LEHRSTUHL FÜR RECHNERTECHNIK UND RECHNERORGANISATION

Aspekte der systemnahen Programmierung bei der Spieleentwicklung

Projektaufgabe - 15: Pixelwiederholung

Saman Miran
Robin Ostner
Maria Schulze

1 Einleitung

Diese Ausarbeitung wird im Rahmen der Veranstaltung "Aspekte der systemnahen Programmierung bei der Spieleentwicklung" angefertigt, in dem Studierende des Studiengangs Informatik: Games Engineering an der Technischen Universität München mit der Aufgabe konfrontiert werden, ein zufällig zugeteiltes Problem algorithmisch via ARM-Assemblerprogrammierung zu lösen. Diese Ausarbeitung wird erstellt von der Projektgruppe 15 mit der Projektaufgabe "Pixelwiederholung", in der der *Nearest Neighbor*-Algorithmus zur Skalierung von Bildern umgesetzt werden soll.

2 Problemstellung und Spezifikation

Das Pixelwiederholungsverfahren (engl. *Nearest Neighbor*) wird zur Vergrößerung digitaler Bilder benutzt. Ein Ausgangsbild wird um einen beliebigen Faktor skaliert und die einzelnen Pixel des Originals an neue Positionen gerückt. Die resultierenden Lücken zwischen den Originalpixeln werden mit den Farbwerten des nähesten Pixels gefüllt.
In der vorliegenden Aufgabenstellung wurde das Projektteam 15 dazu angehalten, zwei Image-Processing-Algorithmen und ein I/O-Rahmenprogramm zu implementieren, welche digitale Bilder im Bitmap-Format einlesen, bearbeiten und schreiben können.
Beide Image-Processing-Algorithmen sind in ARM-Assembler zu implementieren. Der erste soll das original eingelesene Bild auf eine beliebige Größe zuschneiden, der zweite den oben beschriebenen Pixelwiederholungsalgorithmus umsetzen. Das Rahmenprogramm, geschrieben in C, sorgt für das korrekte Einlesen der Bilddatei, die Übergabe der sequentiellen Bilddaten, sowie Parameter an die Assemblerroutinen und Schreiben einer neuen Bitmap-Datei mit den neuen, von den Assemblerfunktionen berechneten, Werten.
Während dem Projektteam beim Rahmenprogramm keine Vorgaben für Struktur und Aufbau gemacht wurden, wurde für die Image-Processing-Algorithmen spezifiziert, wie sie aussehen sollten:
Der zuschneidende Algorithmus sollte durch die Funktion `void window(char* data, int x, int y, int width, int height)` implementiert werden. Der Eingabeparameter `data` stellt einen Pointer auf die sequentiellen Bilddaten des Originalbildes als `char`-Array dar. Die beiden Integer `width` und `height` sind jeweils Breite und Höhe, auf welche das Bild zugeschnitten werden soll, in Pixeln, während `x` und `y` einen Offset für

x- und y-Koordinate darstellen, ebenfalls in Pixeln. Das Projektteam entschied sich dazu, von der vorgegebenen Funktion etwas abzuweichen: es wurde ein weiterer Parameter hinzugefügt, die originale Bildbreite in Pixeln als Integer (`ogWidth`), und ein `char*` als Rückgabewert (siehe 5.1).
Bei `void zoom(char* data, int org_width, int org_height, int factor)` sollte das *Nearest Neighbor*-Verfahren umgesetzt werden. Auch `zoom(...)` soll einen Pointer auf die Bilddaten als `char`-Array übergeben bekommen und die Integer-Werte der Breite und Höhe (in Code umbenannt zu `width` und `height`), sowie den Skalierfaktor. Nach Absprache mit einem Tutor, ist das Projektteam bei der Implementierung etwas abgewichen. Auch hier wurde ein Rückgabewert zugefügt (siehe 5.1).
Nach Implementierung des Algorithmus sollte mittels Benchmark auf Performanz geprüft werden. Eine optionale Bonusaufgabe sieht vor, dass der Algorithmus auch Dateien im JPEG-Format bearbeiten können sollte, diese wurde allerdings vernachlässigt.

