Grayscale, Histogramm und Helligkeit mit OpenMP

von
Sonja Albers, Enrico Gamil Toros de Chadarevian

Programmierkonzepte und Algorithmen

WiSe 22/23
07.02.2023

Inhaltsverzeichnis

1 Aufgabenbeschreibung

Das Ziel der Aufgabe ist es, PNG- oder JPG-Dateien aus dem RGB-Format mit Hilfe von Parallelisierung in drei andere Ausgabeformate umzuwandeln:

1. ein Grauwertbild
2. ein Histogramm
3. ein Bild mit veränderter Helligkeit

Dabei sollten ursprünglich die Performance-Unterschiede zwischen den Umwandlungen mit und ohne die Nutzung von OpenMP verglichen werden. Nach Rücksprache mit dem Dozenten Prof. Dr. Nikita Kovalenko wurde sich darauf geeinigt, die Aufgabe mit dem Java-internen Runnable-Interface zu lösen. Die vollständige Umsetzung ist auf GitHub einzusehen.

2 Umsetzung der Parallelisierung

Zur Umsetzung wurden eine Blocking- und eine Non-Blocking-Variante genutzt, die beide später miteinander verglichen werden. Die Bilddatei wird aus einem gemeinsamen Speicher ausgelesen, ein threadsicheres Verfahren, bei dem keine zusätzliche Synchronisation hinzugefügt werden muss. Jeder Thread bearbeitet eine eigene Pixelreihe. Beim Blocking-Verfahren wird ein geteilter Speicher genutzt, in dem die einzelnen Threads ihre Ergebnisse direkt und synchronisiert eintragen. Dadurch werden Race-Conditions verhindert und es sorgt für weniger Overhead beim Abruf der Ergebnisse nach der Ausführung. Allerdings führt diese Variante auch zu einer höheren Threadlaufzeit, da zwar alle Threads parallel rechnen können, aber nicht schreiben. Bei der Non-Blocking-Variante hat jeder Thread einen eigenen Speicher für die berechneten Werte, wodurch auch hier Race-Conditions vermieden werden. Sobald alle Threads mit der Berechnung durch sind, werden die Ergebnisse zusammengeführt. Es wird jedoch zusätzlich Zeit für die Ergebniszusammenfassung benötigt. Die ganze Parallelisierung und Synchronisation wird durch das Java-Interface „Runnable“ ermöglicht. Der von den Threads auszuführende Code muss lediglich in die run-Methode geschrieben werden.

3 Umsetzung der Bildverarbeitung

Der Code enthält für jede Bildverarbeitungsaufgabe eine Processor-Klasse und jeweils drei Task-Klassen, eine abstrakte Klasse, in der der Algorithmus zur Berechnung enthalten ist und zwei weitere, die in Blocking und Non-Blocking eingeteilt sind. Über die Processor-Klasse wählt man die Art der Verarbeitung und legt die Threadanzahl fest. Daraufhin kann man ein Bild generieren, wobei man angeben muss, welcher Task-Type dafür zum Einsatz kommen soll - Blocking oder Non-Blocking. Die genaue Struktur ist in Abbildung 1 zu sehen.

