Technische Universität München
Fakultät für Informatik
Rechnerarchitektur-Praktikum
SS 2018

ASSEMBLER POLYNOME

SPEZIFIKATION

Bearbeitet von:
Oleg Patrascu
Matthias Unterfrauner

10.05.2018

Inhalt

1. Aufgabenkurzbeschreibung

Es soll ein Assembler Programm zur Berechnung von Polynomkoeffizienten und der Normierungskonstante realisiert werden.

$$f(x) = \sum_{i=0}^{N} a_i x^i = c(1 * x^N + \sum_{i=0}^{N-1} b_i x^i)$$

Das Polynom ist in C dargestellt als:

```
struct polynom {
int iDegree;
float* p_fCoefficients ;
};
```

Das Polynom mit Koeffizienten a_i erhalten wir als Input und als Output liefern wir die b_i Koeffizienten und die Normierungskonstante c, d.h. wir müssen folgende Funktion in Assembly implementieren:

```
float norm(struct polynom *input, struct polynom *output)
//liefert c zurück
```

Beispiel:

Input :

```
struct polynom A  = {
iDegree = 2;
float p_fCoefficients = [1, 2, 0.5];
};
//Äqv. zu 1 + 2*x + 0.5*x^2
```

Gewünschter Output :

```
struct polynom B  = {
iDegree = 2;
float p_fCoefficients = [2, 4, 1];
};
//Äqv. zu 2 + 4*x + 1*x^2

c = 0.5 //Normierungskonstante
```

1. **Ist-Analyse**
   Wie ein Polynom aufgebaut ist wissen wir schon und wir kennen auch algebraische Regeln um die Formel zu vereinfachen, sodass wir die Koeffizienten und Konstante leichter berechnen können. Auch wissen wir wie Assembler Funktionen aus einem C Programm aufgerufen werden und wie man C Programme in Assembler übersetzt.

2. **Soll-Analyse**
   Wir müssen uns zuerst mit den FPU Befehlen vertraut machen, da sich diese unterscheiden von jenen, welche wir in der Vorlesung kennengelernt haben. Wir müssen uns auch über die Struktur der Make Datei informieren, damit wir unser Programm leichter kompilieren können.

3. Erfolgskriterium

Ob unsere Implementierung die benötigte Ergebnisse liefert, werden Unit - Tests alle Fälle abdecken aber auch mögliche Fehler werden beim Code Refactoring korrigiert.

2. Lösungsansätze

Natürlich müssen wirzuerst die Formel $f(x) = \sum_{i=0}^{N} a_i x^i = c(1 * x^N + \sum_{i=0}^{N-1} b_i x^i)$ vereinfachen, um die Koeffizienten b_i und die Konstante c leichter finden zu können.

1. $\sum_{i=0}^{N} a_i x^i = c(1 * x^N + \sum_{i=0}^{N-1} b_i x^i)$
2. $\sum_{i=0}^{N} a_i x^i = cx^N + c\sum_{i=0}^{N-1} b_i x^i$
3. $\sum_{i=0}^{N} a_i x^i = cx^N + \sum_{i=0}^{N-1} cb_i x^i$
4. $a_N x^N + \sum_{i=0}^{N-1} a_i x^i = cx^N + \sum_{i=0}^{N-1} cb_i x^i$
5. $a_N x^N = cx^N$ und $\sum_{i=0}^{N-1} a_i x^i = \sum_{i=0}^{N-1} cb_i x^i$
6. $c = a_N$ und $a_i = cb_i$
7. $c = a_N$ und $b_i = \frac{a_i}{c} = \frac{a_i}{a_N}$

Wir haben 2 Lösungen zu der Aufgabe gefunden, essentiell ist hierbei die effizienz des Ladens der Koeffizienten in den Hauptspeicher. **Lösungsansatz B** verwendet spezielle Befehle und ist somit effizienter aber auch koplexer als **Lösungsansatz A**. **Lösungsansatz A** hingegen iteriert einfach durch den gegebenen Speicherbereich und ist somit langsamer aber weniger komplex.

