Aufgabe 4

RGB ZU GRAYSCALE, HISTOGRAMM, HELLIGKEIT

ANDREY BORISOV
MATRIKEL NR.: 563858
ANDREY.BORISOV@STUDENT.HTW-BERLIN.DE

HTW Berlin
Programmierkonzepte und Algorithmen
Prof. Dr. Nikita Kovalenko

Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

Contents

1. Aufgabenstellung

Das Ziel der gegebenen Aufgabe besteht darin, eine Anwendung zu entwickeln, die Bilder als Input bekommt, dann bestimmte Operationen mit diesen Bildern durchführt und anschließend die Ergebnisse anzeigt. Folgende Funktionalitäten müssen dabei umgesetzt werden:

- **RGB zu Grayscale:** Das Input-Bild muss von der Anwendung so transformiert werden, dass man ein Schwarzweiß-Bild als Output bekommt.
- **Helligkeit:** Die Helligkeit des Input-Bildes muss um einen bestimmten Wert erhöht werden.
- **Histogramm:** Die Anwendung sollte ein Histogramm des Input-Bildes erzeugen können und dieses anzeigen.

Eine weitere Anforderung besteht darin, die Anwendung so zu implementieren, dass alle Berechnungen parallel in mehreren Threads ausgeführt werden. Die implementierte Anwendung sollte in der Dokumentation beschrieben werden, wobei man insbesondere auf den Zusammenhang zwischen der Parallelisierung und der Ausführungsgeschwindigkeit eingehen muss. Außerdem muss die Ausführungsgeschwindigkeit von ähnlichen Anwendungen oder Libraries mit der Ausführungszeit der erstellten Anwendung für dieselben Operationen vergleichen.

2. Beschreibung der Lösung

Für die Entwicklung der vorliegenden Anwendung wurde die Programmiersprache C++ benutzt. Um bestimmte Operationen an Bildern durchführen zu können, müssen diese Bilder vom Programm zuerst geladen werden, was mit der OpenCV-Bibliothek gemacht wird. Die Ausführung der implementierten Funktionalitäten wurde dabei mithilfe von OpenMP parallelisiert. Die umgewandelten Bilder und das Histogramm sollten dem Benutzer außerdem angezeigt werden, wofür ebenfalls die OpenCV Bibliothek benutzt wurde. Da das primäre Ziel der Parallelisierung die Verbesserung der Ausführungsgeschwindigkeit ist, wurde außerdem entschieden bestimmte Teile des Programms in Assembler zu implementieren, wodurch die Ausführung noch effizienter werden könnte[1].

Nachfolgend werden die implementierten Funktionalitäten detailliert beschrieben.

2.1. **Bildtransformation-Funktionalitäten.**

Beide Bildtransformationen (Änderung der Helligkeit und RGB zu Grayscale) können sehr gut parallelisiert werden und zwar weil die einzelnen Threads nicht auf das Ergebnis anderer Threads warten müssen, da die einzelnen Pixel unabhängig von einander verarbeitet werden können. Das heißt insbesondere, dass die einzelnen Threads weder synchronisiert werden müssen, noch besitzen sie gemeinsame Variablen. Bei der Implementierung von den gegebenen Bildtransformationen können daher die einfachen for-Schleifen eingesetzt werden, deren Ausführung durch OpenMP parallelisiert werden kann: die erste Schleife wählt die Reihe und die zweite die Spalte des Pixels. Bei beiden Bildtransformationsoperationen wurde die äußere Schleife parallelisiert, da das in diesem Fall die effizienteste Methode ist, was im Abschnitt 3.1 gezeigt wird.

