LEHRSTUHL FÜR RECHNERTECHNIK UND RECHNERORGANISATION

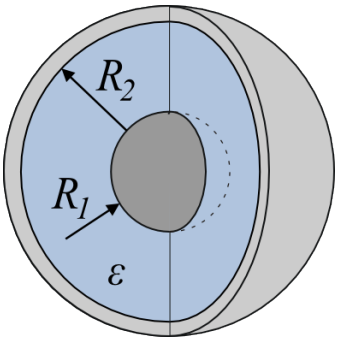
Aspekte der systemnahen Programmierung bei der Spieleentwicklung

Projektaufgabe 1.77: Kugelkondensator mit verschiedenen Dielektrika

Gruppe π:
Moritz Mackiewicz
Elias Marquart
Claus Strasburger

1 Problemstellung und Spezifikation

1.1 Aufgabenstellung



In der von uns bearbeiteten Aufgabe geht es um die Berechnung der Kapazitäten von mehreren Kugelkondensatoren.
Ein Kugelkondensator besteht aus zwei konzentrisch angeordneten Kugeln, die durch ein Isoliermaterial, genannt *Dielektrikum*, getrennt werden. Die Kapazität des resultierenden Kondensators hängt von dem Verhältnis der Radien der beiden Kugeln und dem verwendeten Dielektrikum ab.
Dazu wird eine Liste von Radien — jeweils für den Radius der inneren und der äußeren Kugel — in Form einer Textdatei eingegeben. Zu dieser werden dann in einem Assemblerprogramm die jeweiligen Kapazitäten für verschiedene Dielektrika berechnet (in unserem Fall Hartgummi und Hartpapier). Die Abmessungen bestehen jeweils aus zwei Radien, dem der inneren Kugel r_1 und dem der äußeren Kugel r_2.
Die Formel zur Berechnung der Kapazität eines Kugelkondensators lautet:

$$C = 4\pi\varepsilon_0\varepsilon_r\frac{r_1 \cdot r_2}{r_2 - r_1}$$

ε_0 bezeichnet dabei die elektrische Feldkonstante und beträgt $8{,}85418781762 \cdot 10^{-12}$.
ε_r bezeichnet den Dielektrizitätswert des Materials (für Hartgummi $3{,}0$, für Hartpapier $5{,}0$).

1.2 Implementierungsumgebung

Wir verwenden zur Entwicklung das *BeagleBoard xM* , ein ARM Cortex-A8-System (aufbauend auf der ARMv7-Architektur, getaktet mit 1 GHz) mit NEON-VFP. Die NEON-Architektur erlaubt es, Berechnungen mit bis zu vier Fließkommazahlen (32 bit) gleichzeitig durchzuführen, wodurch eine hohe Performanz beim *Number Crunching* erreicht wird. Dieses Verfahren wird *SIMD* (von ARM selbst *Advanced*-SIMD) genannt. Beispiele für fließkommaintensive Berechnungen sind Audioverarbeitung, 3D-Visualisierungen oder Physik-Simulationen.
Weiterhin bietet das *BeagleBoard xM* 512-MB LPDDR RAM, einen High-speed USB 2.0 OTG-Port, 10/100 Ethernet, DVI-D und S-Video-Anschlüsse, Stereo-Audio out/in, einen High-capacity microSD Slot, JTAG und einen Kamera-Port, was enorme Vielfalt in den Erweiterungs- und Anwendungsmöglichkeiten bietet.

Unsere Entwicklungsumgebung besteht aus dem *ARM Development Studio 5 für Eclipse*, *GNU make* und *gcc 4.7*.
DS-5 ist eine auf der Entwicklungsplattform Eclipse aufgebaute Umgebung, die eine einfach zu benutzende graphische Oberfläche zum Entwickeln, Ausführen und Debuggen von Programmen für ARM-Systeme bereitstellt. Die Einrichtung ist jedoch etwas umständlich, da DS-5 nicht ganz fehlerfrei ist und keinen klaren Leitfaden zum Erstellen eines Projekts bietet. Das Programm selbst ist kostenpflichtig, sowie nur unter einer eingeschränkten Lizenz distributiert, was eigene Erweiterungen erschwert.
GCC ist eine der weitverbreitesten Compilersuiten, ist vollständig Open-Source und in der Lage, für alle bekannten Architekturen und Betriebssysteme C, C++ und Assemblercode zu Binärcode umzubauen, wobei zusätzlich auch umfangreiche und effektive Optimierungsoptionen vorhanden sind.

