AALBORG UNIVERSITET
STUDENTERRAPPORT

Title:
CUDAfy og funcalc

Project period:
P10, forår 2016

Project Group:
?

Author:
Niels Brøndum Pedersen
Supervisor:
Bent Thomsen

Total Pages: - 24
Completion date: - 00-00-0000

Synopsis:

In this project, there have been conducted research on if it is possible for making it easier for programming to 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 on the could be useful time wise and the test showed if the expression that is being calculated is large enough or there is a large amount of data for the expression it could benefit from being calculated on the GPU.

1 Intro

Alt kode kan finde på min GitHub profil Dugin13, her er et link til selve projektet [8]

I tidligere Projekt [4] kiggede på forskellen mellem en CPU, der bliver brugt til hverdagen og som er god til sekventielle beregningen, og GPU, der for det meste buges 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 godt dokumentation og er kompliceret at lave. Et papir fundet i løbet af det projektet viste også at maget af tiden bliver brugt på at overføre data til GPU [2].

Derfor er formålet med dette projekt at lave en simple programmering metode for at kunne bruge en GPU, uden man skal havde de store viden inden for det. 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 ligge udregningen over på GPU. Regneark programment der blev brugt 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.

Inden der vil blive lavet på *Corecalc and Funcalc*, vil der blive 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, for at give et eksempel kunne være *Chestnut*[12]. 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 genskrive brugeren 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 leves 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 runtime. Der er dog nogen der har kigget på ideen med at compile GPU kode på runtime, såsom *Firepile* [6] der er en udvidelse til *Scala*

Et anden problem, dog mindre, var det ikke rigtig finde løsning på, hvordan et program kunne håndtere forskellige mængder af tråde på run time til en GPU, dette problem bliver løst på en meget simple måde der også kan klare forskellige GPU korts.

2 relateret arbejde

Det tidligere projekt [4] blev der kigget på hvad forskellen mellem CPU og GPU og 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*. Det blev fundet at der er mange måder at gøre GPU programmering, men for at få noget ud af det skal men havde faring og viden først. i prjektet blev der kigget på CUDA, AMP, F# Brahma OpenCL og Harlan.

Bogen *Spreadsheet Implementation Technology: Basics and Extensions*[10] af Peter Sestoft er en samling af information inden for programmering af regneark, for at hjælpe programmører der skal i gang med at udvikle til regne ark.

Artiklen *THE GPU COMPUTING ERA*[5] 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*[3] kigger på hvad der er sket inden for den videnskabelige ramme inden for Heterogene Computing Teknikker, såsom partitonering af arbejdsbyrde for at udnytte CPU'er og GPU'er til at forbedre ydeevnen og/eller energieffektivitet. Der bliver også kigge på benchmark der kan bruges til at evaluere Heterogene computersystemer.

For at gøre noget ved den dårlige beskrivelse af sprog der er ved GPU programmering har artiklen*GPU Concurrency: Weak Behaviours and Programming Assumptions*[1] kigget på dem. Disse beskrivelsers af sprog føre til at mange programmører bliver nød til at bruge antagelser, når der laves software til GPU. Har dette papir gennemført en undersøgelse af GPU'er, Ved at bruge *litmus* tests. Der bliver kigget på antagelser i programmering guider og leverandør dokumentation om de garantier, som hardware giver.

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*[9].

Til at hjælpe med at fremstille resultater der kunne bruges, er Papiret *Microbenchmarks in Java and C#*[11] blevet brugt til at hjælpe med at fremstille test programmer, der kunne give resultater angånde den brugte til på at løbe en funktion igennem.

3 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++.

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#.

Testene er blevet fremstillet efter papiret *Microbenchmarks in Java and C#* [11]. Af de versioner af test papiret beskriver bruges der Mark4 version.