2.1.1. *RGB zu Grayscale.*

Um ein RGB Bild in ein Grayscale-Bild umzuwandeln, müssen zuerst alle Werte des RGB Pixels zu einem Wert zusammengeführt werden. Danach wird dieser Wert jedem Kanal des RGB-Pixels zugewiesen, wodurch der Pixel einen bestimmten Grauton zugewiesen bekommt. Der Grauwert des Pixels kann zum Beispiel mit der folgenden Formel berechnet werden:

$$gray = 0.21 * r + 0.72 * g + 0.07 * b \tag{1}$$

Wenn man jedoch bedenkt, dass die Pixel-Werte durch ganze Zahlen (Werte zwischen 0 und 255) und nicht durch Float-Zahlen dargestellt werden, kann man die Multiplikation mit Float-Zahlen durch die Division mit ganzen Zahlen ersetzen. Das würde zur folgenden Gleichung führen:

$$gray = r/4 + (g/2 + g/4) + b/14 \tag{2}$$

Die RGB zu Grayscale Funktionalität wurde in Assembler implementiert: deshalb sollte man bei der Division die Zweierpotenzen benutzen, da dies besonders effizient ist. Somit ergibt sich die Folgende Gleichung:

[1] Ob das tatsächlich der Fall ist wird im Kapitel 3.2.1 untersucht

$$gray = r/4 + (g/2 + g/4) + b/16 \tag{3}$$

Der Quellcode der RGB zu Grayscale Funktionalität ist im Listing 1 zu sehen. Die Umwandlung wurde durch OpenMP parallelisiert und zwar nur die äußere Schleife. Als erstes wird der RGB-Wert geladen: anstatt die Werte der RGB-Kanäle einzeln aus dem Speicher zu laden, wird der ganze RGB in den *EAX* Register mit nur einem Speicherzugriff geladen. Danach werden an *EAX* Operationen durchgeführt, die zu denen aus der Gleichung 3 äquivalent sind. Anschließend wird der berechnete Grayscale-Wert in den Speicher des entsprechenden Pixels geschrieben. Ein Beispiel für die Umwandlung mit dieser Implementierung ist auf der Abb. 1 zu sehen.

Listing 1: RGB to Grayscale

```
#pragma omp parallel for
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            uchar* pixel = ptr + channels * (i * cols + j);
            __asm {
                    mov rcx, pixel    //rcx = pixel
                    mov eax, [rcx]    //eax=*rcx   -> eax=bgr

                    shr al, 4     //al=al/16
                    shr ah, 1     //ah=ah/2
                    add al, ah    //al=al+ah
                    shr ah, 1     //ah=ah/2
                    add ah, al   //ah=ah+al

                    shr eax, 8     //shift the r-Value into ah
                    shr ah, 2     //ah=ah/2
                    add al, ah   //al=al+ah
                    jnc label     //check if value <= 255
                    mov al, 255

            label:
                   //*pixel = rgb(al,al,al)
                    mov [rcx], al
                    mov [rcx+1], al
                    mov [rcx+2], al
            }
        }
    }
```

Abb. 1. Umwandlung: RGB zu Grayscale

2.1.2. *Helligkeit.* Die Helligkeit eines Bildes kann verändert werden, indem man eine konstante Zahl mit jedem Kanal von jedem Pixel summiert. Dies kann mit der folgenden Gleichung beschrieben werden:

$$pixel = rgb(r + \alpha, g + \alpha, b + \alpha) \tag{4}$$

Auch bei dieser Operation ist es sinnvoller mit ganzen Zahlen zu arbeiten, da die Kanäle der Pixel durch ganze Zahlen dargestellt werden. Beim Summieren muss man jedoch darauf achten, dass die resultierenden Werte den Wert 255 nicht übersteigen. Im Listing 2 sieht man wie das Anpassen der Helligkeit implementiert wurde. Bei der Implementierung wurden SIMD-Operationen[2] des x86-Assemblers verwendet: zuerst werden die Daten eines Pixels in den mm1-Register geladen und dann wird ein Array bestehend aus den Helligkeitswerten, die dazuaddiert werden sollten, in den Register mm0 geladen. Danach werden beide Register mit nur einer Operation paddusb[3] summiert. Das Ergebnis der Addition wird dann in den Speicher des Pixels geschrieben. Auf der Abb. 2 sieht man ein Beispiel für die Umwandlung eines Bildes mit dieser Funktionalität.

