Dokumentation: CUDA/MPI - Grayscaling und Embossing

Ilona Eisenbraun, Thomas Diewald
s0561762, s0554334
30. Juli 2020

Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Hochschule für Technik und Wirtschaft Berlin
Fachbereich 4: Informatik Kommunikation und Wirtschaft
Studiengang: Angewandte Informatik Master
Seminar: Programmierkonzepte und Algorithmen
Betreuer: Prof. Dr. Mykyta Kovalenko

Diese Arbeit ist im Rahmen des Moduls „Programmierkonzepte und Algorithmen“des Masterstudiengangs „Angewandte Informatik“an der Hochschule für Technik und Wirtschaft in Berlin im Sommersemester 2020 unter der Aufsicht von Prof. Dr. Mykyta Kovalenko entstanden.

Inhaltsverzeichnis

1 Einführung

Im Folgenden wollen wir anhand unserer Arbeit zeigen, wie die parallelisierte Bildverarbeitung in der „State of the Art“-Prozedur gehandhabt wird.

Dabei fungiert die Programmierschnittstelle MPI (Message Passing Interface) als Lastenverteiler von Berechnungen verschiedener Prozesse. Diese wiederum werden an CUDA (Compute Unified Device Architecture) - als API für parallelisierte Berechnungen auf der GPU (Graphical Processing Unit) - weitergeleitet.
Das Zusammenspiel beider APIs ermöglicht einen optimale Nutzung der Rechenkapazität - so zum Beispiel auch die eines Rechenclusters - für die Verarbeitung von Bildern. Dies ist dadurch ebenso auch für large-scale Instanzen effizient.

Durch Nutzung dieser beiden Interfaces wollen wir ein Programm schaffen, welches die Ressourcen eines Rechenclusters Optimal nutzen kann. Insbesondere lag unser Augenmerk darauf, dass sowohl die CPU als auch die GPU als Recheneinheiten hervorragend genutzt werden. Dies bedeutet für die GPU, dass diese möglichst kleine, gekapselte Aufgaben bekommt, sodass die genutzten CUDA-Cores diese parallel abarbeiten können. Die Funktion der CPU dagegen liegt bei der Berechnung von langen zusammenhängenden Aufgaben.

Auf diese Art und Weise werden wir beleuchten, wie ein beispielhaftes Bild möglichst effizient anhand seines RGB-Farbraums zuerst in Graustufen verarbeitet und anschließend seine Kanten mit dem Emboss-Algorithmus erkannt werden können.

2 Reproduktion der Arbeit

Für das lokale Nachstellen dieser Arbeit ist es notwendig, sowohl das CUDA Toolkit [1] als auch Microsoft MPI [2] installiert zu haben. Da auf Windows-Systemen gearbeitet wurde, haben wir hierbei als IDE Microsoft Visual Studio 2019 [3] verwendet. Um das Kompilieren unseres Projekts zu ermöglichen, werden die Linker hierbei intern statt über eine Makefile gesetzt. Dafür war es wichtig, bereits zu Beginn ein „CUDA Runtime“-Projekt zu erstellen (siehe Abb. 1), sodass der NVCC-Compiler genutzt wird.

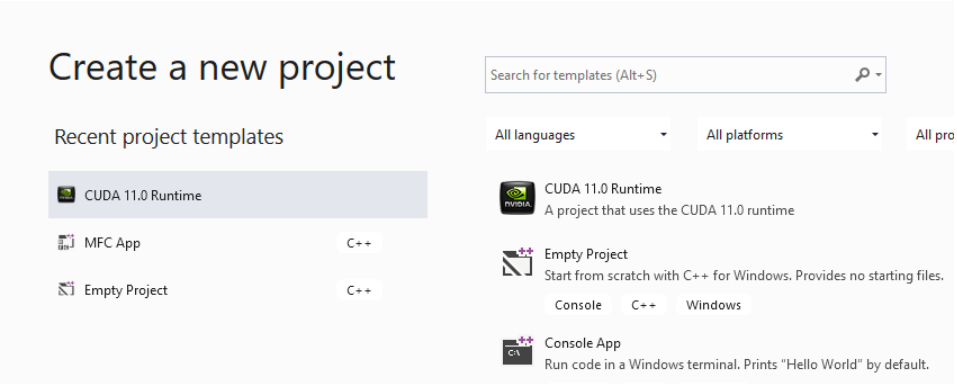

Abbildung 1: Erstellen des „CUDA Runtime“-Projekts

Ferner müssen sowohl Microsoft MPI als auch OpenCV als Libraries eingebunden werden, sodass sie aktiv im Projekt genutzt werden können. Für MPI wird `Include` und `Lib` angegeben, wie in Abildung 2 zu sehen ist. `Lib` wurde hierbei über Linker-Settings inkludiert. Ähnlich wurde auch OpenCV, welches für das Einlesen genutzt wird, integriert.