2 Lösungsalternativen

2.1 Vector Floating Point (VFP)

Vorstellung

In der von uns enwickelten Lösung verwenden wir die Vector Floating Point Erweiterung des ARM-Prozessors. Diese erlaubt es uns mit *single* (32 bit)- und *double* (64 bit)-precision Fließkommazahlen nach dem IEEE 754-Standard zu rechnen.

Vorteile

- beliebige Menge an zu verarbeitenden Daten
- lauffähig auch auf Architekturen ohne NEON

Nachteile

- eventuell langsamer, da der VFP-Koprozessor (bzw. der NEON-Befehlssatz) nicht voll genutzt wird (d.h. ohne SIMD) bzw. mehr Taktzyklen für die Berechnung benötigt werden (je nachdem, ob der schnellere *VFPv3*- oder nur der *VFPLite*-Koprozessor vorhanden ist)
- VFP-Koprozessor wird benötigt

2.2 SIMD (NEON-Coprozessor)

Vorstellung

In dieser alternativen Lösung wird der NEON Coprozessor von ARM verwendet, um von *SIMD* (Single Instruction, Multiple Data) zu profitieren. Das bedeutet, dass pro Instruktion nicht mit einer, sondern mit bis zu vier Zahlen gleichzeitig gerechnet werden kann. Unsere Tests haben gezeigt, dass die Berechnung dadurch mindestens um den Faktor 6 beschleunigt wird.

Vorteile

- Um Faktor 6-8 schneller als VFP ohne SIMD

Nachteile

- SIMD Befehle müssen jeweils zwei oder vier Fließkommazahlen verarbeiten. Falls die Eingabedatenmenge kein Vielfaches von vier ist, kann für den entsprechenden Rest nicht derselbe Code verwendet werden.
- Der NEON Befehlssatz verfügt über keinen Divisionsbefehl. Die einzige Möglichkeit ist die Approximation des Kehrbruchs mit *VRECPE* und anschließende Multiplikation. Die einmalige Annäherung ist nicht sehr genau, durch mehrmalige Iteration des Newton-Raphson-Verfahrens mittels *VRECPS* erhält man jedoch ähnliche Genauigkeit wie mit nicht-SIMD-Operationen.

2.3 Entscheidungsprozess

Um keine Entscheidung zugunsten einer Lösung treffen zu müssen, haben wir einfach alle Alternativen selbst implementiert, wobei bei der NEON-Variante nur der Berechnungsteil in Assembler umgesetzt wurde, die Schleifenstruktur zur Übergabe der Daten wurde dagegen in C geschrieben.

3 Dokumentation der Implementierung

3.1 Entwickler-Dokumentation

3.1.1 Rahmenprogramm

Das in C geschriebene Rahmenprogramm liest die Datensätze mit den Radien der Kugelkondensatoren aus einer Textdatei in den Speicher. Mit diesen Daten werden dann drei verschiedene Implementierungsvarianten ausgeführt (Assembler mit VFP, Assembler mit NEON und Vektorisierung und die C-Referenzimplementierung), deren Ergebnisse ausgegeben werden und die dafür benötigte Zeit (in Mikrosekunden) gemessen wird.

Zunächst werden über die Methode $datei\_lesen$ die in der Datei $ui.txt$ stehenden Datensätze eingelesen. Die einzulesende Datei muss dabei folgenden Richtlinien folgen:

Zeile 1: $n - 1$
Zeile 2: $[x_0]_{r_1}$ mm $[x_0]_{r_2}$ mm
$\vdots$
Zeile n: $[x_{n-1}]_{r_1}$ mm $[x_{n-1}]_{r_2}$ mm

Hierbei ist x (mit der Mächtigkeit $|x| = n$) die Menge aus den zu berechnenden Datensätzen bestehend aus den 2-Tupeln $(r_1, r_2) \in \mathbb{R}^2$, wobei r_1 den Radius der inneren Kugel und r_2 den Radius der äußeren Kugel bezeichnet.