[2]Single Instruction, Multiple Data

[3]Eine SIMD Operation: Add Packed Unsigned Integers with Unsigned Saturation

Listing 2: Helligkeit

```
uchar brightnessArr[4] = {brightness, brightness, brightness, 0};
#pragma omp parallel for
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            uchar* pixel = ptr + channels * (i * cols + j);
            __asm {
                    mov rcx, pixel
                    movd mm1, [rcx]
                    movd mm0, brightnessArr
                    paddusb mm1, mm0
                    movd [rcx], mm1
            }
       }
    }
```

Abb. 2. Umwandlung: Helligkeit + 100

2.2. **Histogramm.** Bei dem der Erstellung des Histogramms eines Bildes zählt man die Kanäle aller Pixel, die denselben Wert haben und erstellt dann eine Abbildung, die die ermittelten Werte darstellt. Da die Kanäle der Pixel den Wertebereich 0-255 haben, gibt es dabei 256 Zähler für jeden Farbkanal des Pixels. Das Erstellen des Histogramms kann in drei Schritte aufgeteilt werden:

(1) Werte der Farbkanäle der Pixel werden gezählt.
(2) Der höchste Wert unter allen Zählern wird bestimmt (Dieser Wert wird beim Zeichnen des Histogramms benötigt, damit das Histogramm richtig skaliert wird).
(3) Das Histogramm wird gezeichnet.

Bei dem ersten Schritt braucht man einen Zähler-Array für jedes Farbkanal. Es ist klar, dass im Falle der Parallelisierung des ersten Schrittes mit *"#pragma omp parallel for"*, es zu Race-Conditions kommen wird, da unterschiedliche Threads gleichzeitig auf den Wert eines bestimmten Zählers zugreifen und diesen manipulieren könnten. Damit keine Race-Conditions entstehen können wird stattdessen *"#pragma omp parallel for reduction"* für die Parallelisierung benutzt. Im Listing 3 kann man sehen, dass die dabei verwendete Reduktionsoperation '+' ist, wobei diese auf die drei Zähler-Arrays angewendet wird. An dieser Stelle könnte man alternativ die Zähler-Arrays bei der parallelisierten For-Schleife

als 'shared' festlegen, jedoch wurde festgestellt, dass die bereits beschriebene 'reduction'-Methode bei der Ausführung effizienter ist[4].

Listing 3: Schritt 1

```
#pragma omp parallel for reduction(+ \
                                   : histogramDataR,\
                                     histogramDataG,\
                                     histogramDataB)
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                uchar* pixel = ptrImg + channels*( i*cols + j);
                histogramDataB[pixel[0]] += 1;
                histogramDataG[pixel[1]] += 1;
                histogramDataR[pixel[2]] += 1;
            }
        }
```

Im zweiten Schritt muss der höchste Wert, der in einem der Zähler-Arrays ist, ermittelt werden. Die Implementierung des zweiten Schritts ist im Listing 4 zu sehen: bei diesem Schritt wurde ebenfalls die 'reduction'-Methode eingesetzt, allerdings mit der 'max' Operation. Der ermittelte Maximal-Wert wird in der *highestCounter*-Variable gespeichert.

Listing 4: Schritt 2

```
       int highestCounter = 0;
#pragma omp parallel for reduction(max : highestCounter)
       for (int i = 0; i < 256; ++i) {
            if (histogramDataB[i] > highestCounter) {
                highestCounter = histogramDataB[i];
            }
            if (histogramDataG[i] > highestCounter) {
                highestCounter = histogramDataG[i];
            }
           if (histogramDataR[i] > highestCounter) {
                highestCounter = histogramDataR[i];
            }
        }
```

Im dritten Schritt wird das Histogramm erstellt. Wie das Histogramm erstellt wird, kann man im Listing 5 sehen. Die Ausführung von diesem Schritt wurde nur durch *#pragma omp parallel for* parallelisiert. Die Größe des Histogramms kann durch den Parameter

[4]Wird im Abschnitt 3.1.2 gezeigt

'scale' festgelegt werden. Ein erstelltes Beispiel-Histogramm ist auf der Abb. 3 zu sehen: blaue Farbe gehört zum b-Kanal, grüne Farbe zum g-Kanal und rote zum r-Kanal.

