Schleifen wegoptimieren

Timm Knape (timm@knp.de)

17. März 2021

1 Aufgabenstellung

Während meines Informatik-Studiums haben wir die Programmiersprache Modula-2 gelernt. Mühevoll zogen sich die Wochen dahin, in denen wir die Grundstrukturen verdauten. Irgendwann wurden Schleifen eingeführt und es kam auf einem Übungszettel zu folgender Aufgabe:

Schreibe eine Funktion, welche die Summe der ersten n natürlichen Zahlen berechnet.

Erwartet war irgendetwas in der Form

```
PROCEDURE Sum (n: INTEGER): INTEGER;
    VAR i, s: INTEGER;
BEGIN
    s := 0;
    FOR i := 1 TO n DO
        s := s + 1;
    END;
    RETURN s;
END Sum;
```

In C++ kann die Funktion etwa so aussehen:

```
int sum (int n) {
    int s { 0 };
    for (int i { 1 }; i <= n; ++i) {
        s += i;
    }
    return i;
}
```

Bleiben wir für den Rest dieses Beitrags in dieser Sprache. Meine Modula-2 Kentnisse sind doch arg eingerostet.

2 Gaußsche Summenformel

Ich war natürlich vorwitzig und sah nicht ein, nur zur Übung eine Schleife zu verwenden, wenn es nach der Summenformel von Carl Friedrich Gauß auch viel eleganter geht:

```
int sum (int n) {
    return n * (n + 1)/2;
}
```

In der Schule bekam die Klasse die Aufgabe gestellt, die ersten 100 Zahlen (andere Quellen sprechen von 60) zusammenzuzählen. Der neunjährige Gauß hatte nach wenigen Sekunden die richtige Lösung im Kopf berechnet.

Sein Rechenweg funktioniert am besten für gerade n. Statt dessen können wir die ersten n Zahlen zweimal aufschreiben. Einmal vorwärts und darunter einmal rückwärts:

1	2	3	$\cdots$	$n-1$	n
n	$n-1$	$n-2$	$\cdots$	2	1

Die Tabelle hat n Spalten und die Summe jeder Spalte ist $n+1$. Die Summe aller Zahlen in der Tabelle

beträgt also $n \cdot (n + 1)$. Da die Zahlen aber zweimal aufgeschrieben wurden, muss das Ergebnis noch durch zwei geteilt werden.

3 Überlauf

Ich habe auf diese Aufgabe die vollen Punkte bekommen. Inzwischen frage ich mich jedoch, ob dies gerechtfertigt war: Kann es nicht sein, dass durch einen Überlauf falsche Werte berechnet werden, die in einer Schleife langsam, aber korrekt, berechnet worden wären?

In vielen Sprachen hat eine Integer-Variable nur eine feste Länge. Bei meinem System sind das 32 Bit. Eine ganze Zahl kann damit nur 2^{32} unterschiedliche Werte annehmen. Das sind die Zahlen -2^{31} bis $2^{31} - 1$, da der Datentyp **int** auch negative Zahlen unterstützt (genauso wie der Datentyp **INTEGER** in Modula-2).

Auch die Schleife liefert nicht für jedes mögliche n die richtige Lösung. Irgendwann wird die Summe größer oder gleich 2^{31} und es entsteht ein Überlauf. Das Ergebnis ist gültig, so lange es klein genug ist:

$$2^{31} > \frac{n \cdot (n + 1)}{2}.$$

Hier nutze ich die Gauß-Formel für die Abschätzung der Schleifen-Implementierung. Denn nun befinden wir uns in der reinen Mathematik, in der es keine Überläufe gibt.

Durch Multiplizieren mit 2 ergibt sich

$$2^{32} > n \cdot (n + 1).$$

Die Ungleichung bleibt bestehen, wenn das $n + 1$ durch n ersetzt wird. Dadurch wird die rechte Seite nur noch kleiner:

$$2^{32} > n^2.$$

Und durch Ziehen der Quadrat-Wurzel ergibt sich

$$2^{16} > n.$$

Diese Grenze ist scharf! Schon mit $n = 2^{16}$ läßt sich $n \cdot (n + 1)$ nicht mehr mit 32 Bit darstellen. Somit können wir auch mit der Schleife nur bis zur Zahl 65.535 aufsummieren, die gerade noch mit 16 Bit dargestellt werden kann. Bei größeren Zahlen gibt es einen Überlauf und ein falsches Ergebnis.

Bei der oben angegebenen Umsetzung der Gauß-Methode tritt der Überlauf schon viel früher ein. Schon wenn n mit $n + 1$ multipliziert wird, ist das Ergebnis nur gültig, wenn

$$2^{31} > n \cdot (n + 1)$$

gilt. Durch die gleiche Abschätzung wie oben ergibt sich

$$2^{31} > n^2$$

und wieder durch die Anwendung der Quadrat-Wurzel resultiert

$$\sqrt{2} \cdot 2^{15} > 46.340 \geq n.$$

Schon mit $n = 46.341$ passt das Ergebnis nicht mehr in 31 Bit und liefert ein negatives Ergebnis. Für alle Zahlen von 46.341 bis 65.535 erhalten wir mit dem Gauß-Algorithmus keine korrekte Lösung. Die Schleife kann für über 40 % mehr Werte das richtige Ergebnis berechnen!

4 Gauß ohne Überlauf

Der Überlauf kann verhindert werden, wenn wir entweder n oder $n + 1$ erst durch 2 teilen und dann mit dem anderen Faktor multiplizieren. Jedoch dürfen wir nur die Zahl durch 2 teilen, die gerade ist. Andernfalls entstehen Rundungsfehler.