Die Methode wird mit den benötigten Parametern aufgerufen und der Rückgabewert gesichert. Dieser spezifiziert einen eventuell aufgetretenen Fehler ($\neq 0$ bei einem Fehler). Wenn ein Fehler aufgetreten ist wird dies anschließend erkannt und das Programm mit diesem Fehlerwert beendet.
Die Parameter von $datei\_lesen$ lauten wie folgt:

1. Name der einzulesenden Datei
2. Pointer auf ein *float*-Array, in dem die inneren Radien gespeichert werden
3. Pointer auf ein *float*-Array, in dem die äußeren Radien gespeichert werden

4. Pointer auf einen *int*, hier wird die Länge des Datensatzes gespeichert

```
int erfolg = datei_lesen("ui.txt", &rad1, &rad2, &length);
if(erfolg != 0) {
    return erfolg;
}
```

Im folgenden wird auf die Arbeitsweise dieser Methode anhand von weiteren Auszügen aus dem Quelltext eingegangen.

```
FILE* handle = NULL;
```

Erzeugt eine Variable namens $handle$, in der ein Pointer auf ein $FILE$-Struct, definiert in der Headerdatei $stdio.h$ (Teil der Standard-C-Bibliotheken) gespeichert werden kann. Damit wird ein späterer Zugriff auf die einzulesende Datei ermöglicht. Zunächst wird die Variable aber auf $NULL$ gesetzt, weil noch keine Datei geöffnet wurde.

```
char line[MAX_LINE_LENGTH];
```

Erzeugt eine Variable namens $line$, in der ein Array der Größe *MAX_LINE_LENGTH* (als *1024* definiert) mit Elementen vom Typ $char$ gespeichert wird. Benötigt wird diese Variable zum Speichern des Inhalts der aktuellen Zeile in der einzulesenden Datei.

```
handle = fopen(filename, "r");
if(!handle) {
    fprintf(stderr,"FEHLER: Datei konnte nicht geoeffnet werden!\n");
    return 1;
}
```

Mit *fopen* wird versucht die Datei, die unter dem Pfad spezifiziert vom Parameter *filename* zu finden ist, im Lesemodus (deshalb das *"r"* als zweiter Parameter) zu öffnen und anschließend eine Zugriffsreferenz für die spätere Verwendung unter der vorher definierten Variable *handle* zu sichern. Schlägt dies fehl, d.h. *handle* hat den Wert *0*, wird eine entsprechende Ausgabe an den systemweiten Error-Stream gesendet und die Methode mit dem Rückgabewert *1* abgebrochen.

```
if(feof(handle) || fgets(line, MAX_LINE_LENGTH, handle)==NULL) {
    fprintf(stderr,"Fehler: Datei ist leer\n");
    return 1;
}
```

Hier wird überprüft, ob es sich bei der einzulesenden Datei um eine leere Datei handelt bzw. von der Datei überhaupt gelesen werden kann. Zunächst wird mittels *feof* sichergestellt, dass das Ende des Dateistreams noch nicht erreicht ist, um anschließend zu überprüfen ob die erste Zeile gelesen werden kann. Die Methode *fgets* erwartet hierbei einen Pointer auf ein *char*-Array (*line*), in das bei einem erfolgreichen Lesevorgang die nächsten *1024* (*MAX_LINE_LENGTH*) Zeichen des Streams *handle* geschrieben werden.

Schlägt einer der Tests fehl, wird wiederum eine entsprechende Ausgabe an den systemweiten Error-Stream gesendet und die Methode mit dem Rückgabewert *1* abgebrochen.

```
1 *length = atoi(line);
2 if(*length <= 0) {
3     fprintf(stderr, "Fehler: '%s' ist keine sinnvolle Zeilennummer\n", line);
4     return 1;
5 }
```