Listing 5: Schritt 3

```
#pragma omp parallel for
        for (int i = 0; i < 256 * scale; i += 1 * scale) {
            float r = histogramDataR[i / scale];
            float g = histogramDataG[i / scale];
            float b = histogramDataB[i / scale]
            for (int j = 0; j < rows; ++j) {
                uchar* pixel = ptrHist+channels*(j*cols + i);
                float rowTemp = (float)j / histMatrix.rows;
                if (r / highestCounter > rowTemp) {
                    pixel[2] = 255;
                }
                if (g / highestCounter > rowTemp) {
                    pixel[1] = 255;
                }
                if (b / highestCounter > rowTemp) {
                    pixel[0] = 255;
                }
            }
        }
```

Abb. 3. Erstelltes Histogramm

2.3. **Konsolen-User-Interface.** Das implementierte Anwendung lässt sich durch Konsole-Argumente steuern. Folgende Argumente sind zulässig:

Argument	**Beschreibung**
-p [path]	Legt den Pfad zum Bild fest, das untersucht werden soll.
-g	Aktiviert die Umwandlung von RGB zu Grayscale.
-s	Wenn das Original-Bild auch angezeigt werden soll.
-b [brightness]	Addiert den Wert *brightness*(integer) zur Helligkeit.
-t [numOfThreads]	Zum ändern der Anzahl an aktiven Threads.
-pn	Dadurch wird nur die Ausführungszeit ausgegeben[5].
-c	Das Problem wird automatisch geschlossen nach dem die Ausführung fertig ist[5].
-h [scale]	Legt fest ob ein Histogramm erzeugt werden soll und falls ja wie groß(=scale).
-cpp	Sorgt dafür, dass nur reiner C++ Code (=ohne Assembler) ausgeführt wird.

[5]Diese Option ist für das automatisierte Testing notwendig

3. Evaluierung

In diesem Kapitel wird die erstellte Anwendung detailliert evaluiert und untersucht. Bei jedem durchgeführten Test wurde die getestete Funktionalität 50 mal ausgeführt, sodass am Ende ein arithmetisches Mittel von den ermittelten Ausführungszeiten errechnet werden konnte. Die Tests wurden auf einem Rechner mit einem Intel i5 Prozessor der 10 Generation unter macOS durchgeführt.

3.1. **Wahl der Parallelisierungsart.**

OpenMP bietet mehrere unterschiedliche Werkzeuge an, mit denen man die Ausführung von bestimmten Vorgängen parallelisieren kann. Es ist dabei wichtig, dass man die Werkzeuge wählt, die für eine bestimmte Aufgabe geeignet sind.

3.1.1. *Parallelisierung von Bildtransformation-Operationen.*

Bei den beiden Bildtransformation-Operationen (Grayscale und Helligkeit) werden zwei Schleifen gebraucht: eine um die Reihe des Pixel auszuwählen und eine für die Wahl der Spalte des Pixels. Es gibt dabei mehrere Möglichkeiten die einzelnen Pixel mit OpenMP parallel zu verarbeiten. Folgende Varianten wurden bei der Wahl der optimalen Lösung untersucht:

Variante 1

```
#pragma omp parallel for
for (int i = 0; i < rows; ++i) {
    for (int j = 0; j < cols; ++j) {
        ...
    }
}
```

Variante 2

```
for (int i = 0; i < rows; ++i) {
#pragma omp parallel for
    for (int j = 0; j < cols; ++j) {
        ...
    }
}
```

