I)Theoretischer Teil:

1)Eine Wahrscheinlichkeitsverteilung:

Damit eine Menge von Zahlen eine Wahrscheinlichkeitsverteilung sein kann, muss diese Menge Zahlen zwischen $(0, 1]$ enthalten und die Summe dieser Zahlen muss 1 sein.

2)Näherung vom Logarithmus Basis 2:

Wir wissen von der Analysis Vorlesung dass:

$$ln(x+1) = \sum_{v=1}^{n} \frac{(-1)^{v+1}}{v} * x^v$$

das heit, wir können nur$ln(x)$ mit $x$in $(0, 2]$ berechnen , und das genügt , da wir nur ln von Wahrscheinlichkeiten , die in [0,1] liegen , berechnen werden. Nachdem wir $ln(w)$ berechnen, wir können $Log_2(w)$ bestimmen, indem wir ein Basiswechsel machen.Durch:

$$log_2 x = \frac{lnw}{ln2}$$

3)Speicherstruktur fr Fliesskommazahlen:

Nach dem Recherchieren über dieses Thema, haben wir schon einige Lernstoffe gefunden. (z.B. https://www.h-schmidt.net/FloatConverter/IEEE754.html) In C und in Assembler, eine Fliesskommazahl (float) hat 32 Bits. Beim Speichern von einem float wird das float zuerst in Binärsystem in Form von Produkt von 1.abcdef und 2er Exponent verwandelt. Zum Beispiel:
0.5 in Dezimalsystem $= 1.0 * 2^{-1}$
16.75 in Dezimalsystem $= 1.000011 * 2^4$

daher kann man die Form so darstellen:

$$1.abcdef....*2^n$$

Die Ziffer vom Komma ist immer 1, und die Ziffern a, b, c, d, e, usw.. sind entweder 0 oder 1 in Binärsystem, und n ist das 2er Exponent. Beim Speichern von float wird genau diese Form verwenden. Man teilt dann 32 Bits so auf:

1 Bit fr Flag (0 für positiv, 1 für negativ)

8 Bits fr Exponent (reales Exponent +127)
23 Bits für Mantisse (die Ziffer 1 vom Komma ist nicht eingeschlossen)

Man muss hier auf zwei Punkte achten:

1. Nicht das echte Exponent, sondern das echte $Exponent + 127$ wird gespeichert. Zum Beispiel, für 0.5 in Dezimalsystem ist $dasExponent-1$, so hier wird $-1+127 = 126$ gespeichert. Analog dazu wird $4+127 = 131$ gespeichert.

2. Die Ziffern 1 vor dem Komma wird nicht gespeichert.Da für alle Zahlen steht immer 1 vor dem Komma,braucht man nicht die Ziffer zu speichern, so kann man mit 23 Bits aber 24 Bits von Mantissen behandeln. Zum Beispiel,für 0.5 ist die zu speichernden Mantisse 0000(23 mal 0).Analog dazu ist die Mantisse für 16.75 so: 0000110000.(23 $6 =$ 17 mal 0 nach 1111).

Hier würden wir aber ein totales Beispiel einführen: das Speichern von float Wert 8.5:
$8.5 = 1.0001 * 23$
Exponent: $3 + 127 = 130$, in Binärsystem $130 = 10000010$
Mantisse: 00010000.(23 $4 = 19$ mal 0 nach 1), in Dezimalsystem $= 524288$
So 8.5 wird so gespeichert:

0 1000 0010 0001 00000000000000000000

$->$ 1 Bit für Flag(0 , so der Wert ist positiv)
$->$ 8bit für Exponent($3 + 127 = 130$(dezimal)=1000 0010(binaer))
$->$ 23 bits für Mantisse

Das ist aber auch der Grund dafür, warum man sofort den Befehl vcvt aufrufen muss, wenn man ein Wert aus ARM Register in FPU Register mit dem Befehl vmov kopiert. Der Befehl vmov kopiert nicht den Wert, sondern alle Bits. Aus demselben Grund werden alle Bits von einem FPU Register nach ARM Register kopiert, wenn man z.B. vmov r0, s0 aufruft. Das bedeutet aber, wenn s0 beispielsweise 8.5f speichert, sind die Bits von r0 nach Aufruf von vmov r0, s0 so:

