**Hochschule für Technik und Wirtschaft Berlin**
**Fachbereich 4: Informatik Kommunikation und Wirtschaft**
**Studiengang: Angewandte Informatik (M)**
**Seminar: Programmierkonzepte und Algorithmen**
**Seminarleiter: Prof. Dr. Volodymyr Brovkov**

# Bildverarbeitung mit Cuda in C/C++ RGB umrechnen zu luminosity Grau

Version 0.1

**Vorgelegt am 02.01.2018 durch:**
Florian Schwab, s0562789 , s0562789@htw-berlin.de
Elsa Buchholz, s0544180, s0544180@htw-berlin.de

# Table of Contents

# 1 Einleitung

Mit Hilfe von Cuda soll ein beliebiges Bild in verschiedene Farbkonstellationen konvertiert werden. Dabei können vier Möglichkeiten gewählt werden; Blur, Emboss, Grün-Blau-Wechsel und luminosity Grau.

Das Bild wird in dieser Arbeit im png-Format verarbeitet. Mit der Programmiertechnik Cuda und der Programmiersprache C/C++ wird die Bildverarbeitung auf der GPU parallelisiert, um eine hohe Rechenleistung zu realisieren.

Um das Bild in seinen Farbwerten verändern zu können, müssen die Pixel des Bildes, die jeweils einen RGBA-Wert speichern, in einer Matrix gespeichert werden. Danach wird mit Hilfe von Cuda ein Algorithmus verwendet, der die Pixel-Werte neu berechnet und somit eine neue RGBA-Matrix hergestellt wird. Aus dieser Matrix wird anschließend eine neue png-Datei erstellt.

Ein Vergleich von der Berechnung der Matrix auf der CPU anstatt auf der GPU soll zeigen, dass das sequenzielle Durchlaufen der Pixel über eine Schleife langsamer ist, als das parallele Berechnen der Pixel auf der GPU. Zusätzlich wird die Programmierbibliothek OpenCV verwendet und mit Cuda verglichen. OpenCV und Cuda vollziehen beide die Berechnungen auf der GPU.

# 2 Konvertierung einer png-Datei

Eine png-Datei ist eine Rastergrafik, die aus Pixeln besteht. Die Gesamtheit der Pixel ergeben zusammen ein Bild, wobei jedes Pixel eine Farbe repräsentiert. Die Farbe ergibt sich aus den drei Farbkanälen, Rot, Grün und Blau und optional aus einem vierten Kanal Alpha. Die Gesamtzahl der Pixel ergibt sich aus dem Produkt der Höhe und Breite des Bildes. Auf jedes Pixel des Ausgangsbilds wird eine Matrix angewandt, die die Werte der Farbkanäle enthält. Darauf werden für die Konvertierung zu Emboss, Blur, luminosity Grau oder Austausch grüner mit blauen Komponenten der entsprechende Algorithmus angewandt. Da auf jedes Pixel der gleiche Algorithmus angewandt wird, ist er für eine Parallelisierung geeignet und dementsprechend auch für Cuda.

Im Cuda Programmiermodel werden die CPU und die GPU für Berechnungen verwendet. Dabei wird in der Cuda Terminologie die CPU und deren Speicher als Host bezeichnet und die GPU und deren Speicher als Device. Der auszuführende Code wird auf dem Host ausgeführt. Vom Host aus wird der Speicher auf der Seite des Hosts und Devices gemanaged. Die auszuführenden Algorithmen bzw. Funktionen werden vom Kernel auf dem Device bereit gestellt. Es können ein oder mehrere Kernel aufgerufen werden, die auf dem Device mehrere Threads parallel ausführen können. Jeder Pixel repräsentiert ein Thread.

Ein Cuda Programm durchläuft dementsprechend folgende Schritte:

1. Speicher auf dem Host und Device zuweisen
2. Daten auf dem Host initialisieren
3. Daten vom Host auf das Device transportieren
4. Ausführen eines oder mehrerer Kernel
5. Ergebnisse vom Device auf den Host transportieren

### 2.1 Erstellen einer RGBA Matrix aus einer png-Datei

Die Bilddatei wird mit Hilfe der pnglite Bibliothek geladen und gespeichert.

```
#include "pnglite.h"
extern const char *__progname;
```

Das Programm wird aufgerufen, indem die zu lesende Datei, die Konvertierungsart und die Ergebnisdatei aufgerufen werden. Darauf hin wird das Bild geladen und der Speicher auf der CPU zugewiesen.

```
png_init(0, 0);

Image    *img          = read_image(argv[2]);
uint32_t *img_data     = img->pixels;
uint32_t *out_img_data = (uint32_t *) alloc_image_buffer(
    img);
```

Mit den Funktionen read_image() und der Funktion alloc_image_buffer() werden die Pixelanzahl des Bildes ermittelt und dafür entsprechend der Speicherplatz auf der CPU des Host zugewiesen.

