Schleifen wegoptimieren

Timm Knape

12. 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 ganzen Zahlen liefert.*

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.

2 Gauß-Methode

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

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

Der junge Gauß hat der Legende nach in der Schule die Strafarbeit erhalten, die ersten hundert Zahlen zusammenzuaddieren. Er war jedoch nach kurzer Zeit fertig.

Er schrieb die Zahlen einmal vorwärts und einmal rückwärts auf:

1	2	3	...	$n-1$	n
n	$n-1$	$n-2$	...	2	1

Die Summe der Spalten ist stets $n+1$. 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. Auf meinem System sind das 32 Bit. Wenn das Produkt zweier Zahlen größer oder gleich 2^{32} wird, dann wird das Ergebnis abgeschnitten. Schon wenn das Ergebnis 2^{31} erreicht, entstehen Probleme: Das oberste Bit ist für das Vorzeichen reserviert. Das Ergebnis wird als negative Zahl angezeigt, wenn das höchste Bit gesetzt ist.

Natürlich klappt auch nur die Lösung mit der Schleife in einem bestimmten Werte-Bereich. Dieser ist jedoch viel größer. Bei der Schleife muss nur das Ergebnis kleiner 2^{31} sein, also

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

Durch Multiplizieren mit 2 ergibt sich

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

Die Ungleichung bleibt bestehen, wenn das $n + 1$ durch n ersetzt wird

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

Und durch Ziehen der Quadrat-Wurzel ergibt sich

$$2^{16} > n$$

Und diese Grenze ist scharf! Schon mit $n = 2^{16}$ ist $n \cdot (n + 1)$ größer als 32 Bit. Auch mit der Schleife können wir nur bis zur Zahl 65.535 aufsummieren, die gerade noch in 16 Bit hineinpasst. Bei größeren Zahlen gibt es einen Überlauf im Ergebnis.

Bei der trivialen Umsetzung der Gauß-Methode tritt der Überlauf schon viel früher ein. Schon wenn n mit $n + 1$ multipliziert wird, kann ein Überlauf entstehen, wenn nicht

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

gilt. Durch die gleiche Abschätzung ergibt sich

$$2^{31} > n$$

und weiter durch die Wurzel

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

Schon mit $n = 46.341$ passt das Ergebnis nicht mehr in 31 Bit.

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.

Eine erste Lösung ist eine Fallunterscheidung:

```
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 weniger Takte benötigen (und damit schneller sind).

Wenn man dem Compiler nicht traut, kann man 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);
    }
}
```

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. Hier der Vorschlag:

```
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.

5 Andere Programmiersprachen

Viele andere Sprachen haben das gleiche Problem:

1. Java,
2. C#,
3. Rust.

4. Go.

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

1. Scheme/Lisp und
2. Python,
3. Ruby.

Unter Python können wir einfach 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 man dies nicht beachtet, 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 es, 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 man ein klein wenig mehr nachgedacht hat.

7 Weiter denken

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

Wie verhalten sich Systeme, auf denen 64-Bit Integer-Zahlen verwendet werden können?

Wie verhalten sich Systeme, die direkt mit **unsigned** Zahlen rechnen? Warum kann die Shift-Variante dort nicht verwendet werden?

Ein ähnliches Problem tritt beim Berechnen eines Mittelwerts auf:

$$\frac{a+b}{2}$$

Wie verhalten sich hier die Grenzen?

Kann auch eine Shift-Variante entwickelt werden, welche das zusätzliche Vorzeichen-Bit ausnutzt?