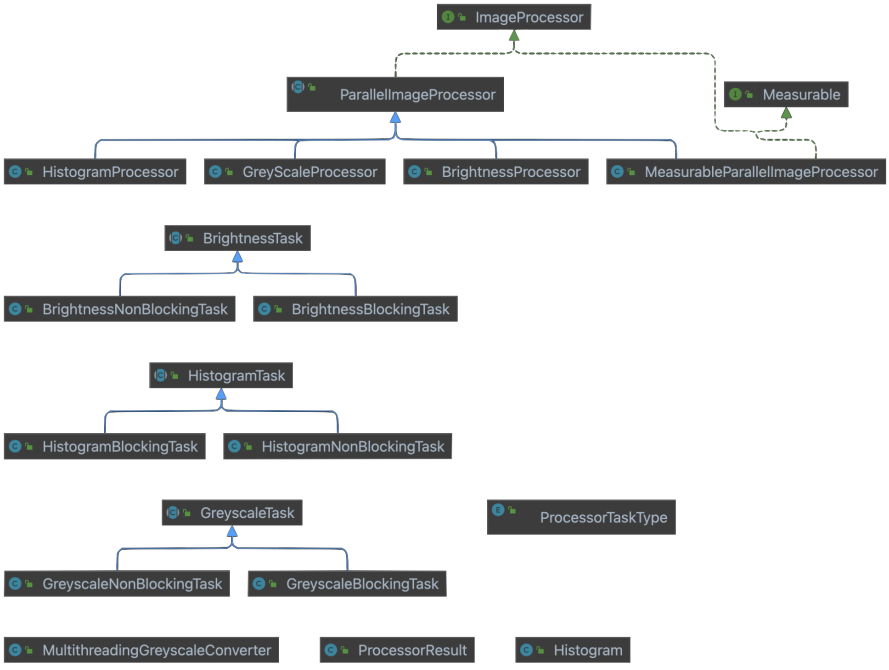


Abbildung 1: Klassendiagramm des Projekts.

4 Grauwertbild

Zur Grauwertberechnung wird mit der Bitshift-Methode gearbeitet. Jede Zeile des Bildes wird durchlaufen und jeder einzelne Pixel betrachtet. Dabei werden die einzelnen Farbkanäle extrahiert und die Farben rot und grün um 16 bzw. 8 Bits nach rechts verschoben. Der Alphakanal wird nicht betrachtet. Alle drei Farben erhalten jeweils eine Maskierung für ihren Wertebereich. Daraufhin wird jeder Farbwert mit einer festgelegten Gewichtung multipliziert. Am Ende werden die Werte auf ihre ursprüngliche Position verschoben und das Ergebnis für diesen Pixel wird als 32-Bit-Wert gespeichert. Ob das Ergebnis direkt in der finalen Datei gespeichert wird oder erstmal zwischengespeichert und später ausgelesen, hängt davon ab, ob man einen Blocking- oder Non-Blocking-Task-Variante verwendet.

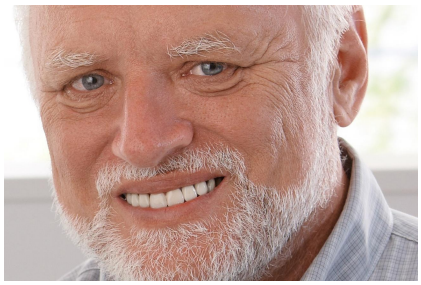


Abbildung 2: Originalbild.

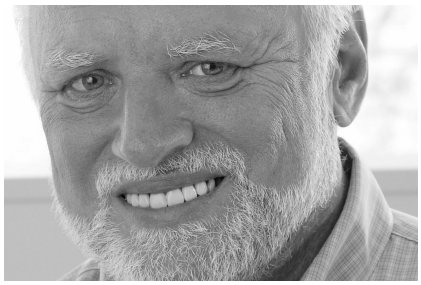


Abbildung 3: Grauwertbilderstellung.

Grauwertbilderstellung - die Bilder sind bei Blocking und Non-Blocking identisch.

```
1       protected int[] greyscaleTask() {
2          int[] row = new int[imgRgbArray.length];
3          for (int i = 0; i < row.length; i++) {
4             int pixel = imgRgbArray[i][rowIndex];
5             int red = (pixel >> 16) & 0xFF;
6             int green = (pixel >> 8) & 0xFF;
7             int blue = (pixel) & 0xFF;
8             int rgbRes = (int) (red * 0.21 + green * 0.72 + blue *
                  0.07);
9             row[i] = ((0xFF) << 24) | // alpha not needed
10                  ((rgbRes & 0xFF) << 16) |
11                  ((rgbRes & 0xFF) << 8) |
12                  ((rgbRes & 0xFF));
13         }
14         return row;
15      }
```

Code zur Berechnung der Grauwerte.

5 Histogramm

Für die Berechnung des Histogramms benötigt man die einzelnen Farbwerte eines jeden Pixels und deren Häufigkeit. Dabei wird für jede Farbe - Rot, Grün und Blau - ein sogenannter „Bucket“ erstellt, in dem die Werte gespeichert werden. Für die Erstellung des Histogramms werden die Werte normalisiert. Das ermöglicht eine einheitliche Darstellung und besseren Vergleich der unterschiedlichen Farbwerte. Anders als bei den anderen beiden Bearbeitungsmethoden, ist der Code zur Ermittlung der Werte in der jeweiligen Blocking-Task- und Non-Blocking-Task-Klasse hinterlegt. Bei der Blocking-Variante werden die Werte direkt synchronisiert in der Histogramm-Einheit gespeichert. Die Non-Blocking-Variante hingegen extrahiert die Werte für jeden Farbkanal und überreicht diese danach der Histogramm-Klasse.