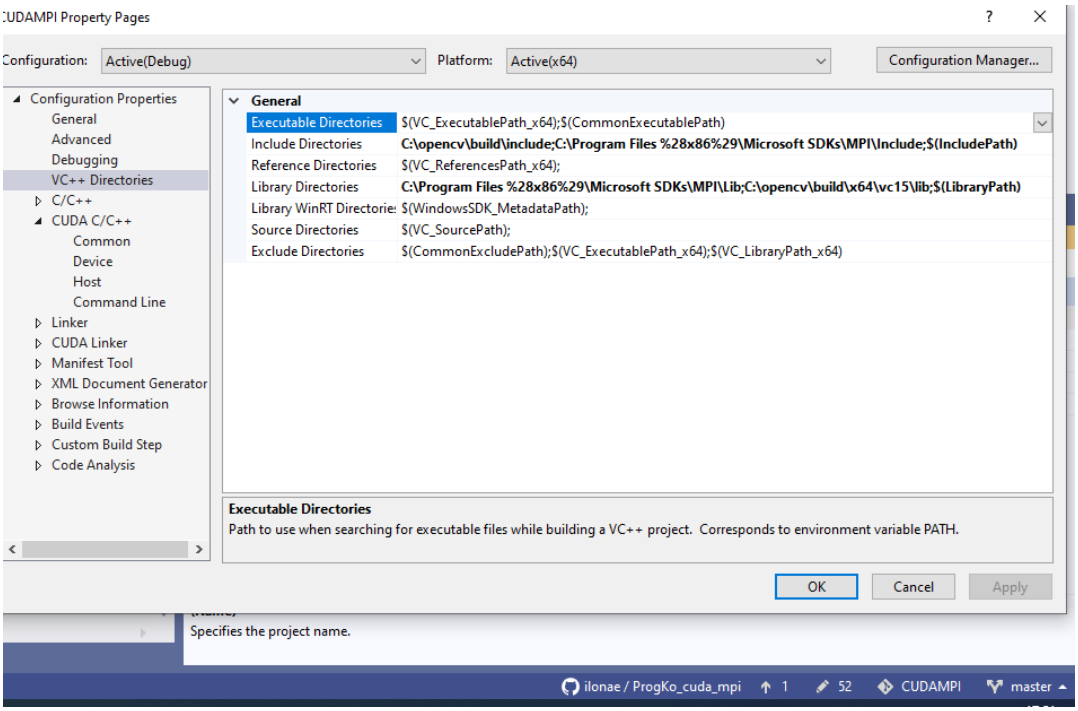

Abbildung 2: Einbinden von OpenCV und MPI

3 Lösungsansatz

3.1 Einlesen eines Bildes

Für den Ansatz unserer Implementation haben wir uns dazu entschieden, die Bilddatei in jeglichem Format - sei es bspw. eine PNG- oder JPEG-Datei - einlesen zu können. Hierfür haben wir die von OpenCV bereitgestellte Funktion `imread` verwendet. Diese liest die Datei vom Dateipfad ein, der von unserem Programm als Argument über die Kommandozeile empfangen wird (hier als `argv[1]`). Die Flags, die wir für `cv::imread` setzen sind `cv::IMREAD_COLOR||cv::IMREAD_UNCHANGED`, sodass letztendlich eingelesene Bilder entweder zum BGR-Farbraum konvertiert werden oder unverändert eingelesen werden.

3.2 Parallelisierung mit MPI

In unserer `main`-Funktion initialisieren wir die MPI-API, ebenso wie wir die Anzahl an möglichen Prozessen festlegen, wie in Listing 1 zu sehen ist.

```
1 int size, rank;
2 MPI_Init(&argc, &argv);
3 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
4 MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Listing 1: MPI: Initialisierung

In diesem Beispiel werden die Variablen `size` und `rank` initialisiert. Im Anschluss liest `MPI_Init` ein, welche Parameter für MPI übergeben werden sollen. Ein Beispiel für den Parameter wären hier die Anzahl der Prozesse. Im Anschluss wird eine Kommunikationsgruppe `MPI_COMM_WORLD` erstellt. Da es keinen Anlass gab die Kommunkationsgruppe zu begrenzen, werden sämtliche durch `MPI_Init()` erstellten Prozesse in der Kommunikationsgruppe `MPI_COMM_WORLD` zugewiesen und ihnen jeweils ein individueller Rang zugewiesen.

Dies wird dahingehend wichtig, da auch die Aufteilung unseres Bildes abhängig wird von der Anzahl der Prozesse, wie in Listing 2 zu sehen ist.