Um die Größe des Datensatzes herauszufinden, wird hier der Inhalt der ersten Zeile mittels *atoi* in einen *Integer* umgewandelt, der dann im vom Parameter *length* referenzierten Speicher gesichert wird. Um fehlerhafte Daten auszuschließen, wird danach überprüft, ob es sich eine sinnvolle Länge handelt (> 0). Ist dies nicht der Fall wird eine entsprechende Fehlermeldung ausgegeben und die Methode mit dem Rückgabewert *1* abgebrochen.

```
1 *rad1 = (float*)malloc(*length * sizeof(float));
2 *rad2 = (float*)malloc(*length * sizeof(float));
3 if(!rad1 || !rad2) {
4     fprintf(stderr,"Fehler: keinen Speicher bekommen\n");
5     return 1;
6 }
```

Zunächst wird für die inneren (*rad1*) und äußeren (*rad2*) Radien der Kugelkondensatoren des Datensatzes versucht, entsprechend großen Speicher zu allozieren (Länge des Datensatzes multipliziert mit der Größe der Daten selbst, die ja aus *float*s bestehen). Anschließend wird überprüft, ob dieser Vorgang erfolgreich ausgeführt werden konnte, wobei bei einem Fehler ähnliches passiert wie in den vorangehenden Absätzen.

Die nachfolgende Schleife versucht, die Datei zeilenweise einzulesen und die dabei gewonnenen Daten im passenden Speicherbereich zu sichern. Dabei wird jeweils ähnlich wie zu Beginn der Methode sichergestellt, dass weder das Ende der Datei erreicht noch eine leere Zeile eingelesen wurde.

```
1 if(sscanf(line, "%f mm %f mm", (*rad1+index), (*rad2+index)) != 2) {
2     fprintf(stderr, "Fehler in Z. %d: %s\n", index, line);
3     fprintf(stderr, "Ich bin mal weg\n");
4     return 1;
5 }
```

Die Daten in der aktuellen Zeile der Datei werden hier mittels *sscanf* extrahiert und an der entsprechenden Position in den Arrays gespeichert. Die Parameter dieser Funktion verhalten sich dabei gleich wie *sprintf*, nur die Funktionsweise ist umgekehrt (Werte werden eingelesen statt ausgegeben). Der Rückgabewert entspricht dabei den erfolgreich ausgelesenen Werten, wenn dieser also nicht *2* ist, was bedeuten würde, dass die aktuelle Zeile ein ungültiges Format hat, wird das Einlesen abgebrochen und eine entsprechende Fehlermeldung ausgegeben. Außerdem werden Gegenmaßnahmen durchgeführt, falls in der Datei zu viele (> *length*: restliche Daten werden ignoriert) oder zu wenige (< *length*:

length wird entsprechend geändert) Daten zu finden sind.

Nun sollte sichergestellt sein, dass gültige Daten eingelesen wurden, diese werden dann, nachdem genügend großer Speicher für die Ergebnisse alloziert wurde der Assemblerroutine übergeben. Vorher wird noch der aktuelle Timestamp gespeichert, damit eine Zeitmessung erfolgen kann. Nachdem die Berechnung erfolgt ist, werden Auszüge aus den Ergebnissen ausgegeben sowie die benötigte Zeit.

Abschließend wird die geöffnete Datei noch mittels *fclose* geschlossen und ein Fehler ausgegeben, falls sie nicht geschlossen werden konnte.

```
1 if(fclose(handle) != 0) {
2     fprintf(stderr, "Fehler: Konnte Eingabedatei nicht schließen.\n");
3 }
```

3.1.2 Assemblerroutine – VFP-Version

Der Einstiegspunkt der Assemblerroutine ist die Funktion *_calc*. Sie erwartet folgende Parameter:

1. 2 *float*-Arrays (*data1*/*data2*), in denen die Abmessungen der Kondensatoren erwartet werden
2. 2 *float*-Arrays (*result1*/*result2*), in denen die berechneten Kapazitäten für die beiden Dielektrika gespeichert werden
3. Ein *int* (*length*), der die Größe des Datensatzes angibt

Im folgenden wird auf die Arbeitsweise dieser Routine anhand von Auszügen aus dem Quellcode näher eingegangen.

```
1 _calc :
2     PUSH {r4-r10,r11}
```

Hier werden zunächst alle in der Routine verwendeten Register auf dem Stack gesichert, um sie später wiederherstellen zu können.

```
1     ADD r11,sp, #0x1c
```