Koden der bliver gået igennem er CUDAfy med en dimension array. På kode snippet 1 kan del et af *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,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 simple blive lavet, som vil blive lagt sammen. På linje 18 vil *Mark4* blive kald, 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.

```
1  int[] testSize = new int[] { 5, 10, 20, 50, 100, 200, 300,
       400, 500, 600, 700, 800, 900, 1000 };
2              double[,] result = new double[testSize.Length,
                   3];
3              int i, n = 10, count = 100;
4
5              for (i = 0; i < testSize.Length; i++)
6              {
7                  int Size = testSize[i];
8                  int Size1d = Size * Size;
9                  int[] A = new int[Size1d];
10                 int[] B = new int[Size1d];
11                 int[] C = new int[Size1d];
12                 for (int x = 0; x < (Size1d); x++)
13                 {
14                     A[x] = 2;
15                     B[x] = 3;
16                 }
17                 Console.WriteLine(testSize[i] + " starting
                       ");
18                 double[] Mark4_time = Mark4(A, B, C, Size,
                       Size1d, n, count);
19                 result[i, 0] = testSize[i];
20                 result[i, 1] = Mark4_time[0];
21                 result[i, 2] = Mark4_time[1];
22             }
```

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

næste til der vil blive beskrevet er *Mark4* som kan ses på kode snippet 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 lave af 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.

```
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.

Funktion *MA* kan ses på kode snippet4. 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.

på linje 4 bli

```
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#* [11].

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.

```
public static int MA(int[] A, int[] B, int[] C, int Size,
    int Size1d, GPGPU gpu, int max_threadsPerBlock)
        {
            // allocate the memory on the GPU
            int[] GPU_A = gpu.Allocate<int>(A);
            int[] GPU_B = gpu.Allocate<int>(B);
            int[] GPU_C = gpu.Allocate<int>(C);

            // copy the arrays 'a' and 'b' to the GPU
            gpu.CopyToDevice(A, GPU_A);
            gpu.CopyToDevice(B, GPU_B);

            int threadsPerBlock = 0;
            int blocksPerGrid = 0;

            if (Size1d < max_threadsPerBlock)
            {
                threadsPerBlock = Size1d;
                blocksPerGrid = 1;
            }
            else
            {
                threadsPerBlock = max_threadsPerBlock;
                blocksPerGrid = (Size1d /
                    max_threadsPerBlock) + 1;
            }

            // launch add on N threads
            gpu.Launch(threadsPerBlock, blocksPerGrid).
                GPU_MA(GPU_A, GPU_B, GPU_C, Size, Size1d);

            // copy the array 'c' back from the GPU to the
                 CPU
            gpu.CopyFromDevice(GPU_C, C);

            gpu.Free(GPU_A);
            gpu.Free(GPU_B);
            gpu.Free(GPU_C);
            return 1;
        }
```

Fig. 4: *matrix multiplikation Algoritme* funktion.

```
1  [Cudafy]
2          public static void GPU_MA(GThread thread, int[]
               GPU_A, int[] GPU_B, int[] GPU_C, int Size, int
                Size1d)
3          {
4              int i = thread.threadIdx.x + thread.blockDim.x
                   * thread.blockIdx.x;
5
6              if (i < Size1d)
7              {
8                  GPU_C[i] = 0;
9                  int x = i / Size;
10                 int y = i % Size;
11                 for (int z = 0; z < Size; z++)
12                 {
13                     GPU_C[i] += GPU_A[(x * Size) + z] *
                           GPU_B[(z * Size) + y];
14                 }
15             }
16         }
```

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

```
string lines = "CUDAfy 1D MA in C Sharp  mean  , sdev \r\n
    ";
            for (i = 0; i < testSize.Length; i++)
            {
                lines = lines + "size: " + result[i, 0] +
                    " time: " + result[i, 1] + " " +
                    result[i, 2] + "\r\n";
            }

            // Write the string to a file.
            string path = @"c:\result\
                CUDAfy_1D_MA_in_C_Sharp.txt";
            System.IO.StreamWriter file;
            if (!System.IO.File.Exists(path))
            {
                file = System.IO.File.CreateText(path);

            }
            else
            {
                file = new System.IO.StreamWriter(path);
            }
            file.WriteLine(lines);
            file.Close();
```