```
1 int newcol = std::ceil(((double)image.cols / (size)));
2 int newrow = std::ceil(((double)image.rows / (size)));
3 fullnewcol = newcol * size;
4 fullnewrow = newrow * size;
```

Listing 2: MPI: Slicing

Hier wird die gesamte Anzahl an Reihen und Spalten durch die Prozessanzahl geteilt und danach gerundet, um für die zukünftig entstehenden Bilder feste Größen für jeden einzelnen Prozess setzen zu können. Entsprechend wird das Bild immer separat „gesliced“, sodass wir diese später mit MPI jeweils versenden können (siehe Listing 3).

```
blankslicegrey = cv::Mat(fullnewrow, fullnewcol, CV_8UC4);
blanksliceemboss = cv::Mat(fullnewrow, fullnewcol, CV_8UC4);
sendslice = cv::Mat(fullnewrow, fullnewcol, CV_8UC4);
```

Listing 3: MPI: Slices zu Bildmatrizen

3.2.1 Parameterverteilung an Prozesse

MPI besitzt die Eigenschaft, den Speicher unter den Prozessen nicht zu teilen. Da jedoch benötigte Variablen an die entsprechenden Subprozesse verteilt werden müssen, wurden diese über einen Broadcast an alle beteiligten Subprozesse gesendet. Dies geschieht mittels `MPI_BCast()`, wie in Abb. 3 zu sehen.

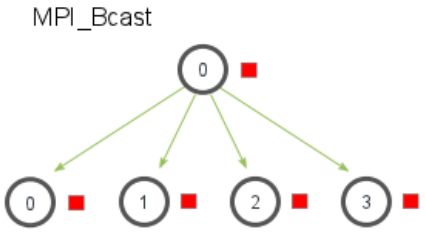

Abbildung 3: MPI: Darstellung von `MPI_BCast()`, entnommen von [4]

Hierdurch warten alle Prozesse auf die benötigte Variablen, um im Anschluss direkt mit der Ausführung des Codes zu beginnen.

```
//Initialisierung der Variablen
int rowstep=0;
...
//root Process gibt die Variable an jeglichen Subprozess weiter.
MPI_Bcast(&rowstep, 1, MPI_INT, root, MPI_COMM_WORLD);
```

Listing 4: MPI: Broadcasting

Wie in Listing 4 - als repräsentativer Ausschnitt - zu sehen ist, erhält jeder der Prozesse in der Kommunikationsgruppe `MPI_COMM_WORLD` den Wert der Variable `rowstep`. Dadurch können alle Prozesse dieser Gruppe mit dieser weiterarbeiten.

3.2.2 Streuen und Bündeln von Operationen

Das Versenden von Daten in gleich großen Paketen an alle vorhandenen Prozesse dagegen geschieht mit `MPI_Scatter()` (siehe Abbildung 4).

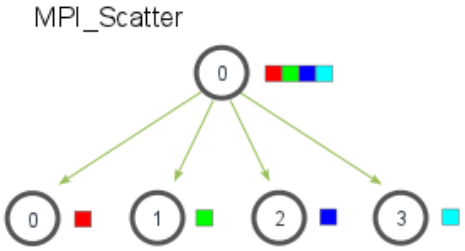

Abbildung 4: MPI: Darstellung von `MPI_Scatter()`, entnommen von [4]

Hierbei wird angegeben, welche Größe die gesendeten und empfangenen Daten haben, ebenso welchen Variablen diese beim Empfang zugewiesen bekommen. Zudem muss definiert werden, welcher Prozess die Daten an die anderen versendet. Im Beispiel in Listing 5 versendet der `root`-Prozess die Daten an alle Teilnehmer der Kommunikationsgruppe `MPI_COMM_WORLD`. Hier ist wichtig, dass die Gesamtgröße der Daten gleichmäßig auf die Anzahl der Prozesse aufgeteilt ist. Sendsslice ist das Gesamtbild im Root Prozess und mat.data stellt den Bildabschnitt des Subprozesses dar.