Variante 3

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < rows; ++i) {
    for (int j = 0; j < cols; ++j) {
        ...
    }
}
```

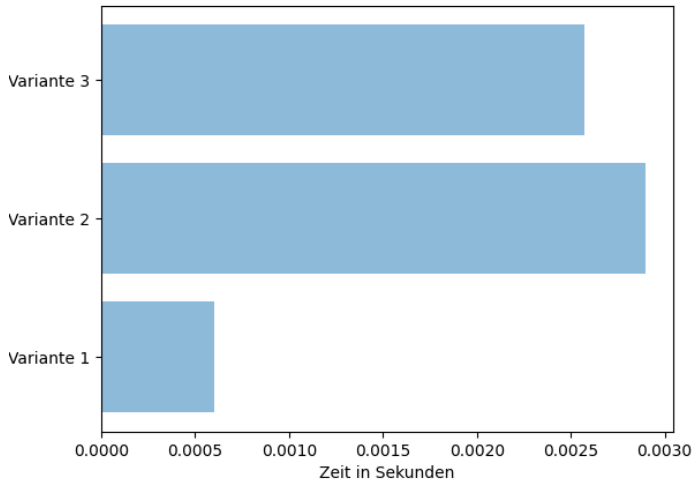

Abb. 4. Vergleich von Varianten der Parallelisierung

Bei der ersten Variante wird die äußere Schleife parallelisiert. Bei der zweiten wird dagegen die innere Schleife parallelisiert. Die dritte Variante wandelt beide Schleifen durch "collapse(2)" in eine Schleife um, sodass aus allen Iterationen beider Schleifen ein großer Iterationsraum gebildet wird, der dann unter den Threads des Programms aufgeteilt werden kann.

Alle drei Varianten wurden untersucht: die Varianten wurden anhand der RGB zu Grayscale Umwandlung getestet, wobei jede Variante 50 mal ausgeführt wurde und ein arithmetisches Mittel zu jeder Variante berechnet wurde. Die Ergebnisse sind auf der Abb. 4 zu sehen. Man sieht, dass die erste Variante eine viel bessere Ausführungszeit hat, weshalb diese Variante bei der Implementierung von den Bildtransformation-Funktionalitäten gewählt wurde. Dass die Ausführungszeiten der vorgestellten Parallelisierungs-Varianten so unterschiedlich sind hängt womöglich damit zusammen, dass die Caching-Mechanismen des Prozessors mit einigen Varianten besser kompatibel sind als mit den anderen.

3.1.2. *Parallelisierung der Histogramm-Erstellung.* Bei der Erstellung des Histogramms ist es so, dass die Threads nicht unabhängig voneinander sind, wie das bei den Bildtransformation-Operationen der Fall war. Aber auch für dieses Problem, bietet OpenMP mehrere Lösungswege an. Es wurden zwei mögliche Lösungen gewählt und untersucht:

- **Shared:** wie der Name bereits verrät, werden bei dieser Variante bestimmte Variablen unter den Threads geteilt, sodass es keine Race-Conditions entstehen können. Wird mit der Pragma-Anweisung *"#pragma omp parallel for shared(%varName%)"* aktiviert.
- **Reduction:** bei dieser Variante werden die Threads zunächst unabhängig ausgeführt, ohne dass die Variablen unter den Threads synchronisiert werden und am

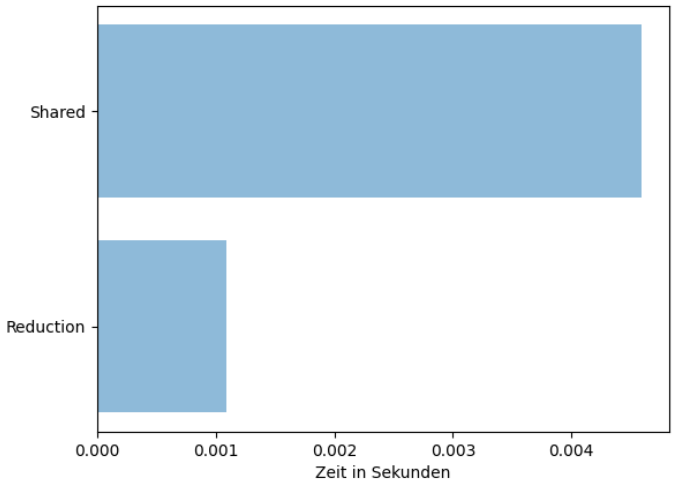

Abb. 5. Vergleich von *shared* und *reduction*