Das Register *r11* wird mit einem neuen Stackpointer beschrieben, der für den benötigten Stackframe verwendet wird. Da vorher *7* Words ($7 \cdot 4 = 28$ Byte groß, ausgenommen *r11* selbst) auf den Stack gepusht wurden, wird der referenzierte Speicher um *28* ($1c_{16}$) Bytes erhöht.

```
1     SUB sp, sp, #20
```

Vom eigentlichen Stackpointer werden nun *20* Bytes (*5* Words) abgezogen, da dieser Platz später zum Sichern der Parameter benötigt wird.

```
1     STR r0, [r11,#-0x8]
2     STR r1, [r11,#-0xc]
3     STR r2, [r11,#-0x10]
4     STR r3, [r11,#-0x14]
5     PUSH {lr}
```

Die ersten *4* Parameter (*r0*: data1, *r1*: data2, *r2*: result1, *r3*: result2) werden auf dem Stack gesichert, jeder Kopiervorgang wird dabei mit einem entsprechenden Offset ausgeführt (jeweils um die Größe eines Words vermindert, beim ersten Vorgang ein Word mehr, da sonst der vorher gesicherte Wert von *r11* überschrieben werden würde). Zusätzlich wird das Link-Register (*lr*) auf dem Stack platziert.

```
1     LDR r0, [r11,#4]
2     STR r0, [r11,#-0x18]
```

Der *5*. Parameter (*length*) wird gemäß Calling-Convention nicht direkt in einem Register übergeben, sondern befindet sich auf dem Stack. Von dort wird er zunächst in das Register *r0* geladen, um ihn anschließend an der von uns benötigten Position im Stack zu sichern.

```
1     LDR r1, [r11,#-0x8]
2     LDR r2, [r11,#-0xc]
3     LDR r3, [r11,#-0x10]
4     LDR r4, [r11,#-0x14]
```

Hier werden die restlichen Parameter wieder in die benötigten Register geschrieben (*r1*: data1, *r2*: data2, *r3*: result1, *r4*: result2).

```
1     VLDR.F32 s0, _M_PI
2     VLDR.F32 s3, _M_E_0
3     VMUL.F32 s0, s0, s3
4     VLDR.F32 s3, _M_4
5     VMUL.F32 s3, s0, s3
```

Nun kann die eigentliche Berechnung beginnen: Der konstante Faktor der Formel ($4 \cdot \pi \cdot \varepsilon_0$) wird vorberechnet. Dazu werden zunächst der Wert von π und ε_0 in den Single-Precision-Registern *s0* und *s3* gespeichert, damit mit diesen Werten Fließkomma-Operationen durchgeführt werden können. Anschließend werden diese beiden Zahlen multipliziert und auf ähnliche Weise mit dem Faktor 4 verrechnet. Das Ergebnis (also der konstante Teil der Formel) steht nun im Register *s3*. Ausgeführt werden all diese (und auch spätere) Befehle mit *32*bit-Fließkommazahlen.

```
1     SUB r0, r0, #1
2 loop:
3     ADD r5, r1, r0, LSL #2
4     ADD r6, r2, r0, LSL #2
5     ADD r7, r3, r0, LSL #2
6     ADD r8, r4, r0, LSL #2
```

Um die richtige Anzahl an Schleifendurchläufen auszuführen, muss von der Größe des Datensatzes (Parameter *length*, in *r0* gespeichert) noch *1* subtrahiert werden, da Arrays ja bekanntlich *0*-basiert sind. Anschließend werden mit einem Label (*loop*) das Beginn der Schleife markiert und die Pointer (für die Eingabe/Ausgabe) an die richtige Position gesetzt. Dafür wird die Zählvariable (*r0*) zunächst mittels *LSL #2* (Logical Shift Left, entspricht Multiplikation mit 2^n) über den Barrel-Shifter mit 4 multipliziert, um gültige Speicheroffsets zu bekommen (ein *float* ist *4* Byte groß).

```
1     VLDR.F32 s0, [r5]
2     VLDR.F32 s1, [r6]
3     VLDR.F32 s2, _E_GUMMI
4     BL _capacity
5     VSTR.F32 s0, [r7]
```

