Übungsblatt 02: Speicherhierarchie richtig nutzen

7. November 2019

Allgemeine Hinweise

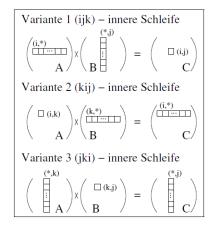
- Abgabetermin für die Lösungen ist Freitag, 15.11.2018, 11:59 Uhr.
- Abgabe an hannah.muelbaier@gmail.com UND carla.moelbert@gmx.de

Aufgabe 2.1: Matrizenmultiplikation (10 Punkte)

Eine Matrizenmultiplikation ist ein gutes Beispiel dafür, wie man die Speicherhierarchie nicht oder auch optimal nutzen kann.

Implementiere eine Matrizenmultiplikation in C/C++ (d.h. lauffähig, kein Pseudocode!):

- (a) Der Einfachheit halber haben die zu multiplizierenden Matrizen die gleichen Dimensionen und sind quadratisch: also z.B. 8x8 Felder groß
- (b) Beachte, dass bei den in Aufgabe 2.2 geforderten Dimensionen ein statisches Array nicht funktioniert, d.h. die Arrays müssen dynamisch sein (Stichworte: Heap, Stack, Speicherverwaltung)
- (c) Die Abbildung der zweidimensionalen Matrizen erfolgt auf eindimensionale Arrays
- (d) Die Quellmatrizen sollen mit zufälligen Integer-Werten gefüllt, die Zielmatrize mit 0 initialisiert
- (e) Implementiere drei Varianten, welche die Berechnung in verschiedenen Reihenfolgen abarbeiten:



Aufgabe 2.2: Auswertung (10 Punkte)

Führe dein Programm mehrfach mit den folgenden Matrixdimensionen aus: 64, 128, 256, 512, 1024, 2048. **Achtung**: die Ausführung für die Größen 1024 und 2048 kann mehrere Minuten dauern!

Notiere die aus den mehrfachen Durchläufen berechneten mittleren Laufzeiten je Dimension:

- (a) für die Variante 1 (Zeile x Spalte = Zielfeld)
- (b) für die Variante 2 (Quellfeld x Zeile = Zielzeile)
- (c) für die Variante 3 (Spalte x Quellfeld = Zielspalte)

Beantworte die folgenden Fragen:

- (a) Welche Variante ist die schnellste?
- (b) Welche Variante ist die langsamste?
- (c) Überlege dir dazu, wie die Daten im Arbeitsspeicher angeordnet sind und wie sie von dort zur CPU transportiert werden. Beschreibe dann kurz, wie die unterschiedlichen Laufzeiten zustande kommen.

Codeschnipsel

```
#include <stdlib.h>
#include <sys/time.h>
   #include <time.h>
   #include <stdio.h>
#include <unistd.h>
    int main (int argc, const char* argv[]) {
       // teste, ob ein Argument uebergeben wird
       // falls nicht, brich das Programm ab if (argc < 2) {
11
          return 1;
       }
13
       // die Dimension der Matrizen entspricht dem Argument
int dim = atoi(argv[1]);
15
        \begin{array}{l} printf("Integergroesse: \%lu \backslash n", \ sizeof(int)); \\ printf("Matrixgroesse: \%d \ x \ \%d \backslash n", \ dim, \ dim); \\ printf("Die Matrizen belegen jeweils \ \%lu \ bytes \ an \ Speicher. \backslash n", \\ \end{array} 
19
       dim * dim * sizeof(int));
printf("Insgesamt also %lu bytes.\n", 3 * dim * dim * sizeof(int));
21
       // AUFGABE: "erzeuge" hier die Matrizen
25
       // Messinstrumente
       struct timeval start, end;
long mtime, seconds, useconds;
27
29
       // Initialisiere Random Number Generator
srand((unsigned) time(NULL));
31
       // AUFGABE: fuelle hier die Matrizen
       // starte die Zeitnahme
       gettimeofday(&start, NULL);
37
       // AUFGABE: multipliziere hier die Matrizen
/* Hinweis: Kleine Matrizen auszugeben ist zwar nuetzlich,
um die Berechnung zu ueberpruefen, aber es kostet viel Zeit.
39
       Darum hat es nichts im entgueltigen Code zu suchen, kann aber
       bei der fehlerfreien Implementierung helfen.
43
       // beende die Zeitnahme und gib die Dauer der Berechnung aus
45
       gettimeofday(&end, NULL);
       gettimeorday(&end, No.D),
seconds = end.tv_sec - start.tv_sec;
useconds = end.tv_usec - start.tv_usec;
mtime = ((seconds) * 1000 + useconds/1000.0) + 0.5;
printf("Die Ausfuehrung dauerte %ld ms.\n", mtime);
49
       // AUFGABE: Raeume den Speicher auf
53
       // beende das Programm
       return 0;
```

Listing 1: Codeschnipsel zur Matrizenmultiplikation