0 1000 0010 0001 00000000000000000000

Dieser Wert ist natürlich nicht 8 (8.5f nach unten abgerundet), sondern 1091043328 in Dezimalsystem.
Das Exponent von einem float f (fr Werte kleiner als 1.0f ist es Exponent + 1) ist aber gerade der Integer-Teil von dem Ergebnis von $log_2 f$.
Zum Beispiel:

$$log_2 64.8 = 6.0179, \text{integer-Teil ist 6, und } 64.8 = 1.00000011 * 2^6$$
$$log_2 0.0047 = -7.0589, \text{integer-Teil ist -7, und } 0.0075 = 1.11101 * 2^{-8}$$

Dann ist der Integer-Teil von $log_2 X$ sehr einfach zu erhalten: man soll zuerst X von FPU Register nach ARM Register kopieren, dann die Bits im ARM Register nach rechts um 23 Stellen schieben. Da wir bekommen immer positive Eingaben, ist das erste Bit immer 0. Das bedeutet, dass der Wert mit Shifts nach rechts um 23 Stellen gerade der Integer-Teil ist.

II)Praktischer Teil:

Wir schauen zuerst mal die Methode calc_entropy(float *data, unsigned int len) ohne SIMD-Befehle. Diese Methode macht folgende:
1. überprüfen, ob die Werte des Eingabearrays (data, mit Laenge len) die Eigenschaften einer Wahrscheinlichkeitsverteilung erfllen. Das bedeutet,für alle $i < len$ soll es gelten: $0 < data[i] <= 1.0f$ und $\sum_{i=0}^{n-1} data[i] = 1.0f$
2. berechnen die Entropie von dem Array und liefert das Ergebnis zurck

Darber hinaus gibt es zu der Implementierung von dieser Methode noch zwei Anforderungen:
1. Man muss den Logarithmus zur Basis zwei ($log_2 X$) in guter Näherung mit Hilfe entweder von einer Reihendarstellung oder von einem Tabellen-Lookup berechnen
2. Nur vier Grundrechenarten sind erlaubt $(+, -, *, /)$

Wir fangen die Implementierung mit Überprüfung auf Inputvalidität an. Wir mssen fr alle Elemente data[i] in dem Array prfen, ob $0 < data[i]1.0f$ ist. Weiter brauchen wir auch alle data[i] aufsummieren und prfen, ob die Summe gleich 1.0f ist. Weil es hier sich immer um Fliekommazahlen (FPU Register) behandelt, kann man hier den Befehl cmp nicht benutzen, was nur fr Integer (ARM Register) funktioniert. So man braucht hier einen anderen Befehl: vcmp (Schwerpunkt 1)
Schwerpunkt 1:vcmp
Die Grammatik fr vcmp bei unserem Projekt sieht so aus:

vcmp.f32 s1, s2 oder vcmp.f32 s1, #0

Man soll hier darauf achten, dass vcmp direkt für Zahlen funktionieren kann, aber diese Zahl darf nur 0 (#0) sein. Weiter ist es ganz entscheidend, dass im Vergleich zum Befehl cmp, vcmp keine Flags in APSR (ARM Statusregister) setzt, sondern in FPSCR (FPU Statusregister). Das heit aber, das Nutzen von allen Suffixen nach vcmp, die von Vergleichsergebnis abhngig sind (z.B. le, gt, ne, usw.), wird zum logischen Fehler fhren. Um dieses Problem zu loesen, braucht man einen anderen Befehl:

VMRS APSR_nzcv, FPSCR

Der Befehl vmrs transportiert den Inhalt in einem FPU Register in ein ARM Register. In diesem Fall werden die Flags von FPSCR an APSR transportiert. Dann kann man Suffixe reibungslos benutzen. Am Ende sieht der Code fr Inputüberprüfung so aus:

```
    input_valid:
mov r2, #0
vmov s0, r2 // s0: sum of elements in array
 vmov s2, #1.0 // s2: 1.0
 mov r3, r0 // r3: pointer to array

    valid_loop:
cmp r2, r1
beq valid_loop_end
vldr.f32 s1, [r3]
vcmp.f32 s1, #0 // vcmp can be directly used for #0, but not for #1-
Here: test if the element is not bigger that 0
vmrs APSR_nzcv, FPSCR // vcmp sets VFP flags, if you want to use
the flags, you have to transport it, which is here transport FPSCR to
APSR
ble invalid
vcmp.f32 s1, s2 // test if the element is bigger than 1.0
VMRS APSR_nzcv, FPSCR
bgt invalid
vadd.f32 s0, s0, s1 // sum up elements
add r2, r2, #1
add r3, r3, #4
b valid_loop

    valid_loop_end:
vcmp.f32 s0, s2 // sum up all elements should be 1.0
VMRS APSR_nzcv, FPSCR
bne invalid
bx lr
invalid:
vmov s0, #-1.0 // in case of invalid input you will get negative
output -1.0f
pop {pc}
```

Nach dem Überprüfen auf Inputvalidität implementieren wir dann die Berechnung von Entropie. Daher ist es am wichtigsten, eine Reihendarstellung oder in Tabellen-Lookup mit nur vier Grundrechenarten für $log_2 X$ zu

finden.