Ende werden die Teilergebnisse der Threads gesammelt und zu einem Ergebnis kombiniert. Wird mit der Pragma-Anweisung *"#pragma omp parallel for reduction(%operation% : %varName%)"* aktiviert.

Beide Varianten wurden getestet und die dabei ermittelten Ausführungszeiten kann man auf der Abb. 5 sehen. Man sieht, dass die Reduction-Variante mindestens vier Mal so schnell ist wie die Shared-Variante, weshalb diese Variante bei der Implementierung gewählt wurde. Dass die Shared-Variante so viel langsamer ist liegt daran, dass die Variable, die von mehreren Threads geteilt wird, unter allen Threads synchronisiert werden muss, weshalb die Ausführung sehr verlangsamt wird. Bei der Reduction-Variante werden die Teilergebnisse der Threads dagegen erst nach dem Beenden der Schleife kombiniert, sodass bei der Ausführung der Schleife nichts synchronisiert werden muss und die Ausführungszeit dadurch viel besser ist.

3.1.3. *Optimale Threadsanzahl.* Bei der parallelen Abarbeitung einer Aufgabe kann man die Anzahl an Threads, die für das Abarbeiten dieser Aufgabe zuständig sind, wählen. Es ist dabei klar, dass von der Anzahl der Threads die Ausführungszeit des Programms abhängig ist. Es ist jedoch so, dass es nicht immer der Fall ist, dass mit einer größeren Threads-Anzahl die bessere Ausführungszeit erzielt wird: wenn die Threads-Anzahl zu groß wird kann es sogar negative Auswirkungen auf die Ausführungszeit haben. Wie viele Threads zu einer optimalen Ausführungszeit führen könnte, wird maßgeblich von den Eigenschaften des Prozessors, wie die Anzahl an Cores oder Threads beeinflusst. Meistens lohnt es sich die Anzahl an Threads auf eine Zahl zu setzen, die größer ist als die Anzahl an Cores des Prozessors, da durch die Cores-Anzahl die Anzahl an parallel ausführbaren Threads festgelegt wird. Es muss jedoch nicht unbedingt der Fall sein, dass die Anzahl

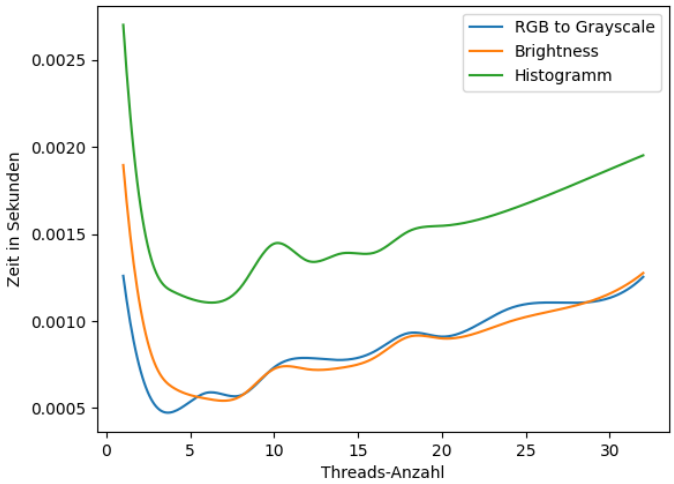

Abb. 6. Zusammenhang der Threads-Anzahl mit der Ausführungszeit

an Cores tatsächlich die optimale Anzahl an Threads für ein Programm ist, da Prozessoren heutzutage sehr komplex sind und außerdem von anderen Komponenten (wie z.B. Arbeitsspeicher und Festplatte) abhängen.

