Medianfilter

Spezifikation Assembler

Praktikum: Einführung in die Rechnerarchitektur
Sommersemester 2018
Gruppe 33

Autoren	Ruben Bachmann
	Franziska Steinle
	Roland Würsching

Eingereicht am	13.05.2018

Inhaltsverzeichnis

1. Aufgabenverteilung

Projektleiter: Ruben Bachmann

Dokumentation: Franziska Steinle

Abschlussvortrag: Roland Würsching

2. Aufgabenbeschreibung

Ziel dieser Aufgabe ist die Entwicklung eines Medianfilters, der aus einem verrauschten Signal ein gefiltertes Signal erstellen kann. Im realen Kontext sollte dieses gefilterte Signal in etwa dem unverrauschten Signal entsprechen. Das verrauschte Signal wird durch eine Folge vorzeichenbehafteter 32-Bit Ganzzahlen simuliert, die an einer zuvor festgelegten Stelle im Speicher stehen. Der Filter soll dabei folgendermaßen funktionieren:
Zu Beginn soll die Folge von Zahlen an eine Zwischenspeicherstelle kopiert werden, damit diese unverändert bleibt. Im Anschluss daran soll die kopierte Folge in mehrere kleine Blöcke gleicher Größe unterteilt werden. Die Blockgröße muss dabei ungerade sein, damit jeder Block einen Median („mittleres Element“) hat.
Im Anschluss daran sollen die Werte innerhalb der Blöcke mit einem geeigneten Verfahren der Größe nach aufsteigend sortiert werden. Hierbei ist es wichtig ein Verfahren zu wählen, das sich sowohl gut in Assembler implementieren lässt, als auch eine angemessene Effizienz hat.
Nachdem die Werte innerhalb der Blöcke sortiert wurden, soll aus jedem Block das mittlere Element ausgewählt werden und entsprechend der Reihenfolge der Blöcke innerhalb der ursprünglichen Folge ab einer festgelegten Zielspeicherstelle abgelegt werden. Im Kontext der Signalverarbeitung entspricht diese Medianfolge dem gefilterten Signal.

Die Filterfunktion soll in x86-Assembler implementiert werden. Außerdem soll ein Rahmenprogramm in C erstellt werden, mit dem die Funktion aufgerufen werden kann. Die Signatur der Funktion ist vorgegeben als:

*void filter5(int N, int *start, int length, int *dest)*

Es soll außerdem explizit die Frage beantwortet werden, aus wie vielen Einzelwerten das gefilterte Signal am Ende besteht.
Für diese Aufgabe kann davon ausgegangen werden, dass ausreichend Speicher zur Verfügung steht.

2.1. Ist-Analyse

Ab der Speicheradresse `start` liegt eine Folge von vorzeichenbehafteten 32-Bit Ganzzahlen, die im realen Kontext einem verrauschten Signal entspricht. Dieses Signal kann z.B. im Rahmenprogramm generiert werden. Dann wird die Funktion *filter5* aufgerufen.
Als Funktionsparameter liegen vor:

- int `N`: Blockgröße
- int `*start`: Startspeicherstelle
- int `length`: Länge der Zahlenfolge
- int `*dest`: Zielspeicherstelle

2.2. Soll-Analyse

Auf die Zahlenfolge ab der Adresse `start` mit der Länge int `length` wird ein Medianfilter angewendet. Dieser funktioniert wie folgt:

1. Die Folge wird in Blöcke gleicher (ungerader) Größe unterteilt.
2. Jeder Block wird mit einem geeigneten Sortierverfahren der Größe nach aufsteigend sortiert.
3. Der Median (mittlerer Wert) jedes Blocks wird an die Zieladresse `dest` geschrieben.

Die ab der Zielspeicherstelle abgespeicherte Medianfolge, bestehend aus $\frac{\texttt{length}}{N}$ (abgerundet) Einzelwerten, entspricht dann dem gefilterten Signal.

2.3. Werkzeug und Hilfsmittel

- 64-Bit Ubuntu 16.04
- nasm (Implementierung der Filterfunktion)
- gcc (Implementierung des C-Rahmenprogramms, Testen)
- Jasmin (Emulieren)

2.4. Bemerkungen

Für den Nachweis der erfolgreichen Implementierung wird eine graphische Darstellung des ursprünglichen sowie des gefilterten Signals zur Verfügung gestellt. Diese Graphen können dann zum Vergleichen übereinander gelegt werden. Des Weiteren werden ein Rahmenprogramm in C, eine Readme-Datei im .md-Format mit einer Bedienungsanleitung, ein Makefile, sowie einige Tests geschrieben, um die korrekte Funktionsweise zu zeigen und zu erklären.

3. Lösungsansätze

Beide Lösungsansätze bestehen aus den folgenden vier Funktionen. Alle Funktionen sichern die verwendeten Register auf dem Stack (*push*) und stellen sie am Funktionsende wieder her (*pop*).