```
MPI_Scatter(sendslice.data,
            fullnewrow * colstep * 4,
            MPI_BYTE,
            mat.data,
            fullnewrow * colstep * 4,
            MPI_BYTE,
            root,
            MPI_COMM_WORLD);
```

Listing 5: MPI: Scattering

Alternativ lässt sich auch die Nutzung von `ScatterV()` herbeiziehen, bei welchem die Datenpakete und deren Größe durch `send_counts` und `displacements` definiert werden müssen. Der jeweilige Prozess wartet bei `ScatterV()` auf den Datenempfang und fängt erst dann unmittelbar mit der Weiterverarbeitung an.

Als Gegenstück zum Scattering fungiert dagegen `MPI_Gather()` und ermöglicht das finale Einsammeln der versendeten Datenpakete. Das Prinzip wird in Abb. 5 ersichtlich.

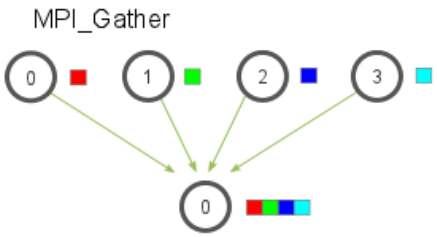

Abbildung 5: MPI: Darstellung von `MPI_Gather()`, entnommen von [4]

Hierbei muss der Buffer (`recv_data`) abseits des `root`-Prozesses keinen Wert besitzen. Sämtliche Daten werden schlussendlich wieder beim `root` vereint. Sowohl `Scatter` als auch `Gather` arbeiten nach dem Senden bzw. Empfangen den folgenden Code parallel ab, bis `MPI_Finalize()` im Subprozess aufgerufen wird und damit die parallele Berechnung beendet. Sofern `MPI_Scatter()` richtig verwendet wurde, führt es zu einer Parallelität und verhindert hierbei Deadlocks.

3.3 GPU-Nutzung mit CUDA

Wie bereits erwähnt ist CUDA eine Schnittstelle, speziell für Nvidia-GPUs. GPUs haben enorm viele Kerne, die parallel zueinander Berechnungen durchführen können. Im Gegensatz zur CPU jedoch liegt der Fokus auf möglichst einfachen, gekapselten Aufgaben, die von den Cores ausgeführt werden. Dadurch, dass diese zahlreich vorhanden sind, entsteht ein erheblicher Performance-Gewinn, da einfache Aufgaben zeitgleich verarbeitet werden können.

Zu Beginn ist es notwendig, die benötigten Variablen vom `Host` auf das `Device` zu übertragen, sodass die Kerne Zugriff auf die zu verarbeitenden Daten erhalten. Hierfür muss der Speicher in Form von Bytes durch `cudaMalloc` vorinitialisiert werden, wie in Listing 6 zu sehen ist.

```
1 const int colorBytes = input.step * input.rows;
2 SAFE_CALL( cudaMalloc <unsigned char> (&d_input, colorBytes), "CUDA
    ↪ Malloc Failed" );
3 cudaMemcpy( d_input, input.ptr(), colorBytes, cudaMemcpyHostToDevice );
```

Listing 6: CUDA: Speicherreservierung

Für das Error-Handling wird hier `SAFE_CALL()` verwendet, um eventuell entstehende Fehler der Speicherplatzreservierung auszugeben.

Weitergehend müssen die Blöcke initialisiert werden, die im Grid zu 32 * 32 Blöcken, also 1024 Threads angeordnet sind. Im Anschluss wird das zu berechnende Bild auf das Grid (List. 7).

```
1 const dim3 block(32, 32);
2 const dim3 grid( ((input.cols + block.x - 1) / block.x),
3                  ((input.rows + block.y - 1) / block.y) );
```

Listing 7: CUDA: Dimensionseinteilung

Durch den Kernel werden nun die Aufgaben an die einzelnen Threads mittels Angabe von `grid` und `block` übergeben (siehe Listing 8). Jeder Thread stellt hierbei einen CUDA-Core dar, dem Berechnungen vom Grid zugewiesen werden. Nach dem Durchlaufen der Berechnungen werden die Ergebnisse schlussendlich wieder vom `device` auf den `host` überführt, indem der hierfür reservierte Speicherbereich `d_output` beschrieben wird.

```
grayscale_kernel << <grid, block >> > ( d_input, d_output, input.cols,
    ↪ input.rows, input.step, output.step );

const dim3 grid( (( input.cols + block.x - 1) / block.x),
                 (( input.rows + block.y - 1) / block.y) );
cudaMemcpy( output.ptr(), d_output, grayBytes, cudaMemcpyDeviceToHost );
```

Listing 8: CUDA: Speicherfreigabe

Final wird der reservierte Speicherbereich der Grafikkarte wieder durch `cudaFree()` freigegeben.

3.3.1 Image-Grayscaling

Grayscaling beschreibt den Prozess der Umwandlung eines Bildes in ein Grauwertstufen. In dieser Arbeit wird die Umwandlung durch die Verwendung von CUDA realisiert. Hierbei wurden folgende Gewichtungen der RGB-Kanäle adaptiert:

Rot = 21%, Grün = 72% und Blau = 7%.

In unserer initialen Grayscale-Implementation funktionierte aufgrund der Verwendung von `ScatterV` die eindimensionale Berechnung des Grauwertbildes nicht, was schlussendlich durch die Verwendung von `MPI_Scatter()` auch für eindimensionale Umwandlung realisiert werden konnte. Hier wurden zuerst die X- und Y-Koordinaten des Bildes bestimmt und daraufhin die neu errechneten Werte in das eindimensionale Output-Array geschrieben. Daher war der ursprüngliche Ansatz, den jeweiligen Pixel `color_tid` entsprechend auf alle Kanäle zu mappen.