Jetzt fangen wir an, calc_entropy zu implementieren. Der ganze Code lsst sich in vier Schritte aufteilen:

1. Implementierung von der Schleife zum Berechnen für lnX:

$x = 1.0f - input;$
$exp\_x = 1.0f;$
$Loop\ for\ i \in [1, 500]:$
$\{$
$exp\_x* = x;$
$result- = exp\_x/i;$
$\}$

Der Assemble-Code dafür ist:

lnX:
// input for lnX is saved in s0
vmov s1, #1.0 vsub.f32 s1, s1, s0 // $s0 = 1 - X$ ¡==¿$X = 1 - s0$, and X is saved in s1
mov r2, #0 // s0:= sum of series, which is 0 at beginning, careful that vmov cannot
vmov s0, r2 // be directlly used for number 0
mov r2, #1 // r2: loop control n
vmov s3, #1.0 // $s3 := X^n$, which is 1 at beginning

loop2:
cmp r2, #500 // loop for 500 times
bgt end2
vmul.f32 s3, s3, s1 // $s3 = s3 * s1$ ¡==¿ $X = X^{(n-1)} * X$
vmov s2, r2
vcvt.f32.u32 s2, s2 // $s2 = n$
vdiv.f32 s4, s3, s2 // $\frac{X^n}{n}$ saved in s4
vsub.f32 s0, s0, s4 // $s3 = s3 - s4$, s3 is accumulated sum of $\frac{X^n}{n}$
add r2, r2, #1 // $n = n + 1$
b loop2

end2:
bx lr

2. Berechnen ln2 (Trick 1:Statt $ln2$ zu berechnen, berechnen wir eigentlich $ln0.5$, und $ln0.5 = ln\frac{1}{2} = -ln2$. Wir machen das so weil wir schon gefunden haben, dass mit unserem Algorithmus ln0.5 präziser als $ln2$ ist.) :

Der Assembler-Code dafür ist:
vmov s0, #0.5 // first calculate $ln2$, but instead of calculating ln2, I calculate $ln0.5$, which is more precise, and $ln2 = -ln0.5$
bl lnX // branch to lnX and set link register to next line
vmov s5 ,s0 // After branch to lnX, $ln0.5$ is saved in s0
vmov s0, #-1.0 vmul.f32 s5, s5, s0 // $ln2 = -ln0.5$, saved in s5

3. Implementierung von der Schleife zum Berechnen für Entropie:
$entropy = 0;$
Loop for i $\in$ [0, len - 1]:
{
$entropy- = data[i] * ln(data[i]);$
}
Der Assembler-Code dafür ist:
calc_entropy_old:
// calc_entropy_old is the method without SIMD instructions push {lr} // I want to use link register, so I have to save it before I use it
bl input_valid
// calculate ln2
vmov s0, #0.5 // first calculate ln2, but instead of calculating ln2, I calculate ln0.5, which is more precise, and ln2 = -ln0.5
bl lnX // branch to lnX and set link register to next line
vmov s5 ,s0 // After branch to lnX, ln0.5 is saved in s0
vmov s0, #-1.0
vmul.f32 s5, s5, s0 // ln2 = -ln0.5, saved in s5
mov r3, #1 // r3: loop control, which is 1 at beginning
mov r2, #0
vmov s7, r2 // s7: accumulated sum of -X * logX
loop_old:
cmp r3, r1 // loop for length of array
bgt end_old

```
vldr.f32 s0, [r0]
vmov s6, s0 bl lnX // After branch to InX, value InX is saved in s0
vmul.f32 s0, s0, s6 // X * lnX
vsub.f32 s7, s7, s0 // sum = sum - X * lnX
add r3, r3, #1 // n = n + 1
add r0, r0, #4 // r0 points to the next element in array
b loop_old
end_old:
vmov s0, s7
vdiv.f32 s0, s0, s5 // log2(X) = lnX / ln2, ln2 is saved in s5
pop {pc}
```

4. Entropy/=ln2 :
Der Assembler-Code dafr ist:

```
end_old:
vmov s0, s7
vdiv.f32 s0, s0, s5 // log2(X) = lnX / ln2, ln2 is saved in s5
pop {pc}
```