3 Lösungsfindung

Das Projektteam entschied sich zunächst dazu, eine Version des Algorithmus in C-Code zu implementieren, um eine funktionierende Basis für den Assemblercode zu erstellen. Zunächst wurde der Input und Output des Rahmenprogramms implementiert, sprich das Einlesen und Ausschreiben von Dateien im BMP-Format, woraufhin ebenfalls die Funktionen `window(...)` und `zoom(...)` anfänglich ihre Implementierung auf einer C-Grundlage fanden.
Die Umsetzung von `window(...)` war insofern eine Herausforderung, dass das Team sich gezwungen sah, die Originalbreite des Bildes als Parameter mitzugeben. Über eine mögliche Alternative wurde diskutiert. Es wurde allerdings keine Lösung gefunden.
Eine wesentlich größere Herausforderung stellte das Herz der Aufgabenstellung dar: die Umsetzung des Pixelwiederholungsalgorithmus. Während die Skalierung der Originalpixel auf die richtige Position kaum Schwierigkeiten bereitete, suchte das Team nach mehreren Lösungsansätzen für das *Nearest Neighbor*-Auffüllverfahren. Der erste Ansatz, mit mehreren `for`-Schleifen durch das Bild zu iterieren und nach zu füllenden Pixeln zu suchen, wurde aufgrund Komplexität und mangels Effizienz des Codes schnell verworfen. Die Lösung, die es letztendlich in die Implementation geschafft hat, ist weiter unten dokumentiert (siehe Punkt *Dokumentation der Implementierung*).
Nach ausgiebigem Testen des C-Codes wurde auf dessen Grundlage der Algorithmus in Assemblercode implementiert. Anfänglich wurden für diese Umsetzung vornehmlich ARM-Register genutzt. Erst nach korrekter Ausführung des Algorithmus wurde der Code durch Nutzung der SIMD-Register optimiert.

4 Dokumentation der Implementierung

4.1 Implementierung des Rahmenprogramms

Das Rahmenprogramm besteht aus zehn Funktionen, davon fünf externe Verlinkungen auf Assemblerfunktionen und eine `main(...)`. Im folgenden Abschnitt werden die unverlinkten Funktionen des C-Codes dokumentiert. Die Dokumentationen der ersten vier Verlinkungen sind den Unterpunkten 4.2 und 4.3 zu entnehmen. Die fünfte ist ein Verbesserungsvorschlag, welcher in 5.4 beschrieben wird. Der eingebaute Debug-Modus sei in diesem Unterpunkt vernachlässigt und findet seine Beschreibung in 4.4.

Zunächst werden im Rahmenprogramm folgende statische Variablen global deklariert bzw. initialisiert:

```
char* data;
int ogWidth;
int ogHeight;
char* filenameInput = "lena.bmp";
char* filenameOutput = "copy.bmp";
int width = -1;
int height = -1;
int x = 0;
int y = 0;
int factor = 3;
int executionMode = 0;
```

Anschließend iteriert eine `for`-Schleife über die `argc` und `argv` Parameter der `main(...)`-Methode und überschreibt je nach Input diese Variablen und fängt darauf ungültige Eingaben ab. Bei `filenameInput` handelt es sich um den Namen der einzulesenden Datei - dieser ist standardmäßig *lena.bmp* (mitgeliefert in der Aufgabenstellung). Bei ungültigem Dateiformat oder Nicht-Vorhandensein wird das Programm beendet. Folglich ist `filenameOutput` der Name der Datei, die am Ende des Programms geschrieben wird und wird, ohne Input, per default als *copy.bmp* gespeichert. Die Variablen `width`, `height`, `x` und `y` sind die Variablen der neuen Bilddimensionen, die in die Funktionen der Pixelwiederholungsalgorithmen gegeben werden. `width` und `height` werden entweder mit dem Input überschrieben oder auf die Originalgröße des eingelesenen Bildes gesetzt, bleiben die Werte auf -1 oder ist der Wert größer als das Originalbild so wird das Programm beendet. Analog dazu `x` und `y`, ihre default-Values sind jedoch 0, da bei fehlendem Input das Bild effektiv nicht zugeschnitten wird. `factor` ist der Skalierungsfaktor, per default ist dieser 3. Mit `executionMode` kann man zwischen den verschiedenen Implementierungen der *Nearest Neighbor*-Algorithmen (C-Version; Assembler ohne Optimierung [SISD]; Assembler mit Optimierung [SIMD]) wechseln. Ohne entsprechenden Input ist diese Variable auf 0 gesetzt und ruft somit die für die Aufgabenstellung relevante SIMD-Assemblerversion `window(....)` und `zoom(...)` auf. Eine genaue Beschreibung über die einzelnen Command Line Arguments und wie sie zu