Der Zusammenhang zwischen der Ausführungszeit der implementierten Anwendung und der Anzahl an Threads wurde untersucht. Bei der Ausführung von Tests wurde ein Rechner mit dem i5 Prozessor der zehnten Generation verwendet (4 Cores und 8 Threads), wobei alle drei Funktionalitäten separat voneinander untersucht wurden (RGB to Grayscale, Helligkeit-Änderung und Histogramm). Die Ergebnisse dieser Untersuchung sind auf der Abb. 6 zu sehen. Man sieht, dass alle drei Kurven recht ähnlich zueinander sind, jedoch trotzdem gewisse Unterschiede aufweisen. Die optimale Anzahl an Threads ist bei der *RGB to Grayscale* Operation liegt bei 4 Threads, was mit der Anzahl an Cores des gegebenen Prozessors übereinstimmt. Dagegen liegt die optimale Anzahl von Threads bei der *Helligkeit-* und *Histogramm*-Operation interessanterweise bei 6 Threads, was zeigt, dass die optimale Threads-Anzahl nicht immer mit der Anzahl an Cores des Prozessors übereinstimmen muss.

Es ist außerdem wichtig zu untersuchen welche Auswirkung die Parallelisierung an sich auf das Programm hat: bei Grayscale ist die Ausführung mit der optimalen Anzahl an Threads 162% besser als die Ausführung mit einem Thread und analog beträgt die Verbesserung bei der Brightness Operation 242% und beim Histogramm 144%.

3.2. **Vergleich mit alternativen Implementierungen.**

3.2.1. *Vergleich von Assembler mit C++.* Bei der Implementierung der beiden Bildtransformation-Operationen wurden bestimmte Teile des Programms in Assembler implementiert. Es

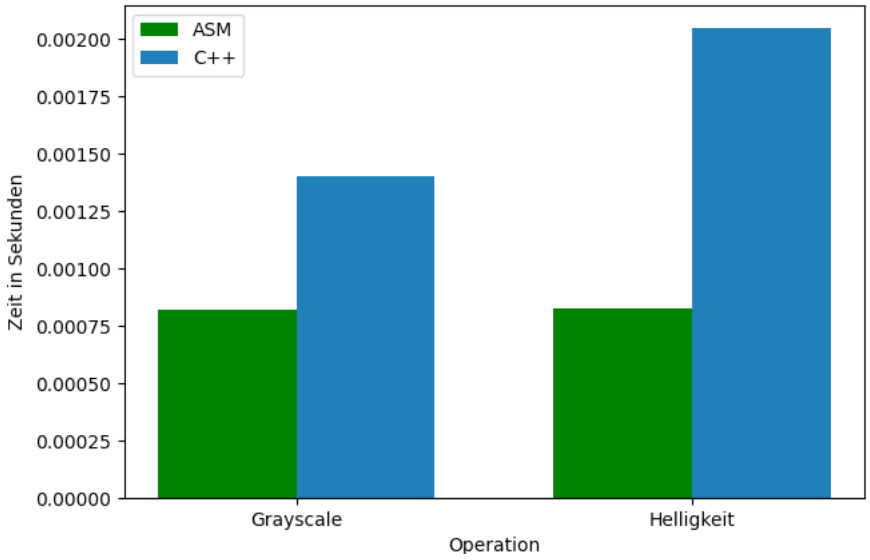

Abb. 7. Vergleich der Ausführungszeit von C++ mit Assembler

wäre interessant zu untersuchen ob die Assembler-Implementierung tatsächlich besser als eine C++ Implementierung ist, denn die modernen Kompilierer besitzen effektive Optimierungsmechanismen, sodass eine Assembler-Implementierung nicht immer effizienter ist als das vom Kompilierer generierte Programm.