Bisher ist die Implementierung zum Berechnen von Entropie schon fertig. Aber wenn man einen Rückblick auf den Code wirft, ist es nicht so schwer zu entdecken, dass daher eine Optimierung mit SIMD-Befehlen mglich ist. Wir konvertieren dann auch Schritt für Schritt die Implementierung mit SIMD-Befehlen:
1.Versuchen, die Implementierung fr Überprüfung auf Eingaben zu konvertieren: Das Aufsummieren von Elementen im Array kann man natrlich mit SIMD-Befehlen verbessern, aber diese Verbesserung finden wir als entbehrlich. Weiter finden wir eine Reform für den Test auf $0.0f < data[i] 1.0f$ mit SIMD-Befehlen unrealisierbar. Deshalb machen wir keine weiteren Optimierungen für berprfung auf Eingaben.
2. Versuchen, die Implementierung zum Berechnen fr lnX zu konvertieren: Dieser Teil ist eigentlich nicht so schwer. Hier ist es aber wichtig zu beachten, dass abgesehen von der SIMD-Optimierung wir noch eine Vernderung gemacht: die Endergebnisse, die in q0 gespeichert werden, sind nicht einfach die Werte von $ln(data[i])$ fr vier floats, sondern die Werte von$data[i] * ln(data[i])$. Darber hinaus muss man darauf aufpassen, dass der Befehl vdiv nicht direkt fr q Register verwendet werden kann. (Trick 2: Statt $\frac{X^n}{n}$ mit vdiv zu berechnen, soll man zuerst $\frac{1}{n}$ rechnen, und dann das Ergebnis mit vdup zu duplizieren,

zuletzt den Befehl vmul fr $\frac{1}{n}$ und X^n verwenden.)

Der Assembler-Code dafr ist:

```
simd_lnX:
vmov s4, #1.0
vdup.32 q1, d2[0] // q1: vector for 1.0
sub r1, r1, #4
vldm r0!, q2 // q2: saves 4 floats from array begins with r0 and increases r0 parallel
vsub.f32 q3, q1, q2 // 1 − X = q2 <==> X = 1 − q2, and vector of X is saved in q3
mov r2, #0
vmov s20, r2
vdup.32 q5, d10[0] // q5: saves X^n/n
mov r2, #1

simd_lnX_loop:
cmp r2, #500 // loops for 500 times
bgt simd_lnX_end
vmov s16, r2
vcvt.f32.u32 s16, s16
vmov s17, #1.0
vdiv.f32 s16, s17, s16 // s16: saves 1/n, careful that vdiv cannot be used for q register, so I have to calculate 1/n and then use vmul instead of vdiv
vdup.32 q4, d8[0]
vmul.f32 q1, q1, q3 // X^(n−1) ∗ X
vmul.f32 q4, q1, q4 // X^n/n
vsub.f32 q5, q5, q4 // q5 saves sum for X^n/n
add r2, r2, #1
b simd_lnX_loop

simd_lnX_end:
vmul.f32 q5, q5, q2 // X ∗ lnX
vsub.f32 q0, q0, q5 // q0 saves sum for X ∗ lnX
bx lr
```

3. Versuchen, die Implementierung zum Berechnen für Entropie zu konvertieren: Dieser Teil ist auch nicht schwierig. Der einzige Punkt, auf den

man achten soll, ist, dass man in der Schleife für jede Runde überprüfen soll, ob es noch mehr als vier Elemente im Array gibt. Das kann man aber durch einen Vergleich zu der Länge von dem Array gut realisieren. Wenn die Anzahl von verbleibenden Elementen im Array kleiner als 4 ist, muss man auf simd_lnX verzichten und auf den alten Code lnX springen.

Am Ende sieht der Assembler-Code für diesen Teil so aus:

```
calc_entropy:
push {lr}
bl input_valid // input_valid tests the validity of input array
vpush {q4, q5} // q4 and q5 are not caller-safe
mov r2, #0
vmov s0, r2 // vmov cannot directly assign 0 to s/d register
vdup.32 q0, d0[0] // q0: vector sum for lnX / X

    loop1:
cmp r1, #4
blt end1 // end1:< 4 elements left in array
bl simd_lnX // simd_lnX: >= 4 elements left in array
b loop1

    end1:
vadd.f32 s0, s0, s1 // q0 saves vector sum for X * lnX, which means if you
sum up s0, s1, s2 and s3, all X * lnX are summed up
vadd.f32 s0, s0, s2
vadd.f32 s0, s0, s3
vmov s7, s0

    // Afer end1, loop_lessthan3 will be run
loop_lessthan3:
cmp r1, #0
ble end_lessthan3
vldr.f32 s0, [r0] // s0 saves a float from array begins with r0
vmov s6, s0
bl lnX // parameter for lnX is saved in s0, and result saved in s0 too
vmul.f32 s0, s0, s6 // X * lnX
vsub.f32 s7, s7, s0 // sum up X * lnX
add r0, r0, #4 // float takes 4 bytes
sub r1, r1, #1
b loop_lessthan3
```

```
    end_lessthan3:
vmov s0, #0.5 // calculate ln0.5
bl lnX
vmov s1, #-1.0
vmul.f32 s0, s0, s1 // ln2 = −ln0.5, I calculate ln0.5 instead of ln2 because
ln0.5 is preciser
vdiv.f32 s0, s7, s0 // log_2(X) = lnX/ln2
vpop {q4, q5}
pop {pc}
```