nutzen sind, sind den Kommentaren der Datei *main.c* zu entnehmen.
Innerhalb der Validitätsabfragen wird die erste Funktion `static int readBMP(char *filename)` aufgerufen, welche für das Einlesen der eingegebenen Bilddatei zuständig ist. Via Filestream wird auf die in der Variable `filenameInput` spezifizierte Datei zugegriffen und ihr Header ausgelesen, um die Höhe und Breite des Bildes zu extrahieren. Diese werden in den statischen Integer-Variablen `height` und `width` abgespeichert. Die Breite wird insbesondere dafür benötigt das Padding, die aufgefüllten Nullbytes jeder Zeile in Bitmaps, auszurechnen. Auf Grundlage dieser Berechnungen wird daraufhin der nötige Speicherplatz alloziert und die Bilddaten in der statischen Variable `image` als `char`-Array gespeichert.

Aufgrund der Möglichkeit den Algorithmus in verschiedenen Versionen laufen zu lassen, wird die eigentliche Pixelwiederholung in zwei `switch`-Statements abgehandelt. Hier kommt der Wert, den die Variable `executionMode` annimmt ins Spiel. `windowC(...)` und `zoomC(...)` sind die anfänglich implementierten C-Algorithmen für `executionMode = 2`; `windowSISD(...)/zoomSISD(...)` werden für `executionMode = 1` ausgeführt und `window(...)/ zoom(...)` analog.
Zusätzlich sind Benchmarks implementiert: Für die `window`- und `zoom`-Funktion wird jeweils in der gewählten Version(C/SISD/SIMD) die Zeit (CPU-Time) gemessen, wie lange der Algorithmus zur Beendigung des Prozesses braucht und anschließend im Kernel auf 6 Nachkommastellen gerundet ausgegeben. Die C-Compiler-eigene Funktion `clock()` wird hierbei zur Messung genutzt.
Nach erfolgreichem Durchlauf der beiden Algorithmen wird die Funktion `static int writeBMP(char* filename, char* data, int width, int height)` aufgerufen, welche aus den errechneten Daten eine neue Bitmap-Datei schreibt. Dafür wird zunächst der Fileheader und dessen Infoheader entsprechend der Spezifikation des BMP-Dateiformats erstellt. Anschließend wird die benötigte Dateigröße berechnet (Breite mal Höhe mal 3; da RGB-Farbwerte). Weiterhin wird auf passendes Padding geprüft, da das BMP-Dateiformat verlangt, dass die Byteanzahl jeder Zeile ein Vielfaches von 4 ist. Sollte dies nicht passend sein, so werden die restlichen Bytes mit Nullen gefüllt; sollte es passend sein, wird das Array direkt geschrieben.
Die Implementierungen der Methoden `windowC(...)` und `zoomC(...)` sind Grundlage der Assemblerroutinen `window(....)/windowSISD(...)` und `zoom(....)/ zoomSISD(...)` und analog zu diesen, ergo sind ihre Dokumentationen den nachfolgenden Unterpunkten zu entnehmen.

4.2 Implementierung von `window(...)`

`char* window(char* data, int x, int y, int width, int height, int ogWidth)` existiert in drei verschiedenen Versionen: einer C-Version, einer SISD-Assemblerversion und einer SIMD-Assemblerversion. Im Folgenden werden die Assemblerroutinen näher beschrieben. Die C-Version ist in ihrer Umsetzung und Funktionsweise analog zur SISD-Version.
Zu Anfang werden die benötigten Register `r4 - r11` auf den Stack geschrieben. Auf-