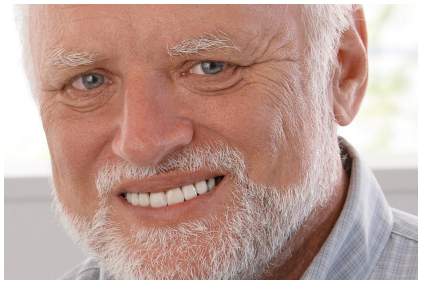


Abbildung 4: Ausgangsbild.

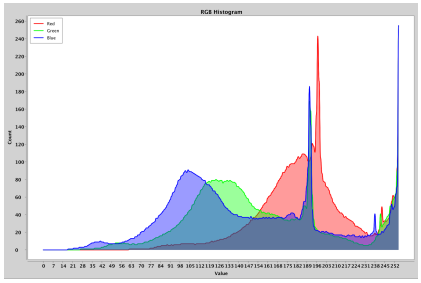


Abbildung 5: Generiertes Histogramm.

Bei Blocking und Non-Blocking identisch.

```
1    @Override
2    public void run() {
3        for (int[] rows : imgRgbArray) {
4            Color c = new Color(rows[rowIndex]);
5            this.redBucket[c.getRed()]++;
6            this.greenBucket[c.getGreen()]++;
7            this.blueBucket[c.getBlue()]++;
8        }
9    }
```

Ausschnitt Histogramm-Ermittlung mit Non-Blocking.

Die Blocking Variante ist identisch, wobei jedes Thread eigene Buckets besitzt (lokaler speicher), welche nach der Ausführung zusammengefasst werden.

```
1   private void mergeResults(LinkedList<int[]> redBuckets,
        LinkedList<int[]> greenBuckets, LinkedList<int[]>
        blueBuckets) {
2       // each thread bucket is merged
3       for (int i = 0; i < 256; i++) {
4           for (int[] bucket : redBuckets) {
5               resultRedBucket[i] += bucket[i];
6           }
7           for (int[] bucket : greenBuckets) {
8               resultGreenBucket[i] += bucket[i];
9           }
10          for (int[] bucket : blueBuckets) {
11              resultBlueBucket[i] += bucket[i];
12          }
13      }
14  }
```

6 Helligkeitsveränderung

Um die Helligkeit zu verändern, muss beim Aufruf ein Double-Wert für das Ausmaß der Veränderung angegeben werden. Wie bei der Grauwertbildgenerierung wird auch hier zu Beginn ein Bit-Shift für Rot und Grün um 16 bzw. 8 Bits durchgeführt. Dann wird für jeden Farbkanal reihenweise der Pixelwert mit dem Eingabewert multipliziert. Damit die neuen Werte im gültigen Bereich von 0 bis 255 liegen, werden alle höheren Werte auf 255 und niedrigere Werte auf 0 gesetzt. Am Ende wird der Bit-Shift wieder rückgängig gemacht.

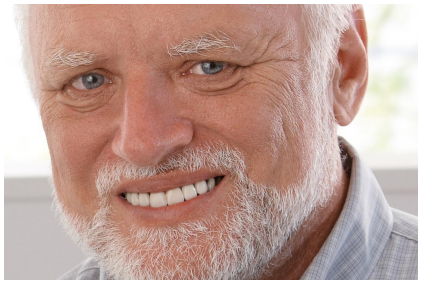


Abbildung 6: Ausgangsbild.

Abbildung 7: Ergebnis.