```
__global__ void grayscale_kernel_single(unsigned char* input, unsigned
    ↪ char* output, int width, int height, int colorWidthStep, int
    ↪ grayWidthStep) {
    {
        const int x = blockIdx.x * blockDim.x + threadIdx.x;
        const int y = blockIdx.y * blockDim.y + threadIdx.y;
        if ((x < width) && (y < height))
        {
            //Loc base Image
            const int color_tid = y * colorWidthStep + (4 * x);

            //Loc in Grayscale
            const int gray_tid = y * grayWidthStep +  x;

            const unsigned char blue = input[color_tid];
            const unsigned char green = input[color_tid + 1];
            const unsigned char red = input[color_tid + 2];
            const unsigned char alpha = input[color_tid + 3];
```

```
17              const float gray = red * 0.21f + green * 0.72 + blue *
    ↪ 0.07f;
18
19              output[gray_tid] = static_cast<unsigned char>(gray);
20          }
21      }
22
23  }
```

Listing 9: CUDA: Grayscaling

3.3.2 Image-Embossing

Die Kanten eines Bildes werden darauf folgend mit dem Emboss-Algorithmus verarbeitet. In unserer Implementation werden die RGB-Werte des aktuellen Pixels mit dem linken, oberen Pixel verglichen. Die Differenz zwischen den beiden RGB-Werten wird als signed Int ermittelt, um diesen mit dem Mittelwert 128 zu addieren (wie im Pseudocode 1 zu sehen ist):

Ergebnis: Grau = 128 + Differenz
wenn *Grau* > *255* **dann**
  Grau = 255 ;
sonst wenn *Grau* < *0* **dann**
  Grau = 0 ;

Algorithmus 1: Pseudocode zum Embossing, entnommen aus [5]

Das Clipping der Werte stützt sich hierbei auf dem im Rahmen des Moduls vorgestellten Beispiel, entnommen aus [5]. Sollte der Grauwert des einzelnen Pixels größer als 255 sein, wird dieser auf 255 festgelegt. Sollte der Wert dagegen negativ sein, wird er auf 0 geclipped. Der Alphakanal bleibt wie im Ursprungsbild erhalten.

```
1  __global__ void emboss_kernel(unsigned char* input, unsigned char*
       ↪ output, int width, int height, int colorWidthStep, int
       ↪ grayWidthStep) {
2
3      bool once=true;
4      const int x = blockIdx.x * blockDim.x + threadIdx.x;
5      const int y = blockIdx.y * blockDim.y + threadIdx.y;
6      const int xminus = blockIdx.x * blockDim.x + (threadIdx.x);
7      const int yminus = blockIdx.y * blockDim.y + (threadIdx.y);
8
9      if ((x < width) && (y < height))
10     {
11       const int color_tid = y * colorWidthStep + (4 * x);
```

```
        int emboss_tid=0;
        //Loc base Image
        if ((xminus > 1) && (yminus > 1)) {
            emboss_tid = (yminus)*colorWidthStep + 4 * (xminus - 1);

        }
        else {
            emboss_tid  = color_tid;
        }


        const float RGB[3]{ input[color_tid] * 1.0f, input[color_tid +
    ↪ 1] * 1.0f, input[color_tid + 2] * 1.0f };

        const float RGBdiff[3]{ input[emboss_tid] * 1.0f,
    ↪ input[emboss_tid + 1] * 1.0f, input[emboss_tid + 2] * 1.0f };


        const float diffs[3]{ RGB[0] - RGBdiff[0], RGB[1] - RGBdiff[1],
    ↪ RGB[2] - RGBdiff[2] };


        float diff = diffs[0];
        if (abs(diffs[1])>abs(diff)) { diff = diffs[1]; }
        if (abs(diffs[2]) > abs(diff)) { diff = diffs[2]; }