grund der sechs Eingabeparameter, werden die letzten beiden (`int height` und `int ogWidth`) vom Stack auf jeweils die Register `r4` und `r5` geladen.
Der Pointer in `r0` wird mittels Offset `int x` und `int y`, sowie der neuen Breite `int width` und neuen Höhe `int height` an die Stelle gesetzt, an der das Bild zugeschnitten werden soll. Register `r11` und `r12` werden dabei verwendet, allerdings nur für Nebenrechnungen.
Anschließend wird durch die `malloc`-Funktion der GNU Compiler Collection (GCC) Speicher auf dem Heap für das neue, zugeschnittene Bildarray alloziert. Die caller-save-Register `r0 - r3` werden vorher auf den Stack gepusht, um Überschreibungen zu vermeiden. Die Arraygröße wird dynamisch berechnet, richtet sich nach `width` und `height` (Formel: `width * height *3`; das Array muss 3x so groß sein, da RGB-Werte einzeln gespeichert werden). Der Wert wird in `r11` gespeichert. Nach Allozierung des Speicherplatzes wird der Pointer auf das neue Array in `r10` gespeichert. Letztendlich werden die caller-save Register vom Stack wiederhergestellt.
Um den letzten Pointer des neu allozierten Arrays zu bekommen, wird auf Register `r11` das Register `r10` addiert. Daraufhin fängt das eigentliche Zuschneiden an. Ab diesem Schritt weichen die Implementierungen der SISD und SIMD Versionen von einander ab. Während beide Versionen zwei Schleifen nutzen und die Pointer auf das neue und alte Array auf jeweils `r1` und `r2` laden, weicht die SISD-Version insofern ab, dass sie jeden Farbwert nur einzeln schreiben kann, die SIMD-Version hingegen schreibt durch Nutzung der `d`-NEON-Register 8 Byte auf einmal.
Die Wertinterpretationen der Register für die SISD-Iteration sind folgende:

```
r0 = data (von Eingabeparameter)
r1 = alloziertes, neues Array
r2 = Pointer auf Element im alten Array in Bearbeitung
r3 = width (zuzuschneidende Bildbreite)
r4 = ogWidth * 3 (originale Bildbreite multipliziert mit 3)
r5 = ogWidth (originale Bildbreite)
r6 = Zwischenspeicher für RGB-Werte
r7 = /
r8 = /
r9 = Zähler der y-Schleife
r10 = Pointer auf das erste Element im neuen Array
r11 = letzter Pointer in r1
r12 = Nebenrechnungen
```

Die beiden Schleifen befüllen das neue Array Byte für Byte mit den Werten aus dem alten. Die RGB-Werte der einzelnen Pixel werden vom alten Bild von Index `r2` auf `r6` geladen und von dort aus auf `r1` gespeichert. Die Pointer in `r0` und `r2` werden in der äußeren `loopY`-Schleife angepasst und verschoben. Der Pointer `r1` wird durch Post-Inkrement in den Speicheroperationen verschoben. Die innere `loopX`-Schleife wird beendet, wenn der Zähler `r9`, welcher mit jeder Iteration inkrementiert wird, denselben Wert wie die zuzuschneidende Bildbreite `r3` hat. Die äußere `loopY`-Schleife wird dann beendet, wenn Pointer `r1` und `r11` sich gleichen. Letztlich wird der Pointer in `r10` auf `r0` geschrieben, damit das zugeschnittene Array zurückgegeben werden kann. Am Ende werden die

callee-save-Register wieder vom Stack genommen.
Die optimierte SIMD-Version verfährt ähnlich. Die äußere `loopX`-Schleife wurde in zwei weitere Labels aufgeteilt: `SIMD` und `SISD`, wobei letzteres Sonderfälle abfängt und genauso operiert, wie die SISD-Version. Wird allerdings kein Sonderfall abgefangen, werden jeweils 8 Bytes aus dem alten Array an Index `r2` auf die SIMD-Register `d8`, `d9` und `d10` zwischengespeichert und von dort auf `r1` übertragen. Die Wertinterpretationen der einzelnen Register bleiben dabei größtenteils dieselben. `r3`, vormals die zuzuschneidende Breite, wird mit 3 multipliziert und `r7`, in SISD unbelegt, wird zum Pointer der Beginn der Zeile, an der geschrieben wird. Die Schleifenabbruchskonditionen und das Ende des Programms sind analog zur SISD-Version.