III)Bonus Aufgabe

1)Random-Input

Um ein Array mit einem groen Length von zufflligen Wahrscheinlichkeiten, deren Summe 1 ist , zu erzeugen:
- Wir erzeugen ein Array (Main Array) mit einem groen Length (zB 30000)
- Alle Elemente von Main Array sind Zahlen zwischen [1..30000]
- Mit Hilfe einem Sortierungsverfahren, wir erzeugen ein anderes Array (Second Array) , das die gleichen Elemente vom Main Array aber ohne Wiederholungen enthlt, und ein anderes Array mit dem gleichen Length (Third Array) , das die Anzahl von Wiederholungen jedes Element vom Second Array enthlt.
- Um die Wahrscheinlichkeiten zu bestimmen, wir erzeugen ein neues Array (Probability Array) mit dem gleichen Length und dessen Elemente $\frac{AnzahvonWiderholungen}{lengthvonmainArray}$ sind.

Beispiel:

-Main Array :

1	5	2	1	6	2	4	6

-Second Array (ohne Wiederholung):

1	2	4	5	6

-Third Array (Anzahl von Wiederholungen):

2	2	1	1	2

-Probability Array:

0.25	0.25	0.125	0.125	0.25

2)calc_entropy_approx Methode:

Bisher ist der praktische Teil für unser Projekt schon fertig.Wir

machen dann die zweite Bonus Aufgabe: eine andere Methode calc_entropy_approx zu implementieren, wobei man eine Annäherung zu $log_2 X$ lediglich mithilfe von Shifts (z.B. lsl, lsr, usw.) simulieren muss. Daher ist ein besonderes Thema betroffen: wie eine Fliekommazahl im Rechner gespeichert wird (Theoretischer Teil 3.).

Der Assembler-Code sieht dann so aus:

```
calc_entropy_approx:

    push {lr}
bl input_valid // input_valid tests the validity of input array
push {r4, r5}
mov r2, #0 // r2: loop control
vmov s0, r2 // s0: entropy sum for all elements in array

    loop3:
cmp r2, r1
bge end3
vldr.f32 s1,[r0] // lade a float from array to s1
bl log2_shift
vmul.f32 s1, s1, s2 // X * logX
vsub.f32 s0, s0, s1 // sum up and saved in s0
add r2, r2, #1
add r0, r0, #4
b loop3
end3:
pop {r4, r5}
pop {pc}

    log2_shift:
vmov r3, s1
lsl r4, r3, #9
lsr r4, r4, #9 // r4: saves mantissa of the float
lsr r3, r3, #23
sub r3, r3, #127 // r3: saves exponent of the float
vmov s2, r3
vcvt.f32.s32 s2, s2 // r3 is the integer part of logX, still have to calcu-
late the float part
b interpolate
```

Das Problem besteht darin, wie man die Mantisse-Teil von $log_2 X$ bekommen kann. Unsere Idee hier ist eine Interpolation (hier unten erklärt).

Interpolieren zwischen zwei ganzen Zahlen

Über dieses Thema haben wir eine Arbeit gefunden, und das Diagramm in dieser Arbeit ist ziemlich bedeutungsvoll (http://degiorgi.math.hr/aaa_sem/Div/97-105.pdf).