        float gray = 128 + diff;
        if (gray > 255) { gray = 255; }
        if (gray < 0) { gray = 0; }
        output[color_tid] = static_cast<unsigned char>(gray);
        output[color_tid + 1] = static_cast<unsigned char>(gray);
        output[color_tid + 2] = static_cast<unsigned char>(gray);
        const unsigned char alpha = input[color_tid + 3];
        output[color_tid + 3] = static_cast<unsigned char>(alpha);
    }
}
```

Listing 10: CUDA: Embossing

4 Ergebnisse

Für den Abgleich der Resultate haben wir die im Rahmen des Kurses bereitgestellten Bilder aus dem Dropbox-Folder entnommen [6]. Die Ergebnisse sind anhand des Würfel-

Beispiels in den Abb. 6, 7 und 8 zu sehen.

Abbildung 6: Original-Bild, entnommen von [6]

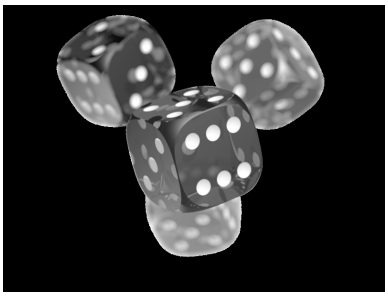

Abbildung 7: Grayscaling-Resultat

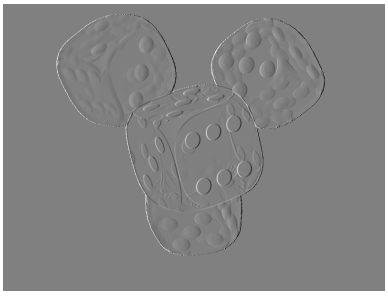

Abbildung 8: Embossing-Resultat

5 Performance-Betrachtung

Um zu schauen, wie sich die Performance unserer Implementation zur Laufzeit verhält, haben wir unterschiedliche Ansätze ausprobiert. Initial wurde seitens Ilona auf dem Remote „deepgreen“der HTW Berlin gearbeitet, da dies aber zu vielen Schwierigkeiten führte, konnte die CUDA-fähige Maschine eines Freundes genutzt werden. Entsprechend haben wir im ersten Schritt die unterschiedlichen Kapazitäten der Maschinen gegen einander gestellt. Die Systemspezifikationen sind dabei in der Tabelle 1 festgehalten sind.

Prozessor	GPU	Arbeitsspeicher	Name
Core I7 9700K	Nvidia RTX 2080Ti	32GB DDR 4 RAM 3200MHZ	Thomas
8-Core FX-9590	Nvidia GTX-1060	32GB DDR 3 RAM 2209.7MHZ	Ilona

Tabelle 1: Systemspezifikationen der Gruppenmitglieder, im Vergleich

5.1 MPI-Implementation

Für den Vergleich der MPI-Operationen haben wir zwischen den einzelnen Schritten Auswertungen mit `MPI_Wtime()` gemessen. Jegliche Zwischenschritte wurden festgehalten, indem diese in eine CSV-Datei gespeichert wurden. Dabei wurden folgende Werte gemessen, die sich ebenso im CSV-Header wiederfinden: „Time before Scatter“, „Begin Grayscale“, „End Grayscale“, „Begin Emboss“, „End Emboss“, „Finishing Time“, „Rank“. Für die Ermittlung der Werte wurden für unterschiedliche Prozessanzahlen jeweils 10 Schritte gemessen, sodass hierüber ein Mean ermittelt werden konnte. Da die Zeitunterschiede zwischen dem Grayscaling und dem Embossing bspw. aber insignifikant sind, wurden lediglich die Differenzen zwischen Start- und Stoppzeiten betrachtet. Für einen Abgleich wurden Tests mit unterschiedlichen Parametern für die Anzahl an Prozessen durchgeführt. Auch die Unterschiede in der Bildgröße sind hervorzuheben, so ist dies in der Abb. 10 zu sehen:

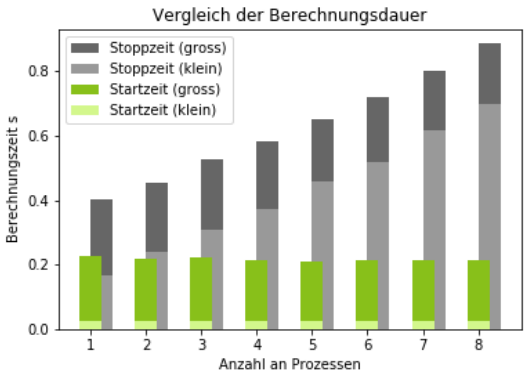

Abbildung 9: Benchmarks Ilona

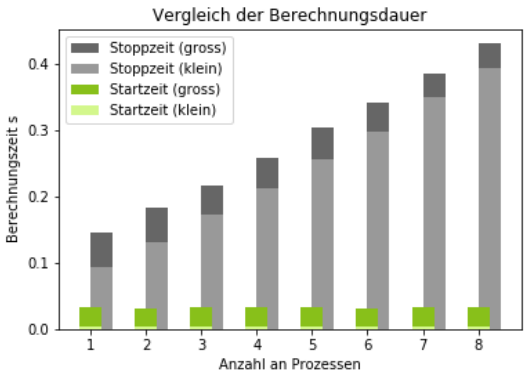

Abbildung 10: Benchmarks Thomas