4.3 Implementierung von `zoom(...)`

Ebenso wie `window(...)`, ist `char* zoom(char* data, int width, int height, int factor)` in drei verschiedenen Versionen implementiert, die beiden Assemblerroutinen werden in diesem Unterpunkt dokumentiert, die Version in C-Code ist analog zur SISD-Version.
Die Register `r4 - r11` werden auf den Stack gepusht. Aus Lesbarkeitsgründen wurden die Eingabeparameter `int width`, `int height` und `int factor` von den Registern `r1 - r3` um jeweils ein Register verschoben, da in `window(...)` das neu allozierte Array auf `r1` lag und das Projektteam es einfacher fand, an diesem Punkt nicht umzudenken. Via `malloc` wird daraufhin der dynamisch errechnete Speicherplatz des neuen Arrays festegelegt, analog zu `window(...)`; die Berechnung ist aufgrund anderer Anforderungen verschieden (Formel: `width * height * (scale*scale) * 3`). Der restliche Algorithmus lässt sich in zwei Teilalgorithmen aufteilen: Die Skalierung und das Befüllen.
Die SISD-Skalierung weicht in ihrer Implementierung von der SIMD-Version nicht ab. Nach der Allozierung werden die Register `r5 - r12` mit Werten belegt, die für die Skalierung benötigt werden. Die Werteinterpretation ist nach allen Berechnungen folgende:

```
r0 = data (originale Bilddaten)
r1 = neues, alloziertes Array
r2 = width (Breite der originalen Bilddaten)
r3 = height (Höhe der originalen Bilddaten)
r4 = factor (Skalierungsfaktor)
r5 = Pointer auf data
r6 = Pointer auf das neue Array
r7 = Inkrement zwischen den Pixeln
r8 = Breiten Counter
r9 = Zwischenspeicher für RGB-Werte
r10 = Kopie von r0
r11 = Letzter Pointer vom neuen Array
r12 = Nebenrechnungen / Inkrement zwischen den Zeilen
```

Eine Schleife iteriert über die RGB-Werte der einzelnen Pixel des Arrays, lädt diese auf

r9 und speichert sie via Pointer r6 im neuen Array ab. Die Pointer r5 und r6 werden pro Iteration inkrementiert; r5 durch Post-Inkrement während des Ladens aus dem Array, r6 durch Addition nach den Lade- und Speicheroperationen. Die Abbruchskondition der Schleife ist das erreichen von r5 des letzten Pointers auf dem neuen Array.
Beim Befüllen unterscheiden sich die Implementierungen von SISD und SIMD. Es wird zunächst die SISD-Version beschrieben. Deren Werteinterpretation seien folgende (ausgenommen der, die nicht überschieben wurden):

```
r0 = Endresultat des Algorithmus (Rückgabewert)
r2 = width * Skalierungsfaktor (neue Breite)
r3 = heigth * Skalierungsfaktor (neue Höhe)
r5 = Z"ahler der fillLoopX
r6 = Z"ahler der fillLoopY
r7 = x-Koordinate des Nearest Neighbor
r8 = y-Koordinate des Nearest Neighbor
r9 = Index des bearbeiteten Pixel
r10 = Zwischenspeicher für RGB-Wert
r11 = Index des Nearest Neighbor
```

Die Koordinaten des *Nearest Neighbor*, also dem Pixel, welcher als Befüllungsvorlage dient, werden durch die Formel: `(x + factor/2)/factor * factor` (ganzzahlige Division) berechnet bzw. analog für die y-Koordinate. Durch diese Koordinaten lässt sich der NN-Index auf dem Array berechnen, welcher in r11 abgespeichert wird. So werden die RGB-Werte des Index r11 an die Stelle am Index r9 geladen. Ist r9 beispielsweise sein eigener *Nearest Neighbor*, so wird sein eigener Farbwert an dieselbe Stelle gespeichert. Die Abbruchskonditionen der beiden Schleifen sind das erreichen der Werte der neuen Bildbreite bzw. Bildhöhe des jeweiligen Zählers der Schleife. Das neue Array wird auf r0 geschrieben, damit dieses zurückgegeben werden kann. Das Programm endet mit dem vom Stack laden der caller-save-Register.
Da das eine die Grundlage des anderen ist, ist die SIMD der SISD Version sehr ähnlich, aufgrund ihrer Parallelisierung allerdings schneller in ihrer Ausführung. SIMD macht sich die q-Register und deren Adressierbarkeit durch die s-Register zu nutze. Aus Verständnisgründen seien hier auch die Wertinterpretation der genutzten NEON-Register aufgelistet (die der ARM-Register weichen kaum ab):

```
q1 = x-Koordinate der nächsten 4 Nearest Neighbor
q2 = y-Koordinate des nächsten 4 Nearest Neighbor
q3 = factor (Skalierungsfaktor)
q4 = Index vom Nearest Neighbor
q5 = Index vom bearbeiteten Pixel
q6 = x-Koordinate des bearbeiteten Pixels
q7 = y-Koordinate des bearbeiteten Pixels
q8 = width * Skalierungsfaktor
q9 = height * Skalierungsfaktor
q10 = Nebenrechnungen
```

```
q13 = Out-of-Boundsüberprüfungsergebnis
q14 = Randüberprüfungsergebnis
```

