage en *funcalc* funktion, skrevet i et funktion ark, i sted for at kun kan tage et *CoreCalc* udtryk. Ved at gøre dette ville bruger venligheden muligvis blive bedre. Dette burde kunne lade sig gøre, da et abstrakt syntaks træ bliver lavet i funktion arket i *funcalc*.

En anden interessant ide kunne være, at give brugeren mulighed for at direkte bruge cellerne som punkt for, hvad der skal ligges sammen, GPU(a1:c5 , a1+b1+c1). Så skal GPU funktion selv finde ud af om de celler i udtrykket inden for input *array* eller om det er noget udenfor der, hvilket vil gøre det til en global værdi.

Unikke GPU funktioner kunne blive lavet over kendte algoritme problemer, en af dem kunne være det samme om der er blevet brug til at teste de forskellige sprog: matrix Multiplatikon, koden er næsten lavet til fulde, der mangler nogen små ændringer, såsom at tage matrixer, der ikke har den samme længe og højde, som input kunne denne funktion tag 2 forskellige markeret celler området.

Ideen ved dette projekt var at gøre det nemmere at kode til en GPU, dette blev aldrig testet. Derfor kunne det være interessant at lave en brugertest af GPU funktion for at se, hvad en bruger mener om funktion og om der muligvis kunne laves nogen forbedring på funktion, som ikke er kommet op under projektet arbejdes tid.

Noget andet der også kunne være interessant at teste, ville være om måden at at GPU funktion kan på sin vis *compile* simpel udtryk på run time, ved brug af en indkodning og læser, ville kunne klare sig mod projekter der compiler GPU kode på run time. Dette kunne være interessant at kigge på, for at se om der er hurtigere at lave GPU kode på run time eller om der hurtigere at lave en indkodning som en GPU kan udregne efter.

11 Acknowledgement

Jeg vil gerne takke Bent Thomsen for hans hjælp under projektet som vejleder igennem dette semester.

References

1. Jade Alglave, Mark Batty, Alastair F Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. Gpu concurrency: weak behaviours and programming assumptions. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 15). ACM, New York*, pages 577–591, 2015.
2. Tim Garbos and Kasper Videbæk. Spreadsheet optimisation on gpus. `http://www.itu.dk/garbos/gpu-spreadsheets/bachelor.pdf`, 2010.
3. jtrudeau. Collaboration and open source at amd: Libreoffice. `http://developer.amd.com/community/blog/author/jtrudeau/`, July 2015.
4. Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 451–460. ACM, 2010.
5. Nichlas Korgaard Møller and Niels Brøndum Pedersen. A comparative study of cpu and gpu programming in different languages and libraries, 01 2016. P9, Autumn Semester 2015.

6. Sparsh Mittal and Jeffrey S Vetter. A survey of cpu-gpu heterogeneous computing techniques. 2015.
7. Niels Brøndum Pedersen. Github repository for the projekt. `https://github.com/Dugin13/P10`.
8. John Nickolls and William J Dally. The gpu computing era. *IEEE micro*, (2):56–69, 2010.
9. Nathaniel Nystrom, Derek White, and Kishen Das. Firepile: run-time compilation for gpus in scala. *GPCE '11 Proceedings of the 10th ACM international conference on Generative programming and component engineering*, 2011.
10. IT University of Copenhagen. Corecalc and funcalc spreadsheet technology in c sharp. `http://www.itu.dk/people/sestoft/funcalc/`.
11. Mehrzad Samadi, Amir Hormati, Mojtaba Mehrara, Janghaeng Lee, and Scott Mahlke. Adaptive input-aware compilation for graphics engines. In *ACM SIGPLAN Notices*, volume 47, pages 13–22. ACM, 2012.
12. Peter Sestoft (sestoft@itu.dk). *Spreadsheet Implementation Technology: Basics and Extensions*. The MIT Press Cambridge Massachusetts, London, Enland, 2014.
13. Peter Sestoft (sestoft@itu.dk). Microbenchmarks in java and c sharp, 9 2015.
14. Andrew Stromme, Ryan Carlson, and Tia Newhall. Chestnut: A gpu programming language for non-experts. *PMAM '12 Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, 2012.