2.1. Lösungsansatz A

Die Idee bei dieser Lösung ist, dass wir nur zwei Register verwenden um die Koeffizienten des Polynoms zu berechnen (zuerst ins Register laden und mithilfe eines anderen Registers dividieren). Die Laufzeit des Algorithmus beträgt O(n) weil wir einfach durch alle Koeffizienten des Eingabepolynoms iterieren und dividieren und so erhalten wir die $b_i = \frac{a_i}{a_N}$.

High Level Algorithmus:

```
//A ist Eingabe- und B Ausgabepolynom
float norm(A, B) {
N = A.iDegree //Grad des Polynoms
B.coefficients = new float[N]
a_N = A.coefficients[N - 1]
i = 0

while (i < N - 1) {
    B.coefficients[i] = A.coefficients[i] / a_N
    i=i+1
}

B.coefficients[N - 1] = 1 //Laut der Angabe

return a_N
}
```

Der Algorithmus berechnet die Koeffizientenwerte für Polynom B und liefert die benötigte Normierungskonstante.

2.2. Lösungsansatz B

Dieser Lösungsansatz verwendet eine SIMD Instruktion *v2 = _mm_div_pd(v1, const)* , diese nimmt einen Vektor als Parameter *v1* und eine Konstante const und liefert wieder ein Vektor *v2* mit *v2 = v1/const*.
Sie erlaubt eine sehr schnelle Division von allen Termen des Polynoms, in diesem Fall müssen wir nicht manuell durch jede Koeffizienten durchlaufen. Das ist eine Eigenschaft einer SIMD Instruktion.
Mit der Instruktion *B.coefficients = _mm_div_pd(A.coefficients, a_N)* werden alle $b_i = \frac{a_i}{a_N}$ auf einmal berechnet d.h. sie braucht deutlich wenig CPU Zykeln im Vergleich zu Lösungsansatz A. Obwohl die Laufzeit des Algorithmus wieder *O(n)* beträgt, wird unser Programm schneller als mit **Lösungsansatz A** ausgeführt, wegen den Eigenschaften der SIMD Instruktion (insbesondere bei vielen Koeffizienten).

3. Bewertung der Ansätze

Lösungsansatz A		**Lösungsansatz B**	
Vorteile	**Nachteile**	**Vorteile**	**Nachteile**
• Einfache und nachvollziehbare Implementierung.	• Falls N sehr groß ist wie z.B. 10^(10) wesentlich langsamer als Lösungsansatz B	• Wegen der SIMD Instruktionen sehr schnell.	• Komplizierte Implementierung. • Laut der Angabe muss die Lösung nur mit FPU Befehle implementiert werden. • Fehleranfälliger als Lösungsansatz A

4. Entscheidungswahl

Da Lösungsansatz B fehleranfälliger und komplexer ist wählen wir Lösungsansatz A.

5. Aufgabenverteilung

Arbeitspaket	**Verantwortlicher**
Dokumentation	Oleg Patrascu
Vortrag	Matthias Unterfrauner

6. Zeitplanung

Zeitraum	**Aufgabe**	**Aufwand (Std.)**
14.05.2018 - 24.05.2018	Notwendige FPU Dokumentation für die Implementierung	6
25.05.2018 - 03.06.2018	Implementierung	14
04.06.2018 - 10.06.2018	Unit-Tests & Refactoring	6
11.06.2018 - 17.06.2018	Erstellung des Readme & Makefile	1
18.06.2018 - 27.06.2018	Entwicklerdokumentation	4
28.06.2018 - 08.07.2018	Anwenderdokumentation	4

Abgabefristen

Zeit	Abgabe
17.06.2018	Implementierung
08.07.2018	Ausarbeitung
23.07.2018 - 03.08.2018	Vortrag