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

4 Test af Matrix Multiplatikon

I denne det 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 igennem, C# og C++. Dette er gjort fordi at C# har muligheden for at måle i tid, kan man ikke rigtig måle i tid i C++. I C++ kan man bruge biblioteket *time.h*, problemet med dette biblioteket er, at det måler i *clock ticks*, der er en tidsenhed af en konstant, men systemspecifikke 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.

På tabel 7 kan målingerne fra C# ses og på tabel 8 kan målingerne fra C++ ses. C++ AMP med to dimensioner array skete der en stack overflow da array blev på størrelsen 300 x 300 eller over.

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.

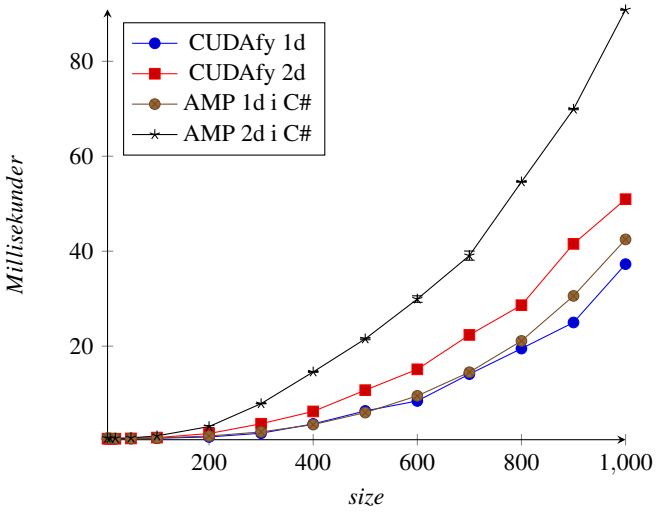

Fig. 7: alle målinger der er lavet i C#

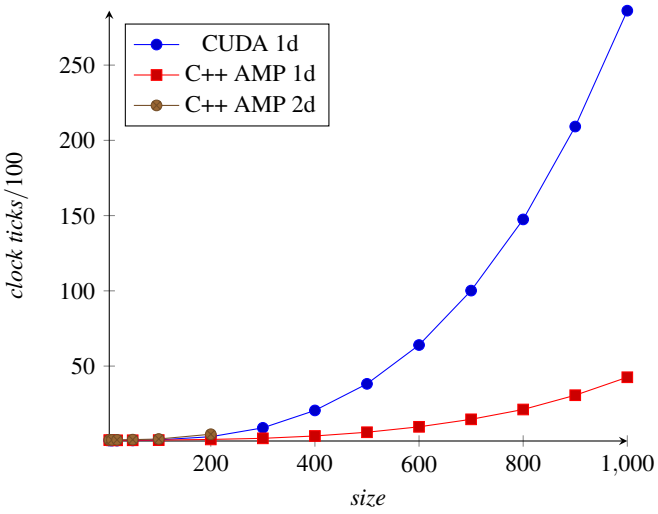

Fig. 8: samlet C++

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

På 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 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.

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.

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 [7]. *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*.