Zunächst werden die an den vorher berechneten Speicheradressen stehenden Werte (also der innere/äußere Radius des Kugelkondensators) in die für die Routine *_capacity* (führt eigentliche Berechnung der Kapazität aus) benötigten Register kopiert und der Dielektrizitätswert für Hartgummi in das richtige Register geladen. Dann wird mittels *BL* ein Sprung zu dieser Routine durchgeführt, wobei die Rücksprungadresse implizit im Link-Register gesichert wird. Ist die Routine fertig, wird das Ergebnis, dass per Konvention in Register *s0* gespeichert wurde, an die richtige Stelle im Ergebnis-Array geschrieben. Danach wird dasselbe (mit anderen Speicheradressen) noch einmal für das Dielektrikum Hartpapier durchgeführt.

```
1     SUBS r0, r0, #1
2     BPL loop
```

Nach dem Dekrementieren der Zählvariable (mit Operationssuffix *S*, um entsprechende Flags zu aktualisieren), wird, falls die Subtraktion kein negatives Ergebnis verursacht hat (d.h. $r0 >= 0$, angezeigt über die *N*-Flag) wieder an den Schleifenanfang gesprungen und die Datenverarbeitung fortgesetzt.

```
1     POP {lr}
2     SUB sp, r11, #0x1c
3     POP {r4-r10,r11}
4     BX lr
```

Am Ende werden alle (vorher gesicherten) verwendeten Register wiederhergestellt, der Stackframe wiederhergestellt und ein Rücksprung zu der im Link-Register gespeicherten Adresse veranlasst.

Im folgenden wird die Routine *_capacity*, die die eigentliche Berechnung der Kapazität durchführt, näher beschrieben. Diese Funktion erwartet als Parameter 4 *float*-Werte (innerer/äußerer Radius, Dielektrizitätswert und konstanter Teil der Formel wie oben erläutert) und liefert das Ergebnis in Register *s0* zurück.

```
1 _capacity :
2     PUSH {r5-r7,r11}
3     ADD r11,sp, #0xc
4     VPUSH {s3-s5}
```

Ähnlich wie vorangehend beschrieben werden hier die verwendeten Register auf dem Stack gesichert und ein neuer Stackframe erstellt. *VPUSH* sichert hierbei auch die verwendeten VFP-Register.

```
1     VSUB.F32 s4, s1, s0
2     VDIV.F32 s4, s0, s4
3     VMUL.F32 s4, s4, s1
4     VMUL.F32 s0, s3, s2
5     VMUL.F32 s0, s0, s4
```

Dieser Teil führt die Berechnung der Kapazität aus: Zunächst wird der innere Radius vom äußeren subtrahiert, anschließend wird mit dem Produkt aus den Radien der Quotient gebildet. Am Schluss wird das noch mit dem konstanten Teil der Formel und dem Dielektrizitätswert multipliziert, das Ergebnis liegt nun im Register *s0*.

```
1     VPOP {s3-s5}
2     SUB sp, r11, #0xc
3     POP {r5-r7,r11}
4     BX lr
```

Nun werden die verwendeten (und vorher gesicherten) Register wiederhergestellt, der Stackframe zurückgesetzt und mittels *BX* ein Rücksprung zum Aufrufer (bzw. an die im Link-Register gespeicherte Adresse) veranlasst.

3.1.3 Assemblerroutine – NEON-Version

Da wir uns zunächst für eine Implementierung ohne SIMD-Vektorisierung entschieden haben, haben wir den Assembler-Aufrufcode nicht an die Vektorisierung angepasst und haben die NEON-Alternative direkt mittels C-Code angesprochen. Der C-Code, der zum Aufruf der Funktion benötigt wird, wird hier nicht näher erläutert, da er zum Verständnis keinerlei Erklärung bedarf.
Die Routine *_fast_capacity* berechnet für 4 Kugelkondensatoren die Kapazität mit jeweils 2 Dielektrika. Sie erwartet — wie *_calc* — als Parameter 2 *float*-Arrays mit Mindestgröße 4, hieraus werden die Radien der Kondensatoren gelesen; ein *2* Elemente großes *float*-Array, in dem die Dielektrizitätswerte stehen und *2 float*-Arrays, in denen die Ergebnisse gespeichert werden.

```
1 _fast_capacity :
2     PUSH {r11}
3     ADD r11,sp,#0x0
4     VPUSH {q0-q5}
5     VLDM r2, {d0}
6     VLDM r0, {q1}
7     VLDM r1, {q2}
```