Das kann durch eine Fallunterscheidung erreicht werden:

```
int sum (int n) {
    if (n % 2) {
        return (n + 1)/2 * n;
    } else {
        return n/2 * (n + 1);
    }
}
```

Damit funktioniert Gauß wieder bis zur Grenze 65.535. Und erst damit ist die Aufgabe eigentlich gelöst.

Der Compiler wird hoffentlich die Modulo-Operation durch eine Und-Verknüpfung und die Division durch eine Verschiebe-Operation ersetzen, die auf den meisten Prozessoren deutlich schneller sind.

Wenn wir dem Compiler nicht trauen, können wir auch direkt auf Bit-Ebene die Anweisungen geben:

```
int sum (int n) {
    if (n & 1) {
        return ((n + 1) >> 1) * n;
    } else {
        return (n >> 1) * (n + 1);
    }
}
```

Ein Test-Szenario hat mit der Schleife knapp eine Sekunde benötigt. Dieser Algorithmus arbeitet den gleichen Test-Parcours in 3 Millisekunden mehr als 100 Mal ab.

Aber es geht *noch* besser. Die Fallunterscheidung macht bei aktuellen Prozessor-Pipelines Probleme. Wenn der Prozessor den falschen Weg rät, muss die Pipeline neu angekurbelt werden. Besser und effizienter sind Lösungen, die keine Sprünge benötigen. Und die gibt es.

Hier möchte ich eine Low-Level Variante vorstellen. Wir wissen, dass n nicht größer als 65.535 werden kann. Dieses n passt aber noch in 16 Bit. Dann passt das Produkt von n und $n + 1$ aber noch in 32 Bit. Nicht in 31 Bit, aber in 32! Somit klappt alles, wenn wir mit Zahlen rechnen, die keinen negativen Wertebereich haben. Damit ergibt sich die folgende Lösung für diesen Artikel:

```
inline int sum(int n) {
    auto un {static_cast<unsigned>(n)};
    return static_cast<int>(
        (un * (un + 1)) >> 1
    );
}
```

Der Wechsel nach **unsigned** ist notwendig, damit bei der Verschiebung immer 0-Bits nachgezogen werden. Bei **int** bleibt das oberste Bit unberücksichtigt (und dadurch bleibt die Zahl negativ).

Sieht gewaltig aus (mit dem **static_cast**), aber der generierte Code ist sehr effizient. Es müssen nur

andere Maschinenbefehle verwendet werden.

Die Laufzeit fällt von 3 Millisekunden auf 2,5 Millisekunden.

5 Andere Programmiersprachen

Viele andere Sprachen haben ebenfalls nur endlich viele Werte für ihre ganzzahligen Datentypen zur Verfügung. Typische Vertreter sind:

Java, C#, Rust, Go.

Es gibt aber Sprachen, die intern mit beliebig großen Zahlen rechnen können:

Scheme/Lisp, Python, Ruby.

Unter Python können wir die Standard-Version des Gauß-Algorithmus verwenden:

```
def sum(n):
    return n * (n + 1) // 2
```

Jedoch ist der Umgang mit beliebig großen ganze Zahlen ein erheblicher Mehraufwand für den Rechner. Anstatt direkt in eimen Register zu rechnen, müssen die Zahlen über mehrere Worte des Arbeitsspeichers verteilt oder in mehreren Registern abgelegt werden. Dabei entstehen zwangsweise Sprünge, da von vorne herein nicht klar ist, wie viel Speicher die Zahl belegt.

6 Zusammenfassung

Selbst bei so einer einfachen Schleifen-Optimierung ist Vorsicht angesagt. Die enorme Kosteneinsparung kann ggf. eine Einschränkung des Anwendungsbereichs zur Folge haben. Wenn wir dies nicht beachten, produziert der verbesserte Code schwer zu findende Fehler. Mit etwas Nachdenken finden sich aber häufig verbesserte Lösungen.

Nicht jeder Code muss bis zu seinem Maximum optimiert werden. Schon die Beschreibung des Maximums ist nicht leicht. Soll der Code möglichst schnell laufen? Oder möglichst klein sein? Beides schließt sich oft aus. Schön ist, wenn wie in diesem Beispiel eine kleine, schnelle Lösung existiert.

Leider rentiert es sich oft nicht, eine solche Lösung zu suchen. Die Rechner sind schnell genug, dass auch suboptimale Lösungen verwendet werden können. Diese verbrauchen jedoch mehr Energie, Speicher und Rechenzeit als eine bessere Lösung, über die wir ein klein wenig mehr nachgedacht hat.

7 Weiter Denken

Weiter Denken 1. JavaScript verwendet nur 64-Bit Floating-Point Zahlen. Wie weit kann die Schleifen-Variante korrekte Ergebnisse liefern? Wie weit der einfache Gauß?

Weiter Denken 2. Wie verhalten sich Systeme, auf denen 64-Bit Integer-Zahlen verwendet werden?

Weiter Denken 3. Wie verhalten sich Systeme, die direkt mit **unsigned** Zahlen rechnen? Warum kann die letzte C++ Variante dort nicht verwendet werden?

Weiter Denken 4. Ein ähnliches Problem tritt beim Berechnen eines Mittelwerts auf:

$$\left\lfloor \frac{a+b}{2} \right\rfloor$$

Formuliere eine ggf. ineffiziente Scheife, die Überläufe verhindert.

Weiter Denken 5. Kann beim Mittelwert auch eine Lösung entwickelt werden, welche das zusätzliche Vorzeichen-Bit ausnutzt? a und b können auch beide gerade oder ungerade sein.