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 tager en god potion tid at hente denne information. Konstruktøren og de globale variabler kan ses i kode snippet 10, *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. 10: 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 med hvordan hvordan GPU'en skal udregne. Denne liste har *X* antal a fire tal som er en enkle

udregning (+,-,*,/), *X* er hvor mange udregning er skal gøres i alt, for at komme til det ønskede resultat. 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.

makeFuncHelper kan ses i fem dele. Første del kan ses på kode snippet 11 hvor den finder ud af om input variabler er en funktion eller en værdi.

```
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. 11: Første del af *makeFuncHelper* som kigger på variabler af input.

Anden del kan ses på kode snippet12, 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 il 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 from for udregning/operation der bliver gjort i input, dette kan ses i kode snippet 13

Fjerde del, der kan ses på kode snippet14 stater med at lave en liste *result* der bliver brugt til holde indkodning af input. Derefter kigger den på om inputs værdier har været funktion, hvis de er vil den fjerne den midlertidig variable for dem, siden de bliver brugt i denne udregning, kan den godt genbruge forrige brugte midlertidig variabler placeringer, for at spare på pladsen på GPU.

Den femte og sidste del 15 i *makeFuncHelper* ligger at 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               if (oneIsFunc)
2               {
3                   temp.AddRange(makeFuncHelper(input.es[0]
                        as FunCall,false));
4                   locationOne = temp[temp.Count - 1][3];
5               }
6               else if(oneIsNumber)
7               {
8                   locationOne = (int)Value.ToDoubleOrNan((
                        input.es[0] as NumberConst).value);
9               }
10              else
11              {
12                  // some kind of error...
13              }
14              if (twoIsFunc)
15              {
16                  temp.AddRange(makeFuncHelper(input.es[1]
                        as FunCall, false));
17                  locationTwo = temp[temp.Count - 1][3];
18
19              }
20              else if (twoIsNumber)
21              {
22                  locationTwo = (int)Value.ToDoubleOrNan((
                        input.es[1] as NumberConst).value);
23              }
24              else
25              {
26                  // some kind of error...
27              }
```

Fig. 12: Anden del af *makeFuncHelper* der laver arbejdet med inputs variabler.

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
- -1,3,-2,0

6.3 findnumberOfTempResult

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

```
int functionValue = 0; ;
string function = input.function.name.ToString
    ();
switch (function)
{
    case "+":
        functionValue = 1;
        break;
    case "-":
        functionValue = 2;
        break;
    case "*":
        functionValue = 3;
        break;
    case "/":
        functionValue = 4;
        break;
    default:
        // some kind of error...
        break;
}
```

Fig. 13: Tredje del af *makeFuncHelper* finder ud af hvad for en operation der sker i input.

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 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 på GPU tråde lokale hukommelse.

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.

6.5 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,

```
List<int> result = new List<int>();

if (oneIsFunc)
{
    tempResult.RemoveAt(tempResult.
        FindLastIndex(x => x == locationOne));
}
if(twoIsFunc)
{
    tempResult.RemoveAt(tempResult.
        FindLastIndex(x => x == locationTwo));
}

int outputPlace = 0;
if(!root)
{
    int x = -1;
    while (outputPlace == 0)
    {
        if(!tempResult.Contains(x))
        {
            outputPlace = x;
        }
        else
        {
            x--;
        }
    }
}
tempResult.Add(outputPlace);
```

Fig. 14: Fjerde del af *makeFuncHelper* fjerner brugt midlertidig variabler placeringer for genbrug og finde ud af hvor inputs resultat skal placeres.

```
result.Add(locationOne);
result.Add(functionValue);
result.Add(locationTwo);
result.Add(outputPlace);

temp.Add(result);

return temp;
```

Fig. 15: Femte del af *makeFuncHelper* sætter alt sammen og sender hvad den har fundet ud af tilbage.

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 liger meget hvordan *makeFuncHelper* er lavet.

6.6 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. de linjer der er sat ind kan ses her16.

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

Fig. 16: Ting der er lavet i *Function* konstruktøren.