Auf dem Device wird der Speicher mit der Funktion cudaMalloc() zugewiesen.

```
CUDA_CHECK(cudaMalloc((void **) &dev_imgdata, buffer_size
    ));
CUDA_CHECK(cudaMalloc((void **) &dev_imgdata_out,
    buffer_size));
```

todo: initialisieren der Daten code

```
        Inhalt...
```

Auf dem Host werden weiterhin die Daten vom Speicher des Host in den den Speicher des Devices kopiert.

```
CUDA_CHECK(cudaMemcpy(dev_imgdata, img_data, buffer_size,
     cudaMemcpyHostToDevice));
dim3 grid(img->width, img->height);
```

Ab jetzt werden die Kernelfunktionen auf dem Device ausgeführt. Dafür wird der Kernel definiert. Cuda besitzt Erweiterungen, den sogenannten Qualifizierer. Der Qualifizierer __global __weist darauf hin, dass es sich um einen Kernel handelt. Er wird einer Funktion voran gestellt und zeigt an, dass die Funktion von der CPU bzw. den Host aufgerufen wird und auf der GPU bzw. dem Device ausgeführt wird. Zusätzlich muss die Größe des Grids und die Größe des Blocks definiert werden auf der der Kernel ausgeführt wird [**?**].
todo grid block definition

### 2.2 Parallele Konvertierung der Matrix zu luminosity Grau

```
__global__ void kernel_gray(uint32_t *in, uint32_t *out,
    int w, int h){
int idx = blockIdx.y * w + blockIdx.x;

// Check if thread index is no longer within input array
if (ARRAY_LENGTH(in) >= idx) { return; }

uint8_t gray = (0.21 * RED(in[idx])) + (0.72 * GREEN(in[
    idx])) + (0.07 * BLUE(in[idx]));

out[idx] = RGBA(gray, gray, gray, ALPHA(in[idx]));
}...
```

### 2.3 Parallele Konvertierung der Matrix zu Emboss

```
__global__ void kernel_emboss(uint32_t *in, uint32_t *out
    , int w, int h) {
if (blockIdx.y < 1 || blockIdx.x < 1) { return; }

int idx     = blockIdx.y * w + blockIdx.x;
int idx_ref = (blockIdx.y - 1) * w + (blockIdx.x - 1);

// Check if thread index is no longer within input array
if (ARRAY_LENGTH(in) >= idx) { return; }
if (ARRAY_LENGTH(in) >= idx_ref) { return; }

int diffs[] = {
(RED(in[idx_ref]) - RED(in[idx])),
(GREEN(in[idx_ref]) - GREEN(in[idx])),
(BLUE(in[idx_ref]) - BLUE(in[idx]))
};

int diff = diffs[0];
```

```
if ((diffs[1] < 0 ? diffs[1] * -1 : diffs[1]) > diff) {
    diff = diffs[1]; }
if ((diffs[2] < 0 ? diffs[2] * -1 : diffs[2]) > diff) {
    diff = diffs[2]; }

int gray = 128 + diff;
if (gray > 255) { gray = 255; }
if (gray < 0) { gray = 0; }

out[idx] = RGBA(gray, gray, gray, ALPHA(in[idx]));
}
```

### 2.4 Parallele Konvertierung der Matrix zu Blur

```
__global__ void kernel_blur(uint32_t *in, uint32_t *out,
    int w, int h, int area) {
int idx = blockIdx.y * w + blockIdx.x;

// Check if thread index is no longer within input array
if (ARRAY_LENGTH(in) >= idx) { return; }

uint32_t min_x      = blockIdx.x < area ? 0 : blockIdx.x
    - area;
uint32_t min_y      = blockIdx.y < area ? 0 : blockIdx.y
    - area;
uint32_t max_x      = (blockIdx.x + area) >= w ? w :
    blockIdx.x + area;
uint32_t max_y      = (blockIdx.y + area) >= h ? h :
    blockIdx.y + area;
uint32_t num_pixels = 0;
uint32_t red_sum    = 0;
uint32_t green_sum  = 0;
uint32_t blue_sum   = 0;
uint32_t alpha_sum  = 0;
int      i          = 0;

for(int x = min_x; x < max_x; x += 1) {
for(int y = min_y; y < max_y; y += 1) {
i = y * w + x;

num_pixels += 1;
red_sum    += RED(in[i]);
green_sum  += GREEN(in[i]);
blue_sum   += BLUE(in[i]);
```

```
alpha_sum   += ALPHA(in[i]);
}
}

out[idx] = RGBA((red_sum / num_pixels), (green_sum /
    num_pixels), (blue_sum / num_pixels), (alpha_sum /
    num_pixels));
}
```

### 2.5 Parallele Konvertierung der Matrix mit Austausch der blauen und grünen Komponenten

```
__global__ void kernel_swap_green_blue(uint32_t *in,
    uint32_t *out, int w, int h){
int idx = blockIdx.y * w + blockIdx.x;

// Check if thread index is no longer within input array
if (ARRAY_LENGTH(in) >= idx) { return; }

out[idx] = RGBA(RED(in[idx]), BLUE(in[idx]), GREEN(in[idx
    ]), ALPHA(in[idx]));
}
```

### 2.6 Erstellen einer png-Datei aus der neuen RGBA-Matrix

## 3 Performancevergleich der parallelen Umwandlung mit OpenCV auf der GPU und ohne Parallelisierung auf der CPU

## 4 Fazit