Zu Beginn werden sämtliche verwendeten Register sowie der Stackpointer gesichert und die übergebenen Parameter in die NEON-Register geladen. *VLDM* lädt dabei mehrere Werte in ein entsprechendes Vektorregister (*Q*uad-Register: *4* Werte, *D*ouble-Register: *2* Werte). Die inneren/äußeren Radien befinden sich nun also in den Registern *q1*/*q2*, die

beiden Dielektrizitätswerte im Register *d0*.

```
1    VSUB.F32 q4, q2, q1
2    VRECPE.F32 q3, q4
```

Danach wird mit der eigentlichen Berechnung begonnen, die im folgenden immer mit 4 Werten gleichzeitig operiert. Zunächst wird die Differenz aus den inneren und äußeren Radien gebildet, um dieses Ergebnis anschließend im ersten Schritt der Kehrbruchapproximation zu verwenden, da NEON direkte Division nicht unterstützt. *VRECPE* führt dabei eine bisweilen etwas ungenaue Approximation von $\frac{1}{d}$ (in unserem Fall also $\frac{1}{r2-r1}$) aus, die danach noch weiter präzisiert wird.

```
1    VRECPS.F32 q5, q3, q4
2    VMUL.F32 q3, q5, q3
3    VRECPS.F32 q5, q3, q4
4    VMUL.F32 q3, q5, q3
```

Um diesen Kehrbruch ($\frac{1}{d}$) genauer anzunähern, werden hier 2 Schritte der Newton-Raphson-Iteration ($x_{n+1} = x_n \cdot (2 - x_n \cdot d)$) durchgeführt (als Startwert wird die vorher berechnete Approximation in Register *q3* verwendet), die im NEON-Befehlssatz schon fest eingebaut ist. *VRECPS* errechnet den zweiten Faktor der Formel ($2 - x_n \cdot d$), anschließend muss noch mit x_n multipliziert werden. Unsere Tests haben ergeben, dass eine zweimalige Durchführung dieser Iteration genügende Genauigkeit ($\approx$ 7 signifikante Stellen) liefert.

```
1    VMUL.F32 q3, q3, q1
2    VMUL.F32 q3, q3, q2
3    VLDR d1, _M_4_PI
4    VMUL.F32 q3, q3, d1[0]
5    VMUL.F32 q3, q3, d1[1]
6    VLDR d1, _M_E_0
7    VMUL.F32 q3, q3, d1[0]
```

Nach Multiplikation der bisherigen Ergebnisse mit $r1 \cdot r2$ werden anschließend π und 4 in das Register *d1* geladen, um die Zwischenergebnisse mit diesen Skalaren zu multiplizieren. Dasselbe wird mit dem Faktor ε_0 gemacht.

```
1    VMUL.F32 q4, q3, d0[0]
2    VMUL.F32 q5, q3, d0[1]
```

Die endgültigen Ergebnisse erhält man letztendlich durch Multiplikation mit den beiden Dielektrizitätswerten, was hier durchgeführt wird. Anschließend befinden sich die Kapazitäten jeweils in den Registern *q4* für Dielektrizitätswert *1* und *q5* für Dielektrizitätswert *2*.

```
1    VSTM r3, {q4}
2    LDR r0, [r11,#0x4]
3    VSTM r0, {q5}
4    VPOP {q0-q5}
5    SUB sp,r11, #0x0
```

```
6     POP {r11}
7     BX lr
```

Damit der Aufrufer später auf die Ergebnisse zugreifen kann, müssen diese noch in den richtigen Speicherbereich kopiert werden. Dafür existiert der Befehl *VSTM*, der (ähnlich wie bei *VLDM*) mehrere Werte aus einem NEON-Register in den Speicherbereich, das von einem Standardregister referenziert wird, schreiben kann. Für *result1* (in Register *r3*) funktioniert das ohne Probleme, für *result2*, dessen Pointer gemäß Calling-Convention auf dem Stack übergeben wurde, ist ein weiterer Schritt nötig — die richtige Adresse muss zunächst vom Stack geholt werden. Anschließend werden die zuvor gesicherten Register und der Stackpointer wieder zurückgeschrieben und es wird mittels *BX* wieder an den Aufrufer (Adresse im Link-Register) zurückgesprungen.