Den sidste del af koden i denne klasse ligger i *GPUFunction*17, 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.

```
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. 17: *GPUFunction* der bliver brugt *Function* konstruktøren.

7 Test af GPU-calculate

Der er lavet 2 test inden for *Funcalc* der har en stor lighed med hinanden. 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. Denne test starter med *A1*B1* og for ver *x* bliver der liget et ekstra *+A1*B1* på udregningen.

I Test to 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 ver *x* bliver der liget et ekstra *+A1*B1*C1*D1* på udregningen.

7.1 Resultater

på tabel 18 kan resultaterne fra første test ses og på tabel 19 kan resultaterne fra første test ses

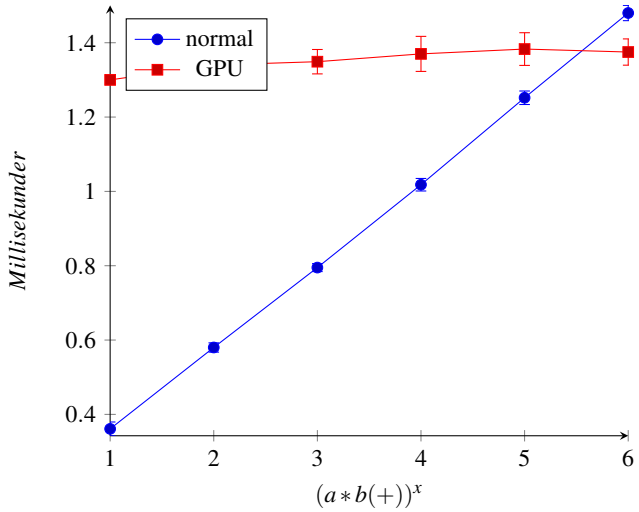

Fig. 18: testen af Funcalc hvor kolonerne er fyldt ud, 1000 tal i vær kolonne, hvor *A=2*,*B=3* og C er hvor funktioner ligger. *x* står for hvor mange udredninger der er af (a1*b1(+)) og for GPU (1*2(+)) .

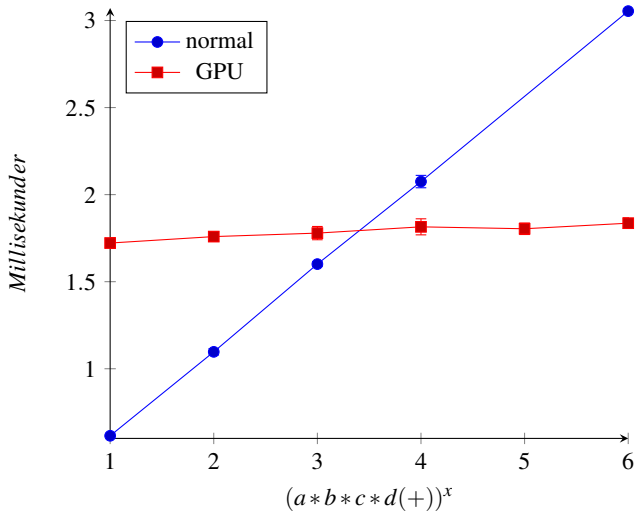

Fig. 19: testen af Funcalc hvor kolonerne er fyldt ud, 1000 tal i vær kolonne, hvor *A=B=2*,*C=D=3* og E er hvor funktioner ligger. *x* står for hvor mange udredninger der er af (a1*b1*c1*d1(+)) og for GPU (1*2*3*4(+)).

8 Diskussion

For at først at kigge på mulighederne for at kode til en *GPU* til C #, som er det sprog *Corecalc* er skrevet i, er der blevet lavet nogen forskellige test der giver nogen interessante resultater. Vis man kigger på *C++ AMP* og *CUDA* i tabellen 8 kan det ses 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 rode med hvor mange tråde og blokke der skal bruges.

En overraskende ting kan ses i CUDAfy 1d og 2d kan det ses, at det er hurtigere at, hvis håndterer data i 1d array i sted for 2d array, når matrixen begunder at blive have den størrelse være 10 eller over. Hvilket er det modsatte af hvad artiklen *Adaptive Input-aware Compilation for Graphics Engines*[9] fokus er.

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 slev 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*[9] 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, hvis det hele ikke kan gøres på en blok, denne metode er dog ikke perfekt. Siden der kunne laves noget optimering af hvordan data bliver gemt til GPU

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 1023. 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, men desværre ikke skulle være muligt med CUDAfy på dette tidspunkt.

Igennem resulterende af testen4.1 for GPU funktion og den simple måde at lave funktioner til *Funcalc*, er det første ting der viser sig at jo mere data der bliver arbejdet med jo hurtigere bliver det bedre at udregne på GPU, dette tilfælde kunne tilfælde kunne skylde måden data bliver gemt i *Funcalc*, hvor der kan være langt imellem den data der skal bruges, hvorimod på GPU vil data ligge ved siden af hinanden for den enkelte tråd, så derved bliver der ikke brugt nær så meget tid på dette. Noget andet der kan ses ud fra testen, er af GPU er stort set en vandret linje, hvorimod at den normal at stille den form for funktion op, bliver til en linje der cirka går med en 45% stigning. Derved kan man sige at hvis en funktion er stor nok eller der er mange data der skal arbejdes, kunne GPU funktion godt bruges