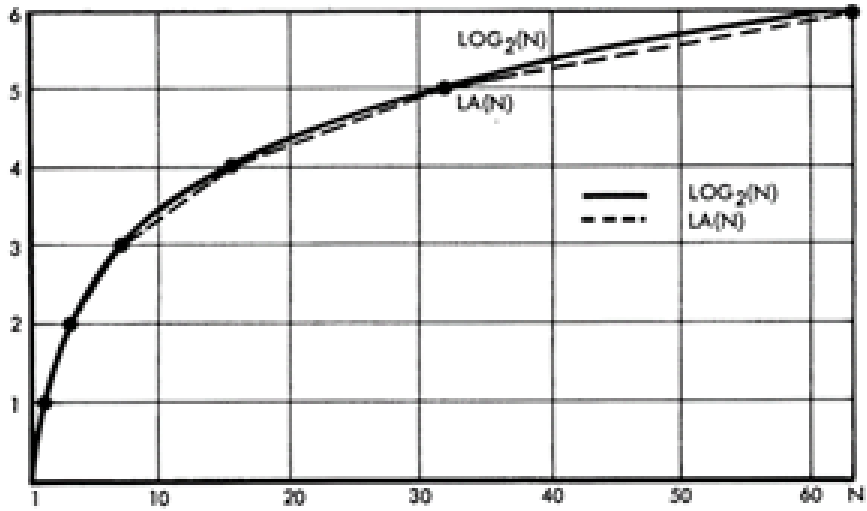


Fig. 1. Piecewise linear approximation to binary logarithm.

Die Idee daher ist, eine Interpolation durch Gerade zu simulieren. Zum Beispiel:

$log_2 150 \approx 7.2288$
$log_2 170 \approx 7.4094$

Die letzte 2er Potenz (kleiner als 150 und 170), der ein ganzzahliges Ergebnis von log_2 hat, ist 128, da $128 = 27$
Die nchste 2er Potenz (gröer als 150 und 170), der ein ganzzahliges Ergebnis von log_2 hat,ist 256, da $128 = 28$
Die Distanz zwischen 256 und 128 ist = 256 - 128 = 128 = 27. Die Distanz ist gleich die letzte 2er Potenz. Das ist aber kein Zufall, da

$2^{n+1}2^n = 2^n * (2-1) = 2^n$
Das Interpolieren sieht so aus:

$$log_2 150 \approx 7 + \frac{Distanz zwischen 150 und 128}{Distanz zwischen 256 und 128}$$
(oder $\approx 8 \frac{Distanz zwischen 256 und 150}{Distanz zwischen 256 und 128}$)

Zum Berechnen:

$log_2 150 \approx 7 + \frac{22}{128} = 7.171875$
(oder $8\frac{106}{128} = 7.171875$)

Analog dazu gilt:

$log_2 170 \approx 7.328125$

Für die Werte, die kleiner als 1.0f sind, ist der Prozess aber gleich.

Wir brauchen jetzt nur das Interpolieren zu implementieren. Und whrend der Implementierung haben wir noch weiteren Trick gefunden. (Trick 3)

Trick 3:
Angenommen, dass ein float f ist:

$$1.abcdef * 2^n$$

Die letzte 2er Potenz, die kleiner gleich als f ist, ist:

$$2^n$$

Die nchste 2er Potenz, die grer gleich als f ist, ist:

$$2^{n+1}$$

Die Distanz dazwischen ist:

$$2^{n+1}2^n = 2^n * (2-1) = 2^n$$

Die Distanz zwischen f und der letzten 2er Potenz 2n ist:

$$1.abcdef * 2^n 2^n = 2^n(1.abcdef - 1)$$

Das Interpolieren ist dann:

$$n + \frac{2^n * (1.abcdef - 1)}{2^n} = n + (1.abcdef - 1)$$

Der Wert 1.abcdef ist aber super einfach zu erzeugen: Man braucht nur den Exponent-Teil von f auf 127 setzen, da das Exponent von $1.abcdef = 1.abcdef * 2^0$ 0 ist, und $0 + 127 = 127$. Und danach kann man sehr leicht $1.abcdef1$ rechnen.

Der Assembler-Code für Interpolieren sieht so aus:

interpolate:
// basic idea:
if a float is $1.abcedf * 2^x$, to calculate(interpolate) between x and x+1 means: $x + (1.abcdef * 2^x - 1.0 * \frac{2^x}{2^x}) = x + 1.abcdef - 1.0$
cmp r4, #0 // r4 is the mantissa of the float. If r4 == 0, means that the float is already 2-exponent, thus the floar of lnX part is 0
bxeq lr
mov r5, #127
lsl r5, r5, #23
add r5, r5, r4
vmov s3, r5 // if the float is $1.abcde * 2^x$, now s3 is 1.abcde
vmov s4, #1.0
vsub.f32 s3, s3, s4 // s3 now is 1.abcdef - 1.0
vadd.f32 s2, s2, s3 // s2 saves x before, which is the integer part, and now s2 is x + 1.abcdef - 1.0
bx lr

3)Performanz und Genauigkeit :