Vor Schleifeniteration, werden die Register `q3`, `q8`, `q9` und `q10` mit den Werten, die für die Berechnungen benötigt werden, gefüllt. In den Schleifen wird zuerst in der inneren `fillLoopX` das `q1`-Register gefüllt. Ein `q`-Register fasst 16 Bytes, weshalb ein `s`-Register, welches 4 Byte lang ist, 4-mal in ein `q` passt. Folglich, schreiben `s4`, `s5`, `s6` und `s7` das gesamte `q1`-Register byteweise. In `q1` stehen ergo die Indizes der x-Koordinate der *Nearest Neighbor* vier konsekutiver Pixel. Da die erste Schleife `fillLoopY` über die y-Achse iteriert, haben die in `q1` geschriebenen Pixel alle dieselbe y-Koordinate, weshalb das errechnen von `q2`, der entsprechenden y-Koordinate, in einem Schritt abgehandelt werden kann. Es folgt ein Check über die Arraygrenzen, dessen Ergebnis in `q13` geschrieben wird. Dieser ist wichtig, um Segmentationsfehler am Ende des Algorithmus zu vermeiden, da immer 8 Bytes geschrieben werden es möglich ist, dass für die letzten paar Pixel die Arraygröße überschritten wird, würden tatsächlich 8 Bytes geschrieben werden. Auch in `q14` wird das Ergebnis einer Überprüfung geschrieben: ein Check auf Randpixel, deren *Nearest Neighbor* eigentlich außerhalb des Bildbereichs liegen würde und dessen Koordinate deswegen verschoben werden müsste. Ist dies der Fall, so werden `q1` und `q2` angepasst und verschoben. Daraufhin wird der Index vom *Nearest Neighbor* kalkuliert und der Pointer auf `q5` geschrieben, selbiges bei `q4` für den Index, an den die Pixel aus `q5` geschrieben werden sollen.
Für das Laden und Speichern der Pixel im Array wird innerhalb der beiden Schleifen eine weitere Schleife benutzt, `SISD_LS`, welche unparallelisiert ist und mit ARM-Registern arbeitet, aus welchem Grund die in `q5` liegenden konsekutiven Pixel einzeln abgehandelt werden. `s16` adressiert den ersten Byte innerhalb von `q4`, analog dazu `s20` auf `q5`. So bekommen `r7` und `r8` jeweils vor Iteration von `SISD_LS` den benötigten Index übergeben. In der Schleife selbst wird auf sowohl `r7` als auch `r8` der Pointer in `r1` addiert, um auch aus den beiden Registern den richtigen Pointer zu machen. Es folgen die bereits aus den vorher dokumentierten Methoden die bekannten Lade- und Speicheroperationen. Durch diese Schleife wird 4 mal iteriert, um die vier Pixel aus `q5` einzeln zu schreiben. `r12` ist Zähler der Schleife. Mit jeder Iterierung wird diser inkrementiert und erlaubt eine Fallunterscheidung für Sprünge in verschiedene Labels innerhalb der Schleife. `r12` kann drei verschiedene Stadien annehmen: kleiner als 2, genau zwei, größer als 2, welche das Programm auf `load1`, `load2` und `load3` springen lassen. Damit wurde versucht ein `switch`-Statement zu emulieren. In `load3`, in welches gesprungen wird, sollte `r12` den Wert 3 oder 4 haben, steckt auch die Abbruchskondition: `SISD_LS` wird verlassen, sollte `r12` den Wert 4 haben, sprich die vierte Iteration über die Schleife abgeschlossen worden sein. Am Ende der `fillLoopY`- und `fillLoopX`-Schleifen werden nur die Zähler `r5` und `r6` gepüft und gegebenenfalls inkrementiert. Gen Ende des Programms wird das neue Array auf `r0` als Rückgabewert geschrieben und die caller-save-Register vom Stack gepusht.