3.1.4 Optimierungen

Durch Vorberechnung des konstanten Faktors der Formel ($4 \cdot \pi \cdot \varepsilon_0$) kann dieser in allen späteren Berechnungen wiederverwendet werden, was einigen Aufwand erspart. Diese Optimierung wurde nur in der VFP-Variante umgesetzt, da nur hier auch eine Schleifenstruktur implementiert wurde. Bei der NEON-Alternative ist also zusätzliches Optimierungspotential gegeben.
Die effiziente (minimale) Verwendung der verfügbaren Register bietet weiterhin Optimierungsmöglichkeiten. Hier haben wir versucht, den idealen Kompromiss zwischen Effizienz und Lesbarkeit zu finden.

3.2 Benutzer-Dokumentation

In dem Ordner, in dem das kompilierte Programm liegt, wird eine Datei namens *ui.txt* erwartet (eine Datei mit Beispielwerten ist bereits vorhanden). Die erste Zeile der Datei enthält die Anzahl der Datensätze. In den folgenden Zeilen stehen die inneren und äußeren Radien der Kugelkondensatoren, deren Kapazitäten zu berechnen ist. Diese Zeilen sind im Format *Radius1 mm Radius2 mm* (siehe auch die Beschreibung in der Entwicklerdokumentation).

Um die Kapazitäten zu berechnen, muss der Benutzer einfach das Programm in der Konsole ausführen.
Falls die Option *debug* angegeben wird, werden nur jeweils vier der berechneten Datensätze ausgegeben: Das erste, an den Stellen $n \cdot \frac{1}{3}$ und $n \cdot \frac{2}{3}$ und das letzte Ergebnis (n ist hier die Größe des Datensatzes).
Ohne weitere Parameter werden alle berechneten Ergebnisse ausgegeben.

4 Ergebnisse

4.1 Vergleich von Assembler und C-Code

4.1.1 Genauigkeit der Ergebnisse

Die Ergebnisse des Assemblercodes weichen um maximal $0{,}000135$ Promille von den Ergebnissen des C-Codes ab. Die erzielte Genauigkeit liegt dabei bei mindestens 5–6 Nachkommastellen. Die Ergebnisse des Assemblercodes mit NEON-Befehlssatz weichen um maximal $0{,}000225$ Promille von den Ergebnissen des C-Codes ab. Die Ergebnisse sind auf 5 Nachkommastellen genau.

4.1.2 Laufzeit

Für die Berechnung der Kapazitäten von 100000 Wertepaaren benötigt der C-Code im Durchschnitt durchschnittlich $25600\mu s$, der Assemblercode durchschnittlich $12800\mu s$ und der Assemblercode mit NEON-Befehlssatz nur durchschnittlich $3900\mu s$.
Damit ist der Assemblercode ungefähr 50% schneller und mit NEON sogar ca. 84% schneller als der C-Code.
Näheres zu den gemessenen Zeiten sowie den Abweichungen kann der beigefügten Datei *galaxys3O0.txt* entnommen werden (Hier muss beachtet werden, dass die Messwerte mangels anderer Testmöglichkeit von einem *Samsung Galaxy SIII* stammen, auf dem *BeagleBoard xM* beträgt die Laufzeit im Mittel das vierfache).

4.2 Analyse und Bewertung

Abschließend kann man sagen, dass die NEON-Implementierung zwar mit Abstand am schnellsten ist, dabei jedoch kleine Ungenauigkeiten auftreten, die durch die Tatsache hervorgerufen werden, dass der NEON-Befehlssatz keine direkte Division ermöglicht und man daher mit einer Annäherung an den Kehrbruch arbeiten muss.
Da sich die Abweichungen jedoch nicht wirklich bemerkbar machen und die Genauigkeitsverlust sehr gering ist, darf diese Implementierung wohl als die annähernd optimale Lösung angesehen werden.