Helligkeitsveränderung mit einem Wert von 2.0 - die Bilder sind bei Blocking und Non-Blocking identisch.

```
1   protected int[] brightnessTask(int[][] imgRgbArray, int rowIndex
        , double brightness) {
2       int[] row = new int[imgRgbArray.length];
3       for (int i = 0; i < row.length; i++) {
4           int pixel = imgRgbArray[i][rowIndex];
5           int red = (pixel >> 16) & 0xFF;
6           int green = (pixel >> 8) & 0xFF;
7           int blue = (pixel) & 0xFF;
8           // increase brightness
9           red *= brightness;
10          green *= brightness;
11          blue *= brightness;
12          // truncate
13          red = Math.min(Math.max(red, 0), 255);
14          green = Math.min(Math.max(green, 0), 255);
15          blue = Math.min(Math.max(blue, 0), 255);
16          row[i] = (red << 16) | (green << 8) | blue;
17      }
18      return row;
19  }
```

7 Tests

Um zu erfahren, ob tatsächlich Threads gearbeitet wird, kann man sich in der IntelliJ-IDE die einzelnen Threads anzeigen lassen. Dabei steht Rot für Momente, in denen Samples von der CPU erstellt wurden und Blau für Speicherzuweisungen. Diese Darstellungen können dabei helfen, kritische Stellen im Code ausfindig zu machen. Für dieses Projekt wurde es lediglich zur Darstellung der Threads und zum Vergleich der Arbeitsweise der unterschiedlichen Task-Varianten genutzt.

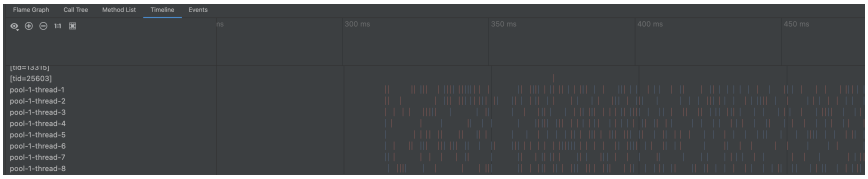


Abbildung 8: Darstellung der Threads für Brightness mit Blocking.

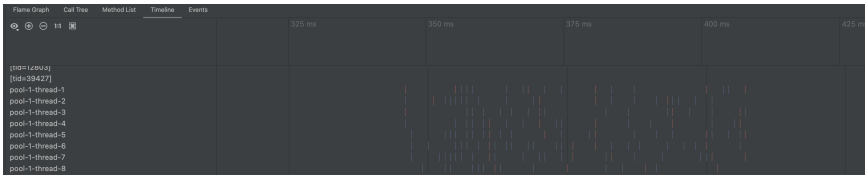


Abbildung 9: Darstellung der Threads für Brightness mit Non-Blocking.

Für jede Bearbeitungsweise (Grauwert, Histogramm und Helligkeitsveränderung) wurde der Code sowohl für Blocking als auch Non-Blocking getestet. Dafür wird eine Datei, die dem gewünschten Ergebnis entspricht, als Vergleichsobjekt hinterlegt. Diese Datei kann beispielsweise mit einer anerkannten Bibliothek wie OpenCV generiert worden sein. Daraufhin werden sowohl die Blocking-Task- als auch die Non-Blocking-Task-Varianten aufgerufen und jeder einzelne Pixelwert des Ergebnisses mit dem der Vorlage verglichen. Zusätzlich erfolgt diese Auswertung 1000 Mal, um sicherzugehen, dass der Code bei Wiederholung identische Ergebnisse liefert und ein Mittelwert zu berechnen. Dieses Testen wurde um einen CSV-Writer ergänzt, der Informationen zur Laufzeit notiert. Dadurch ergeben sich Informationen, die für die Auswertung des Projekts hilfreich sind. Die Zerstreuung wurde hier nicht explizit angegeben, da diese nicht relevant ist (siehe Abbildung 11). Diese können auf GitHub untern den Testdaten gefunden werden.

8 Auswertung

Für die Auswertung wurden die durch den CSV-Writer gespeicherten Daten visualisiert und interpretiert. Während der Ausführung waren keine weiteren Programme geöffnet. Dabei wurde der Code auf folgenden Geräten ausgeführt:

MacBook Pro (2015)	**MacBook Pro (2020)**
2,9 GHz Dual-Core Intel Core i5 Prozessor	Apple M1 Chip
8 GB Arbeitsspeicher	16 GB Arbeitsspeicher

Die dargestellten Ergebnisse beziehen sich auf folgendes Bild mit den Maßen 1300 x 867 Pixel und einer Dateigröße von 172 KB:

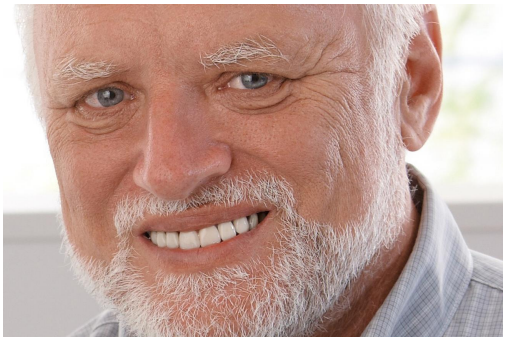


Abbildung 10: Beispielbild mit dem die Berechnungen durchgeführt wurden.