4.4 Debug-Funktion

Um sich die Arbeit einfacher zu machen, hat das Projektteam im Rahmenprogramm eine Debug-Funktion implementiert, um mögliche Fehler innerhalb des Resultatbildes besser abfangen zu können. Da diese Funktion weder Teil der Aufgabe ist, noch die Funktionalität des Codes beeinflusst, aber dennoch Teil der Implementierung ist, sei diese hier nur kurz erwähnt. Bei weiteren Unklarheiten kann das Command Line Argument `-help` zu rate gezogen werden.
Die Debug-Funktion wird nur dann aktiviert, wenn beim starten des Programms der Command `-debug` eingelesen wird. Ferner kann man durch weitere Spezifikationen einschränken, welche Debug-Funktion genau ausgeführt wird. `-help` dokumentiert die einzelnen Parameter dieser Funktion.
Ist die Debug-Funktion aktiv, so wird im Kernel regelmäßig ausgegeben, welche einzelnen Schritte im Code bearbeitet wurden, um nachzuvollziehen, an welcher Stelle sich Fehler eingeschlichen haben könnten und, wo genau das Programm abstürzen könnte. Diese Ausgaben finden sich sowohl in der `main(...)`-Methode, als auch in `readBMP(...)` und `writeBMP(...)`. Lediglich die verschiedenen `window(...)` und `zoom(...)` Implementierungen sind frei von diesen Einträgen.
Es ist durch `-debug window` möglich das Programm vorzeitig abzubrechen und die `zoom(...)`-Methode nicht auszuführen, sodass man nur das zugeschnittene Bild als neue Datei sichert, allerdings beeinflusst die gesamte Debug-Funktion sonst nicht den Restalgorithmus und, da die Laufzeit der einzelnen `if`-Abfragen und `printf(...)`-Statement in $\mathcal{O}(1)$ liegt, kaum bis vernachlässigbar die Laufzeit.

5 Ergebnisse

5.1 Abweichungen von der Aufgabenstellung

Bei der bei der Signatur der Funktion `zoom(...)`, war es notwendig von der Aufgabenstellung abzuweichen, da diese weder einen Rückgabewert spezifiziert, noch einen Pointer auf das neue Array mitliefert. Bei der Funktion `window(...)` war dies nicht notwendig, da das zurückgegebene Bild nie größer ist als das Eingabebild. Das Ausgabebild hätte daher in das Eingabearray geschrieben werden können. Aus Konsistenzgründen, allerdings, geben nun beide Funktionen (in allen ihren Varianten) den Pointer auf das neue Bild zurück.
Des Weiteren wird in der vorgegebenen Signatur der `window(...)`-Funktion bezüglich der Dimensionen der Ein- und Ausgabe nur die Höhe und Breite des Ausgabebildes mitgeliefert. Da in den Arrays die Bilder Zeile für Zeile gespeichert sind, wird die Breite des Eingabebildes gebraucht. Diese wird verwendet, um zu berechnen, wie viele Pixel aus dem originalen Bild beim Sprung auf die nächste Zeile übersprungen werden müssen.

5.2 Ausgabeanalyse

Aufgrund der Aufgabenstellung hat das Ausgabebild keinen uniformen Rand. Horizontal und vertikal ist dieser am Bildanfang dünner und am Bildende dicker. Wie Abbildung 1 entnommen werden kann, werden die Pixel bei der Skalierung anfangs relativ zu ihrer ursprünglichen Position links unten ins Bild geschrieben. Daraufhin wird standardmäßig für jeden Pixel der *Nearest Neighbor* gesucht. Dies bringt eine ungleichmäßige Skalierung der einzelnen Pixel mit sich.