Kopierfunktion: Diese Funktion kopiert die zu bearbeitende Zahlenfolge an die Zielspeicherstelle. Dies ist notwendig, um die ursprüngliche Zahlenfolge nicht zu verändern, und somit nach Ausführung des Programms die neu generierte Zahlenfolge noch mit der ursprünglichen vergleichen zu können.
Die Funktion benötigt drei Register mit den Werten `start`, `length` und `dest`. In einer Schleife wird das aktuelle Doppelwort (4 Byte) vom Startbereich in den Zielbereich geschrieben. Dann werden die Start- und Zielregister jeweils um 4 inkrementiert, und die Länge um 1 dekrementiert. Die Schleife terminiert, sobald die Länge 0 erreicht.

Sortierfunktion: Diese Funktion sortiert die Zahlen des aktuellen Blocks aufteigend. Dabei ist zu beachten, dass beim Vergleichen der Werte die Zahlen als vorzeichenbehaftet behandelt werden und die entsprechenden Befehle (*JL*, *JG*) gewählt werden. Das verwendete Sortierverfahren unterscheidet sich bei den zwei Ansätzen. Bei beiden Verfahren werden der Funktion die Startadresse des zu sortierenden Blocks und die Blocklänge in Registern übergeben.

Swap-Funktion Diese Funktion bekommt als Eingabe die Adressen zweier Werte (jeweils 4 Byte), die vertauscht werden sollen. Sie kommt in der Sortierfunktion zum Einsatz. Dafür werden zwei Hilfsregister benötigt, um die zu vertauschenden Werte zu speichern. Dies ist nötig, da der *MOV*-Befehl nur einen Speicheroperanden nehmen kann. Die gespeicherten Werte werden dann an die jeweils andere Adresse geschrieben.

Hauptfunktion Disee Funktion stellt den eigentlichen Programmablauf dar und verbindet die anderen drei Funktionen zu einem funktionierenden Medianfilter. Zunächst müssen die Eingabeparameter überprüft werden. Folgende Bedingungen müssen eingehalten werden:

- `N` ungerade (damit jeder Block einen Median hat)
- `length` ≥ 0 (sinnvolle Länge der Zahlenfolge)

Dann wird mit der Kopierfunktion das Signal an die Zielspeicherstelle kopiert, um die ursprünglichen Werte zu erhalten. Diese Kopie soll nun blockweise sortiert werden. Dazu wird auf jeden Block die Sortierfunktion aufgerufen, die wiederum von der Swap-Funktion Gebrauch macht. Wichtig für die Hauptfunktion ist es hierbei sicher zu stellen, dass all diesen Funktionen die richtigen Parameter übergeben werden.
Nach dem Sortieren muss von jedem Block der Median extrahiert werden. Die Position des

Medians innerhalb eines Blocks ist $m = \frac{\mathtt{N}-1}{2}$, d.h. auf die Startadresse des Blocks muss m addiert werden. Die Mediane werden sofort an die Zielspeicherstelle gelegt, d.h. die Signalkopie wird zum Teil überschrieben. Diese Medianfolge repräsentiert das gefilterte Signal.
Werte am Signalende, die nicht in einen ganzen Block passen, werden ignoriert.

3.1. Lösungsansatz A: Selection Sort

Die Adresse des ersten Blockelements wird in Register X und Y (Pseudobezeichnungen) geschrieben. In der inneren Schleife soll das kleinste Element rechts von X gefunden werden. Dazu wird der Restblock abgesucht, und falls der aktuelle Wert kleiner als der in Y ist, wird dessen Adresse in Y gespeichert. Am Ende des Blocks angelangt, wird mittels der Swap-Funktion das gefundene Minimum mit X vertauscht. Dann wird in der äußeren Schleife X inkrementiert, Y gleich X gesetzt und wieder das Minimum gesucht. Dies wird in der Regel bis zum Ende des Blocks fortgeführt, aber für unsere Zwecke kann man bei der Mitte des Blocks aufhören, da nur der Median gesucht wird.

3.2. Lösungsansatz B: Bubblesort

Der Algorithmus beginnt beim ersten Blockelement und geht anschließend den Rest des Blocks Element für Element ab. Beim Durchlaufen des Blocks werden jeweils zwei benachbarte Elemente verglichen. Ist das erste Element größer als das zweite, so werden die Inhalte der beiden Speicherzellen vertauscht. Ist das Verfahren am Ende des Blocks angelangt, so kehrt der Algorithmus zum Anfang zurück. Dies wird so lange in einer Schleife wiederholt, bis in einem Durchlauf kein Swap mehr ausgeführt wird, denn dann ist der Block komplett sortiert.