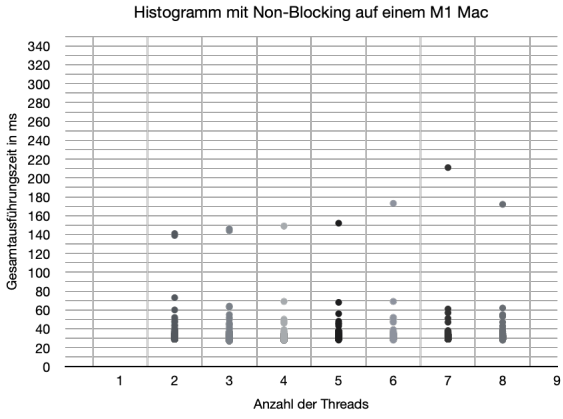
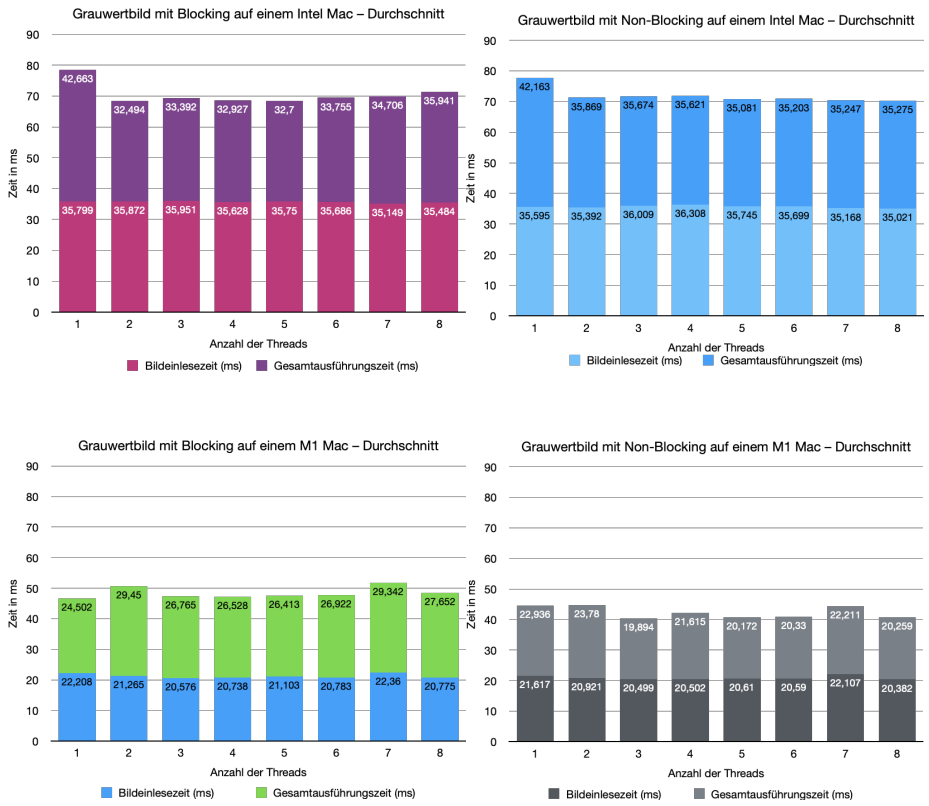