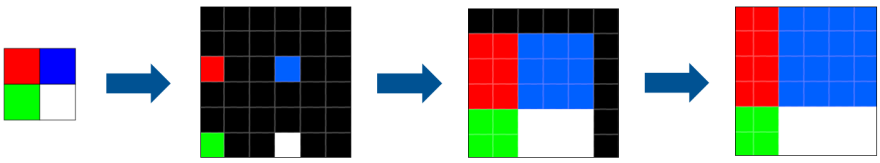

Abbildung 1: Skalierung laut Aufgabenstellung

5.3 Leistung und Effizienz

Die Performanz der Implementierungen wurde sowohl auf den von der Universität bereitgestellten Raspberry Pis, als auch auf privaten des Teams, getestet. Durch Parallelisierung mithilfe der NEON Operationen, schwanken die gemessenen Werte sehr stark. Die Unterschiede in der Messung betragen bis zu 100% bei gleicher Eingabe. Folgende Testergebnisse stellen den Durchschnitt aus zehn Ausfuehrungen dar. Dies soll genannten Abweichungen entgegenwirken, löst das Problem jedoch noch nicht.
Des Weiteren ist anzumerken, dass GCC mehrere Optimierungsstufen anbietet, unter anderem `-O2` und `-O3`. Zusätzlich zu den Optimierungen aus ersterer Stufe, versucht letztere außerdem Code zu parallelisieren. Für folgende Werte wurde einem Bild ein 200 * 300 Pixel großer Ausschnitt entnommen und mit einem Faktor von 5 skaliert.

```
Window:
    C (O2):      0.001611 s
    C (O3):      0.001045 s
    SISD:        0.001692 s
    SIMD:        0.001044 s

Zoom:
    C (O2):      0.088437 s
    C (O3):      0.086181 s
    SISD:        0.101470 s
    SIMD:        0.068441 s
    SIMD Alt.:   0.067203 s
```

Wie hier zu sehen ist, ist im Falle der `window(...)` Funktion der Unterschied zwischen `-O2` und `-O3` sehr ähnlich dem, zwischen der SISD- und SIMD-Version. Die Parallelisierung macht hier einen deutlichen Unterschied, da mithilfe `VLDn` und `VSTn` simultan acht

Pixel geschrieben werden. Da die restlichen Berechnungen einen großen Teil der Funktion ausmachen, und Parallelisierung mit einem gewissen Mehraufwand einhergeht, ist das Programm hiermit nur ca 35% schneller.
Bei der `zoom(...)` Funktion sieht dies dagegen anders aus, da GCC es scheinbar nicht geschafft hat, den Code mit `-O3` merkbar zu beschleunigen. Folglich ist die SIMD-Version (und ihre Alternative, welche im folgenden Kapitel erklärt wird,) hier sichtbar schneller. Außerdem scheint die SISD-Version, aufgrund der Optimierungen aus `-O2` bei der C-Variante, deutlich langsamer zu sein. Da alle Befehle für parallelisiertes Laden und Speichern, verlangen, dass die Anzahl der Bytes eine Potenz von zwei ist, jeder Pixel aber aus drei Bytes besteht, ist es nicht gelungen dies hier zu parallelisieren. Der Leistungsunterschied ist hier im Vergleich zur C-Version ca 20%.

5.4 Lösungsalternativen und Verbesserungsvorschläge

Aufgrund der im Abschnitt 5.2 beschriebenen Problematik bezüglich der Randfälle hat sich das Team eine alternative Version überlegt. In dieser wird die Verschiebung um `+(factor / 2)` ausgelassen. Daher ist es nicht mehr notwendig, Randfälle abzufangen, die keinen Segmentierungsfehler mit sich ziehen würden. Dies beschleunigt das Programm bei der Ausführung. Dieser Unterschied liegt allerdings im Messfehler-Bereich, wie aus Abschnitt 5.3 entnehmbar ist.
Was diese Änderung ebenfalls mit sich zieht, ist dass die Ränder des Ausgabebildes von uniformer Breite sind. (Siehe Abbildung 2) Das Endresultat ist daher sowohl ein schnelleres Programm, als auch ein optisch besseres Ergebnisbild.

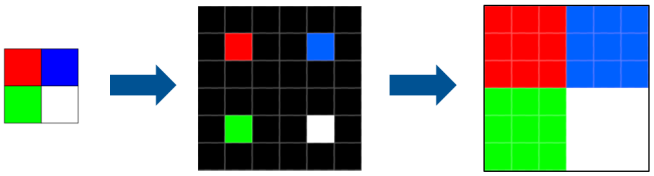

Abbildung 2: Alternative Skalierung mit uniformen Rändern

6 Zusammenfassung und Ausblick

Das Projektteam ist sich einig, dass es mit den gegebenen Vorgaben gute Arbeit geleistet hat. Das *Nearest Neighbor*-Verfahren ist ein primitiver Algorithmus und die Qualität des Resultats wird zwangsläufig darunter leiden, dass keine höherwertige Methode zur Skalierung verwendet wurde. Aufgrund mangelnder Interpolation von Farbwerten und das einfache kopieren der bereits vorhandenen, wird das Bild, welches der Algorithmus ausgibt, sehr blockig und unscharf aussehen, die mangelnde Qualität des Ausgabebildes ist also inhärent zum gewählten Verfahren.