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 sige noget om det er blevet nummerere at kode til en GPU. Det kunne være at GPU funktion var nemmere at bruge en *funcalc* funktion ark for personer der ikke har erfaring med at programmer, siden man ikke skal lave et simple program, men et udtryk som bruger, der har brugt regnearks programmer burde have erfaring med.

Testende 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

På grund af tids præs eller kommet op med ideen forsendt, er der planer om at lave videre på dette projekt.

Først ting kunne være at lave ligt 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 skriver funktion giver lod til at markeret et celle, 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.

Man kunne også gøre at GPU kan tage en *funcalc* funktion, skrevet i et funktion ark, i sted for at kun kan tage en *CoreCalc* udtryk. Ved at gøre dette ville bruger venligheden muligvis blive bedre. Det burde være relativ simple at gøre da en *funcalc* funktion vist også skulle lave et abstrakt syntaks træ over hvordan funktions skal udregnes, derved kan man løbe dette abstrakt syntaks træ igennem på samme måde det bliver gjort for et *CoreCalc* udtryk. Det kunne også være interessant 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 funktion selv finde ud af om de tal i udtrykket pejler på noget inde i input array eller om det er noget udenfor der, hvilket vil gøre det til en global værdi.

Der kunne laves nogen bestemt lavet GPU funktioner over kendte algoritme problemer, en af dem kunne være det samme om der er blevet brug til at test de forskellige sprog matrix Multiplatikon, koden er jo næsten lavet til fulde, 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 bliver aldrig testet. Derfor kunne det være intersant at lave en bruger til på 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 ver intersant at teste, ville være om måden at et udtryk bliver indkodet, som så en GPU funktion kan læse i dette projekt på run time. Kan stille noget op 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.

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 (ASPLOSŠ15). ACM, New York*, pages 577–591, 2015.
2. 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.
3. Sparsh Mittal and Jeffrey S Vetter. A survey of cpu-gpu heterogeneous computing techniques. 2015.
4. Nichlas Korgaard Mller and Niels Brndum Pedersen. A comparative study of cpu and gpu programming in different languages and libraries, 01 2016. P9, Autumn Semester 2015.
5. John Nickolls and William J Dally. The gpu computing era. *IEEE micro*, (2):56–69, 2010.
6. 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.
7. IT University of Copenhagen. Corecalc and funcalc spreadsheet technology in c sharp. `http://www.itu.dk/people/sestoft/funcalc/`.
8. Niels Brndum Pedersen. Github repository for the projekt. `https://github.com/Dugin13/P10`.
9. 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.
10. Peter Sestoft (sestoft@itu.dk). *Spreadsheet Implementation Technology: Basics and Extensions*. The MIT Press Cambridge Massachusetts, London, Enland, 2014.
11. Peter Sestoft (sestoft@itu.dk). Microbenchmarks in java and c sharp, 9 2015.
12. 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.