Um zu untersuchen, ob die in Assembler implementierte Teile der Anwendung effizienter sind als reiner C++ Code, wurden die entsprechenden Teile der Anwendung in C++ implementiert, sodass man beide Varianten verlgleichen kann. Auf der Abb. 7 sieht man die Ergebnisse: man sieht, dass die erstellte Assembler-Implementierung deutlich bessere Ausführungszeit hat als das äquivalente C++ Programm.

3.2.2. *Vergleich mit OpenCV.* Die Performance der Implementierten Anwendung kann außerdem mit anderen Implementierungen von ähnlichen Funktionalitäten verglichen werden. Mit der OpenCV Bibliothek lassen sich schnell Funktionalitäten erstellen, die zu den Funktionalitäten der implementierten Anwendung äquivalent sind. Die Ergebnisse des Vergleichs kann man auf der Abb. 8 sehen. Man sieht, dass die Grayscale- und die Helligkeit-Operation bei der implementierten Anwendung schneller als bei OpenCV ist, wobei insbesondere bei der Grayscale-Operation der unterschied recht groß ist (159% schneller). Bei der Erstellung des Histogramms ist OpenCV allerdings schneller (34% schneller), was vermutlich durch eine effizientere Implementierung zustande kommt.

4. Zusammenfassung

Im Rahmen dieser Arbeit wurde eine Anwendung entwickelt, die zwei Bildtransformation-Operationen (Grayscale und Helligkeit-Änderung) und eine Operation zur Bildanalyse (Histogramm) enthält. Die Ausführung der implementierten Funktionalitäten wurde dabei

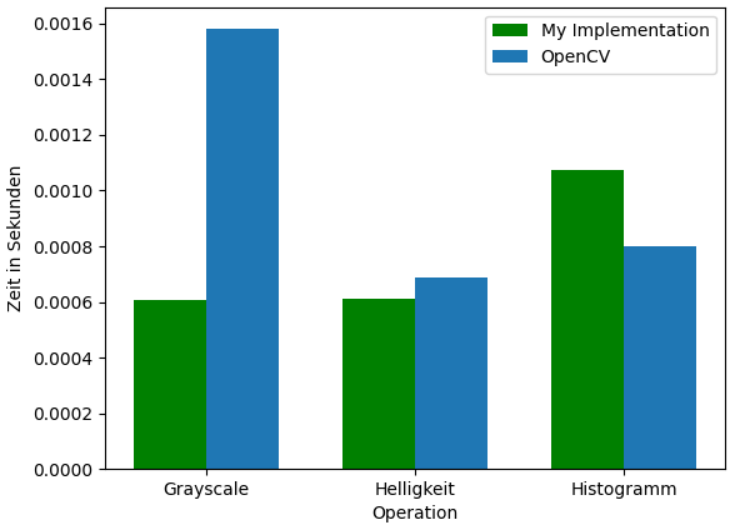

Abb. 8. Vergleich der Ausführungszeit von OpenCV und der implementierten Anwendung

zwecks der schnelleren Ausführung mit OpenMP parallelisiert. Während der Implementierung wurden mehrere unterschiedliche Möglichkeiten der Parallelisierung in Betracht gezogen und analysiert, sodass man für die finale Implementierung die optimalere Methode nehmen konnte.

Des Weiteren wurden einige Teile der Anwendung in Assembler implementiert, was ebenfalls zu einer Verbesserung der Ausführungsgeschwindigkeit geführt hat. Dass die implementierte Anwendung effizient ist, wurde auch durch den Vergleich der beschriebenen Anwendung mit der OpenCV Implementierung gezeigt: beide Bildtransformation Operationen waren effizienter als die OpenCV Implementierung und nur die Histogramm-Berechnung war etwas schlechter als die OpenCV Implementierung.