3.3. Bewertung der Ansätze

Beide Sortieralgorithmen lassen sich vergleichsweise einfach in Assembler implementieren, und sind auch von der Laufzeit her gesehen sehr ähnlich (worst-case: $O(n^2)$). Jedoch haben beide Verfahren einige Vor- und Nachteile: Der erste Lösungsansatz hat den großen Vorteil, dass die Swap-Funktion wesentlich seltener aufgerufen wird als im Zweiten Lösungsansatz. Dies liegt daran, dass sie nur einmal pro Blockdurchlauf verwendet wird, während beim zweiten Lösungsansatz im worst case nach jedem Vergleich die swap Operation aufgerufen wird. Da die Swap-Funktion sehr viele Speicherzugriffe macht, ist dies ein großer Vorteil. Darüber hinaus ist der erste Lösungsansatz etwas einfacher in Assembler zu implementieren, da die Schleifendurchläufe etwas übersichtlicher gestaltet werden können.
Der Vorteil des zweiten Lösungsansatzes ist es, dass nur ein Register benötigt, um den Swap durchzuführen, da die zu vertauschenden Werte immer nebeneinander liegen. Da viele der Funktionen recht viele Register für die Parameterübergabe benötigen, und somit häufig viele Register belegt sind, könnte sich die Ersparnis dieses einen Registers als nützlich erweisen.

Außerdem weist Bubblesort bei Arrays, die bis auf wenige Elemente fast sortiert sind, eine besonders gute Effizienz auf.
Unsere Gruppe hat sich dazu entschieden den ersten Lösungsansatz für die Implementierung zu verwenden, da der Vorteil einer etwas einfacheren Implementierung sowie einer besseren Laufzeit mit weniger Speicherzugriffen die Ersparnis eines Registers unserer Meinung nach überwiegen. Alles in allem sind jedoch beide Ansätze im Vergleich zu anderen Sortieralgorithmen gut für die Implementierung in Assembler geeignet, da andere Verfahren wie beispielsweise Mergesort oder Quicksort nur schwer Assembler implementierbar sind. Da die Blocklänge in der Regel nicht sehr groß gewählt wird (häufigerweise 5, 7 oder 9), erfüllt das gewählte Verfahren unsere Zwecke.

4. Zeitplanung

Termin	Aufgabe
18.04.	Erstes Treffen zur Klärung grundsätzlicher Fragen zur Aufgabenstellung. Festlegung der Rollen. Planung des weiteren Vorgehens. (ca. 1 Stunde)
18.04. - 25.04.	Selbstständiges informieren zu den wichtigsten Punkten der Aufgabenstellung. Gedanken zu möglichen Lösungsansätzen machen. (ca. 5 Stunden)
25.04.	Treffen zum Zusammentragen der in der vergangenen Woche selbstständig erarbeiteten Informationen. Beginn der Ausarbeitung der Spezifikation. (ca. 1,5 Stunden)
25.04. - 09.05.	Fertigstellung der Spezifikation. (ca. 10 – 15 Stunden)
09.05.	Treffen zum Besprechen letzter Unklarheiten bezüglich der Spezifikation. Planung der Implementierungsphase. (ca. 1,5 Stunden)
11.05. - 13.05.	Hochladen der finalen Version der Spezifikation in das SVN.
13.05. - 23.05.	Beginn der Implementierung. Hierbei übernimmt jedes Gruppenmitglied die Aufgaben, welche ihm beim Treffen am 09.05.2018 zugeteilt wurden. (ca. 20 – 30 Stunden)
23.05.	Treffen zum Austausch über bisherigen Verlauf der Implementierung. Diskussion von eventuell aufgetretenen Problemen. (ca. 1,5 Stunden)
23.05. - 06.06.	Fortsetzen der Implementierung. (ca. 20 – 30 Stunden)
06.06.	Treffen zum Austausch über bisherigen Verlauf der Implementierung. Ausarbeiten der Tests. (ca. 1,5 Stunden)
06.06. - 13.06.	Fertigstellung der Implementierung. Schreiben und testen der Tests. (ca. 10 Stunden)
13.06.	Treffen zum Besprechen letzter Unklarheiten bezüglich der Implementierung, sowie Planung der Ausarbeitungsphase. (ca. 1,5 Stunden)
15.06. - 17.06.	Hochladen der finalen Version der Implementierung in das SVN.
17.06. - 27.06.	Beginn der Ausarbeitung. Hierbei übernimmt jedes Gruppenmitglied die ihm/ihr zugeteilten Aufgaben. (ca. 10 – 15 Stunden)

27.06.	Treffen zum Austausch über bisherigen Verlauf der Ausarbeitung (schreiben der Dokumentationen). Diskussion von eventuell aufgetretenen Problemen. => in Kombination mit VHDL-Projekt (ca. 1,5 Stunden)
27.06. - 06.07.	Fertigstellung der Ausarbeitung. Überprüfung der Dokumentationen auf Verständlichkeit. Planung der Vortragsphase. (ca. 15 – 20 Stunden)
06.07. - 08.07.	Hochladen der finalen Version der Dokumentationen bzw. des Lösungsansatzes in das SVN.
08.07. - Vortrag	Gruppenmitglieder bereiten ihren jeweiligen Teil der Präsentation vor und halten sich gegenseitig auf dem Laufenden (ca. 20 – 30 Stunden)