Bisher ist die Implementierung schon fertig. Wir fangen jetzt an, auf die Performanz und die Genauigkeit zu prüfen. Um den Code besser zu testen und die Ergebnisse daraus besser zu vergleichen, schreiben wir noch zwei Funktionen in C:

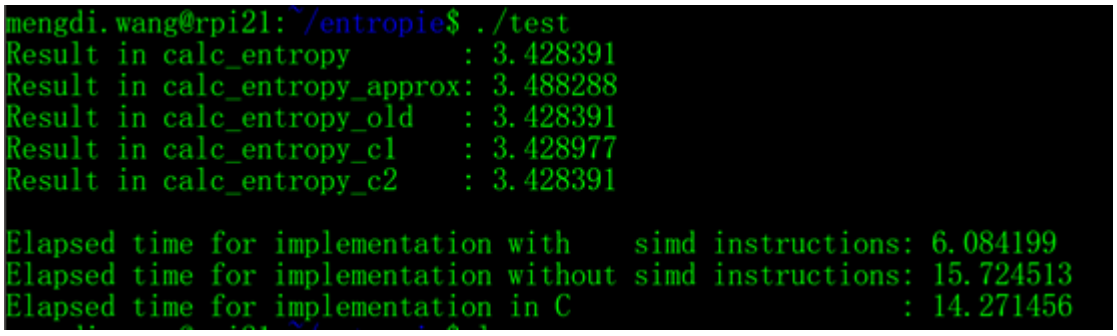
calc_entropy_c1: diese Methode nutzt die Bibliothek-Funktion von C, nämlich log() in ¡math.h¿. Diese Methode dient zum Standard der Berechnungsergebnisse. (Achtung 1: Weil wir hier Funktionen in

¡math.h¿ verwendet haben, mssen wir -lm als Parameter zum Kompilieren übergeben).

calc_entropy_c2: diese Methode implementiert auch die Reihendarstellung wie die Methode calc_entropy_old (ohne SIMD-Befehle), somit kann man zum Vergleich auf Performanz eine vergleichbare Standard-Laufzeit behalten. Diese Methode dient zum Standard der Laufzeit.

Zum Testen haben wir ein Array mit 16 Elementen:
float test[] = {0.05, 0.17, 0.07, 0.01, 0.2, 0.005, 0.13, 0.065, 0.07, 0.08, 0.025, 0.025, 0.006, 0.004, 0.055, 0.035 }; (Achtung 2: Weil manchmal beim Speichern von float die Werte nach unten abgerundet werden, muss man daher das Array modifizieren, sodass die Summe von allen Elementen 1.0f ist).

Beim Testen von Laufzeit führen wir das Programm 10 mal aus. (3 davon werden Screenshots). Und beim jedem Ausführen werden calc_entropy, calc_entropy_old und calc_entropy_c2 100000 mal aufgerufen. Wir stellen uns bevor wir die Tests starten vor, dass calc_entropy, calc_entropy_old und calc_entropy_c2 gleiche Berechnungsergebnisse haben sollen, die sich in der Nhe vom Ergebnis von calc_entropy_c1 befinden, da sie implementieren die Reihendarstellung gleich. Darüber hinaus soll calc_entropy (mit SIMD-Befehlen) fast 4-fach schneller als calc_entropy_old (ohne SIMD-Befehle) und calc_entropy_c2 (C Implementierung, auch ohne SIMD-Befehle). Weiter soll calc_entropy_approx ein Ergebnis hat, das nicht so weit von allen anderen Ergebnissen bleibt. Danach fangen wir die Tests an. Hier sind die 3 Screenshots dazu:



```
mengdi.wang@rpi21: /entropie$ ./test
Result in calc_entropy       : 3.428391
Result in calc_entropy_approx: 3.488288
Result in calc_entropy_old   : 3.428391
Result in calc_entropy_c1    : 3.428977
Result in calc_entropy_c2    : 3.428391

Elapsed time for implementation with    simd instructions: 6.230789
Elapsed time for implementation without simd instructions: 15.724377
Elapsed time for implementation in C                     : 14.271228
```

```
mengdi.wang@rpi21: /entropie$ ./test
Result in calc_entropy       : 3.428391
Result in calc_entropy_approx: 3.488288
Result in calc_entropy_old   : 3.428391
Result in calc_entropy_c1    : 3.428977
Result in calc_entropy_c2    : 3.428391

Elapsed time for implementation with    simd instructions: 6.178382
Elapsed time for implementation without simd instructions: 15.724142
Elapsed time for implementation in C                     : 14.271505
```