Für die Berechung auf der Maschine von Thomas benötigt das kleine Würfelbild - zum kompletten Durchlauf mit einem Prozess - etwa 0.09 Sekunden. Das große Bild dagegen benötigt 0.14 Sekunden. Mit steigender Prozessanzahl, die dem Programm über den Parameter übergeben wurden, stieg hier ebenso auch die komplette Laufzeit: so konnte, wie in Abb. 10 zu sehen ist, für ein kleines Bild bei 8 Prozessen 0.39 Sekunden gemessen werden.

Auf der Maschine, die von Ilona genutzt wurde, dauert der Durchlauf eines kleinen Bildes dagegen 0.26 Sekunden, bei einem Prozess. Ansteigend, für 8 Prozesse, wurde eine Maximallaufzeit von 0.78 Sekunden gemessen. Bei der Messung von dem großen Bild betrug diese 0.88 Sekunden, ebenso bei 8 Prozessen, wohingegen bei einem Prozess 0.4 Sekunden gemessen wurden.

5.2 CUDA-Implementation

Für die Auswertung wurde auch getestet, wie lange die einzelnen Schritte des Grayscalings zur Ausführung benötigt haben. Wie in Abbildung 11 zu sehen ist, benötigte alleine die Speicherzuweisung schon 0.13 Sekunden auf der von Ilona genutzten Maschine, wohingegen bei Thomas ein Mittelwert von 0.04 Sekunden gemessen wurde. Auch hier wurden mehrere Messungen durchgeführt, um eine generalisierte Aussage anhand von Mittelwerten treffen zu können.

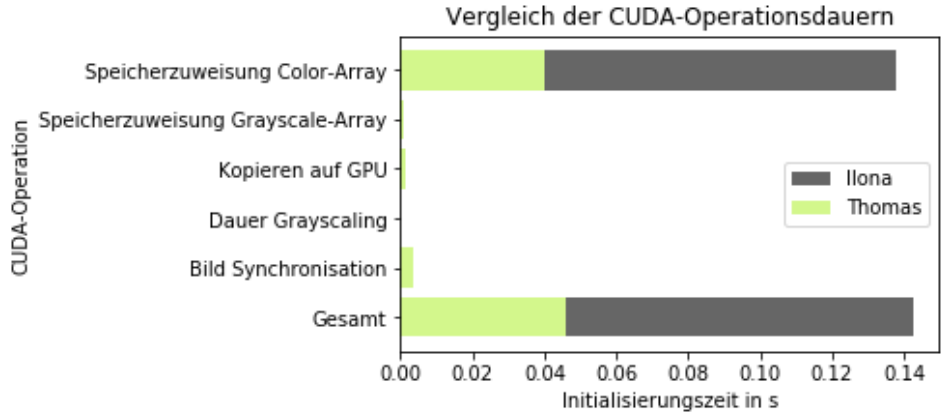

Abbildung 11: Vergleich der CUDA-Operationsdauern

Gemessen wurden ebenfalls die Dauer der Speicherallokation des Grayscale-Arrays, das Kopieren der Daten vom Host auf die GPU, die Kernelzeit für das Grayscaling und die Rücktransformation bzw. -synchronisation aller Threads zum resultierenden Bild.

Die Dauer dieser Operationen war jedoch sehr überschaubar, sodass z.T. auch „NaN"-Werte gemessen wurden.

5.3 CUDA vs. OpenCV

Da die OpenCV-Bibliothek inhärent ein Grayscaling mitbringt, war es interessant diese mit unserer CUDA-Implementation abzugleichen. Hierfür wurde in der Preprocessor-Direktiven `CUDAVSOPENCV` definiert, sodass auch hier durch `MPI_Wtime()` die Zeitdifferenzen gemessen werden konnten.

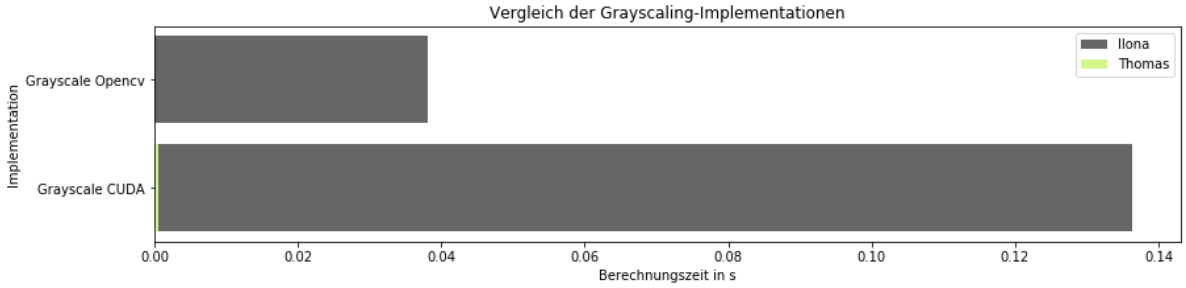

Abbildung 12: Vergleich der Grayscaling-Implementationen

Wie in Abb. 12 zu sehen ist, ist die CUDA-Implementation erheblich langsamer ist als die Grayscaling-Funktion von OpenCV: So braucht OpenCV durchschnittlich jeweils 0.4 und 47 Millisekunden, wohingegen es bei CUDA 95 und 133 Millisekunden waren (jeweils für Ilona und Thomas gemessen).

6 Fazit

Rückblickend konnte man feststellen, dass das Programm wirklich unterschiedliche Laufzeiten auf den verschiedenen Maschinen aufweist, was natürlich an der jeweiligen Rechenkapazität liegt. Was jedoch viel interessanter zu betrachten ist, ist die Entwicklung der Laufzeiten:

MPI-Implementation In der Implementation mit MPI konnte gezeigt werden, dass die Speicherallokation jedesmal ähnlich lange gedauert hat (siehe 5.1). Die Berechnungszeit der Bilder nimmt jeweils linear mit zunehmender Anzahl an Prozessen zu. Dabei ist der Unterschied der Berechnungsdauer zwischen kleinen und großen Bildern - unabhängig von der Anzahl an Prozessen - ähnlich groß. Obwohl der Ansatz vorlag, mittels beider Rechner ein Rechencluster zu schaffen, um so die volle Kapazität von MPI ausschöpfen zu können, ist dies leider nicht im Rahmen dieser Arbeit gelungen. Interessant wäre eine nachfolgende Betrachtung, um zu untersuchen, ob hierdurch eine Steigerung der Effizienz ermöglicht wird.