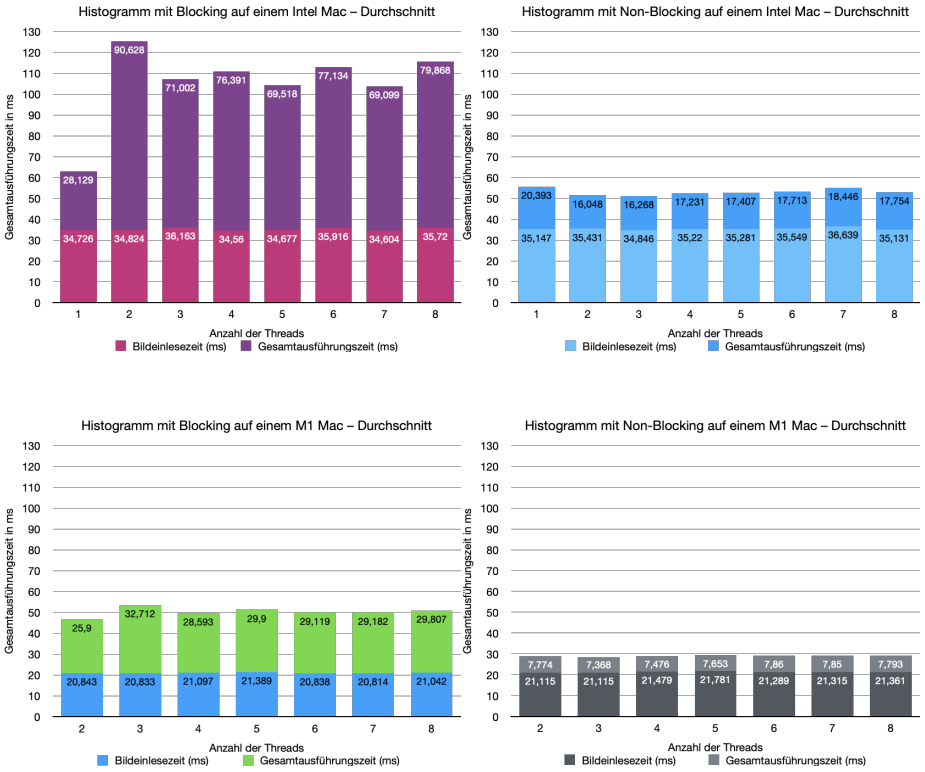
Abbildung 11: Streuungsmaß

8.1 Vergleich Grauwert



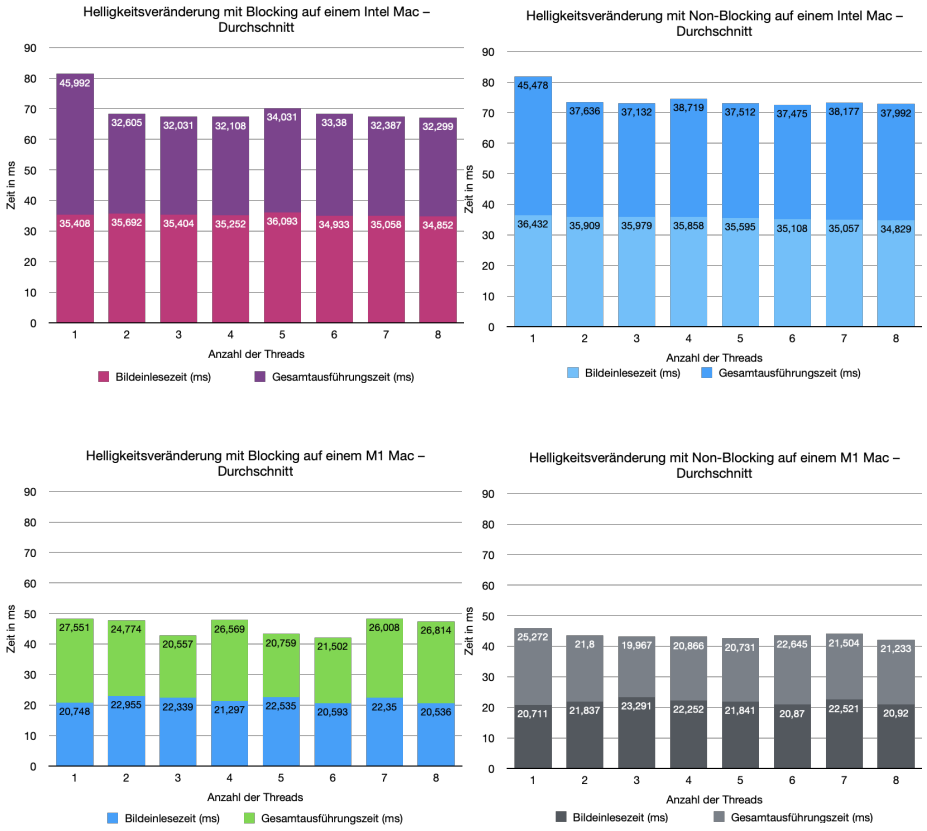
Es wird deutlich, dass die Einlesezeit etwa die Hälfte der Bearbeitungszeit ausmacht, wobei sie beim M1-Mac etwa 10 ms niedriger ist als beim Intel-Mac. Bei der ausgewählten Thread-Anzahl gibt es beim Intel-Mac keine signifikanten Unterschiede zwischen Blocking und Non-Blocking. Auf dem M1-Mac ist die Non-Blocking-Variante durchgängig etwas schneller. Die Laufzeit ist bei einem Thread auf dem Intel-Mac am höchsten. Daraus wird deutlich, dass eine Parallelisierung auch bei weniger komplexen Aufgaben bei diesem Gerät zu Leistungsverbesserung führt. Beim M1-Mac liegt die höchste Bearbeitungszeit jeweils bei zwei Threads. Auch bei sieben Threads ist die Bearbeitungszeit höher als bei den meisten anderen Thread-Anzahlen.