Bei den andren 7 mal Ausführungen haben wir sehr gleiche Berechnungsergebnisse sowie nahe Laufzeit betrachtet. Man kann daher einfach herausfinden, dass calc_entropy, calc_entropy_old und calc_entropy_c2 wie vorgestellt gleiche Ergebisse haben, und calc_entropy (mit SIMD-Befehlen) fast 4-fach schneller als calc_entropy_old und calc_entropy_c2 ist. Das Standard-Ergebnis, 3.428977, das calc_entropy_c1 liefert, ist nicht fern zu den Ergebnissen von Reihendarstellung (3.428391) und von Approximation mit Interpolieren (3.488288). Was uns überrascht ist aber, dass die Implementierung von Reihendarstellung in C (calc_entropy_c2) bessere Laufzeit als die Implementierung in Assembler (calc_entropy_old) hat.

Am Ende führen wir die üfung auf Genauigkeit. Hierbei werden die Berechnungsergebnisse von calc_entropy und von calc_entropy_approx mit Standard-Ergebnis von calc_entropy_c1 verglichen. Da wir interessieren uns nur auf Berechnungsergebnis von $log_2 X$, haben wir den Code ein bisschen verändert. (z.B. keine Überprufung auf Eingaben, Array hat immer nur 1 Element, Rückgabewert ist nicht mehr Entropie sondern einfach $log_2 X$). Beim ersten Test haben wir 14 Eingaben

(von 0.00001f bis 1.0f, aufsteigend sortiert). Wir stellen die Ergebnisse mithilfe von Tabelle dar:

Input	calc_entropy	_approx	Standard
0.00001	-9.792752	-16.689280	-16.609640
0.0001	-9.728719	-13.361600	-13.287712
0.001	-9.159520	-9.976000	-9.965784
0.01	-6.642250	-6.720000	-6.643856
0.1	-3.321925	-3.400000	-3.321978
0.2	-2.321928	-2.400000	-2.321928
0.3	-1.736965	-1.800000	-1.736965
0.4	-1.321928	-1.400000	-1.321928
0.5	-1.000000	-1.000000	-1.000000
0.6	-0.736965	-0.800000	-0.736965
0.7	-0.514573	-0.600000	-0.514573
0.8	-0.321928	-0.400000	-0.321928
0.9	-0.152003	-0.200000	-0.152003
1.0	0.000000	0.000000	0.000000

Beim zweiten Test haben wir 10 Eingaben (von 1.0f bis 0.015625f, absteigend sortiert, alle Eingaben sind 2er Potenz). Wir fassen die Daten wie oben in Form von Tabelle zusammen:

Input	calc_entropy	_approx	Standard
$1.0 = 2^{0}$	0.000000	0.000000	0.000000
$0.5 = 2^{-1}$	-1.000000	-1.000000	-1.000000
$0.25 = 2^{-2}$	-1.999999	-2.000000	-2.000000
$0.125 = 2^{-3}$	-3.000000	-3.000000	-3.000000
$0.0625 = 2^{-4}$	-3.999997	-4.000000	-4.000000
$0.03125 = 2^{-5}$	-4.999994	-5.000000	-5.000000
$0.015625 = 2^{-6}$	-5.999931	-6.000000	-6.000000
$0.0078125 = 2^{-7}$	-6.994024	-7.000000	-7.000000
$0.00390625 = 2^{-8}$	-7.925310	-8.000000	-8.000000
$0.001953125 = 2^{-9}$	-8.671836	-9.000000	-9.000000

Nachdem man die Daten analysiert, ist es nicht schwer zu entdecken, dass:
1. Der Approximationsfehler von calc_entropy wird grösser, wenn In-

putwert klein wird. Wenn der Inputwert ziemlich klein ist, z.B. 0.00001f, ist der Fehler auffallend gro.
2. Der Approximationsfehler von calc_entropy_approx wird grösser, wenn Inputwert grsser wird. (abgesehen davon, dass der Inputwert 2er Potenz ist)
3. calc_entropy_approx hat immer kleine Ergebnisse (grere Betrge) im Vergleich zu calc_entropy. Das ist aber schon wie vorgestellt klar, weil die Funktion $log_2(X)$ konkav ist. Das fhrt dazu, dass das Interpolieren mithilfe von Gerade zwischen zwei benachbarten 2er Potenzen immer unterhalb der Kurve von $log_2(X)$ bleibt.
4. calc_entropy_approx hat gar keinen Approximationsfehler, wenn der Inputwert gerade eine 2er Potenz ist. Das ist auch wie vorher vorgestellt klar, weil in diesem Fall man keine Interpolation braucht.