CUDA-Implementation In der Performance-Betrachtung von CUDA konnte festgestellt werden, dass die Speicherzuweisung den Großteil der Berechnungszeit benötigt (5.2). Restliche Operationsdauern sind nahezu zu vernachlässigen. Wie auch bereits in einem Github-Zitat [7] bereits vermerkt war:

> Your problem is that CUDA needs to initialize! It will always initialize for the first image and generally takes between 1-10 seconds, depending on the alignment of Jupiter and Mars. Now try this. Do the computation twice and then time them both. You will probably see in this case that the speeds are within the same order of magnutide, not 20.000x, that's ridiculous. Can you do something about this initialization? Nope, not that I know of. It's a snag.

Dies bedeutet, dass man sich bei der Verwendung von CUDA zur Berechnung im Rückschluss folgende Gedanken machen sollte:

- Steigt die Rechenzeit durch die CUDA-Initialisierung, im Vergleich zu einer exklusiven Nutzung der CPU?
- Ist die parallelisierte Berechnung auf der Grafikkarte zwangsläufig der performanteste Weg?
- Lohnt sich der Tradeoff der Rechenzeit zur spezifischen Operation, die durchgeführt werden soll?

Ganz klar wäre dies situationsabhängig. Wobei man hier schon absehen kann, dass es sich beispielsweise für einzelne kleine Bilder nicht lohnen würde, sie über CUDA parallelisiert zu verarbeiten. Die Vermutung liegt nahe, dass CUDA besonders im High Performance Computing [8] - besonders für große Datensätze oder größere Bilder und Videodateien - wirklich ergiebig wäre.

Literatur

[1] “Cuda toolkit 11.0 download,” Jul 2020. [Online]. Available: https://developer.nvidia.com/cuda-downloads

[2] “Microsoft mpi v10.1.2.” [Online]. Available: https://www.microsoft.com/en-us/download/details.aspx?id=100593

[3] “Visual studio 2019 für windows und mac herunterladen,” Jun 2020. [Online]. Available: https://visualstudio.microsoft.com/de/downloads/

[4] G. Watson, “Collective operations,” 2017. [Online]. Available: https://nyu-cds.github.io/python-mpi/05-collectives/

[5] “Lab 6: Cuda image processing.” [Online]. Available: http://faculty.ycp.edu/~dhovemey/spring2011/cs365/labs/lab6.html

[6] D. M. Kovalenko. (2020, Jul.) Dropbox image folder. [Online]. Available: https://www.dropbox.com/s/5uhqu8m8a7entcd/images.zip?dl=0”

[7] TimZaman. (2015, Jun) Why opencv gpu code is slower than cpu? [Online]. Available: https://stackoverflow.com/a/30693666

[8] L. Shi, H. Chen, and J. Sun, “vcuda: Gpu-accelerated high-performance computing in virtual machines,” vol. 61, 05 2009, pp. 1–11.

Abbildungsverzeichnis

Listings

Liste der Algorithmen