8.2 Vergleich Histogramm



Durch einen Messfehler sind bei der Histogrammerstellung mit dem M1-Mac keine Daten für die Ausführung mit nur einem Thread vorhanden. Bei der Blocking-Variante auf dem Intel-Mac zeigt sich jedoch, dass die Parallelisierung mit dem Blockieren zu einer Laufzeitverlängerung führt. Vermutlich liegt es daran, dass direkt in der Datei gespeichert wird und sich die Threads gegenseitig blockieren. Die Non-Blocking-Variante ist parallelisiert etwas schneller als mit nur einem Thread. Das erzeugte Histogramm ist dennoch das Gleiche wie beim Non-Blocking, nur werden hier die Werte parallel für die einzelnen Farben ermittelt. Auch bei dieser Aufgabe zeigt sich, dass der M1-Mac insgesamt deutlich schneller als der Intel-Mac ist.

8.3 Vergleich Helligkeitsveränderung



Bei der Helligkeitsveränderung zeigt sich bei allen Varianten eine Verbesserung der Laufzeit durch die Parallelisierung. Eine Abhängigkeit von der Thread-Anzahl lässt sich von den Diagrammen nicht ablesen. Außerdem wird auch hier deutlich, dass der M1-Mac deutlich schneller arbeitet.

9 Fazit

Unter den drei Aufgaben ist die Histogrammberechnung ein Ausreißer, da sich die Blocking-Werte deutlich von den Non-Blocking-Werten unterscheiden und die Ausführung mit Blocking bei mehreren Threads länger dauerte als bei nur einem Thread, zumindest beim Intel-Mac. Ansonsten waren die parallelisierten Prozesse schneller, wobei es kaum Unterschiede zwischen Blocking und Non-Blocking gab. Zusätzlich wurde deutlich, dass der M1-Mac Aufgaben schneller

ausführt als der Intel-Mac. Dies hängt vermutlich mit den unterschiedlichen Prozessoren zusammen. Es fand außerdem ein Vergleich mit OpenCV statt. Die Berechnungen mit dieser Bibliothek erfolgten deutlich schneller als mit dem eigenen Code. Die Ergebnisse dafür wurden nicht festgehalten, da die Ausführung persönlichem Interesse diente und nicht viel über Parallelisierung und Prozessen ausgesagt hätte. In diesem Fall wäre eine Betrachtung des genutzten Algorithmus und die allgemeine Implementierung sinnvoller gewesen. Außerdem wurde der Code auch mit anderen Beispielbildern verwendet. Die Bilddateien und ein Teil der Ergebnisse sind im GitHub-Repository zu sehen. Die Analysen für dieses Projekt können noch weitergeführt werden. Es sollte eine Nutzung von mehr Threads betrachtet werden, um feststellen zu können, ob es einen allgemeinen Trend zur Laufzeitverbesserung gibt, was man zumindest vermuten kann. Außerdem kann die bisher zeilenweise Verarbeitung umgeschrieben werden zu einer Batch-weisen Verarbeitung, bei der direkt zusammenhängende Bildbereiche den Threads zugeordnet werden, ähnlich wie bei OpenMP. Dadurch könnte ebenfalls eine Verbesserung der Laufzeit erfolgen, da die zu bearbeitenden Reihen bereits klar festgelegt sind.