# Effizienzsteigerung durch Arrays und Vektoren

Autor: Stefan Duscher

Anlass: Schulung VHS Ludwigsburg

## Vorwort

In Python kann man sowohl seriell als auch parallel programmieren, wobei dies die Form der Verarbeitung der Rechenschritte beschreibt: Während bei der seriellen Programmierung zunächst ein Element berechnet wird und sich dann dem nächsten Element gewidmet wird, können bei der parallelen Programmierung Rechenschritte gleichzeitig auf eine Vielzahl von Elementen vollzogen werden. Für die parallele Programmierung bietet sich die Problembeschreibung mittels Arrays förmlich an. Gleichwohl ist es aber eine Frage der Programmierung, d.h. Python wird nicht selbständig das effizienteste Verfahren wählen.

## Beispiel - Teilbarkeit von Zahlen

Das folgende Beispiel dient dazu festzustellen, wieviele Zahlen es im Intervall [0;1.000.000] gibt, die ganzzahlig durch 13 teilbar sind. Dabei werden verschiedene Ansätze der Berechnung und die dazu benötigten Zeiten dargestellt. In der Umgebung Anaconda kann ein solcher Benchmark übrigens sehr elegant über die Magic Function "%%timeit" durchgeführt werden.

### Serielle Programmierung

Die einfachste Form der seriellen Programmierung ist, den entsprechenden Zahlenbereich hochzuzählen und zu prüfen, ob die jeweilige Zahl ohne Rest durch 13 teilbar ist. Der Zähler i wird zur globalen Variable erklärt, damit man auch später noch auf ihn zugreifen kann.

In [1]:

```
import sys
```

In [2]:

```
%%timeit
global count
global i
count = 0
for i in range(1000000):
    if i % 13 == 0:
        count += 1
```

```
90.5 ms ± 1.84 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [3]:

```
print(count," Zahlen zwischen 0 und 1.000.000 sind ganzzahlig durch 13 teilbar.")
```

```
76924  Zahlen zwischen 0 und 1.000.000 sind ganzzahlig durch 13 teilbar.
```

Die serielle Programmierung unter Python erfordert bereits merklich an Zeit. Der Zähler i ist jeweils eine Integerzahl, er ist kein Array und keine Liste.

In [4]:

```
print("Der Zähler i belegte dabei folgenden Speicherplatz:", sys.getsizeof(i), "Bytes.")
```

```
Der Zähler i belegte dabei folgenden Speicherplatz: 28 Bytes.
```

## Serielle Programmierung in Numpy

Die serielle Programmierung in Numpy unterscheidet sich dadurch, dass alle zu prüfenden Zahlen in einem Array abgelegt werden und das Programm dann jedes Element des Arrays auf Teilbarkeit ohne Rest prüft.

In [5]:

```
import numpy as np
a = np.arange(1000000)
```

In [6]:

```
%%timeit
global count
global a
count = 0
for i in a:
    if i % 13 == 0:
        count += 1
```

```
503 ms ± 8.64 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [7]:

```
print(count," Zahlen zwischen 0 und 1.000.000 sind ganzzahlig durch 13 teilbar.")
```

```
76924  Zahlen zwischen 0 und 1.000.000 sind ganzzahlig durch 13 teilbar.
```

Die serielle Programmierung unter der Bibliothek Numpy erfodert sogar mehr Zeit pro Loop, als die reine Berechnung in Python. Grund hierfür ist, dass die Fähigkeiten zur parallelen Berechnung nicht aufgerufen bzw. ausgenutzt werden. Da die zu prüfenden Zahlen im Array a abgelegt werden, erfordert das Array a deutlich mehr Speicherplatz als die die Variable i in der vorherigen Variante.

In [8]:

```
print("Die geprüften Zahlen belegen als Array folgenden Speicherplatz:", sys.getsizeof(a),
```

```
Die geprüften Zahlen belegen als Array folgenden Speicherplatz: 4000096 Byte
s.
```

## Parallele Programmierung in Numpy

Hier werden auch alle zu prüfenden Zahlen in einem Array abgelegt, aber es werden dann alle Elemente des Arrays parallel und auf einmal daraufhin geprüft, ob sie restfrei durch 13 teilbar sind. Hieraus ergibt sich ein neues Array, welches genauso groß ist wie das Array a und an jeder Stelle den Wert True oder False hat, je nach dem, ob die Zahl in a an der gleichen Stelle restfrei durch 13 teilbar ist oder nicht. Da True dem Wert 1 und False dem Wert 0 entspricht, muss man im Anschluss nur noch die Summe der Elemente in dem Array bilden, das True oder False enthält.

In [9]:

```
import numpy as np
a = np.arange(1000000)
```

In [10]:

```
%%timeit
global count
global a
count = 0
count = np.sum(np.mod(a, 13) == 0)
```

```
8.52 ms ± 85.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [11]:

```
print(count," Zahlen zwischen 0 und 1.000.000 sind ganzzahlig durch 13 teilbar.")
```

```
76924  Zahlen zwischen 0 und 1.000.000 sind ganzzahlig durch 13 teilbar.
```

Man erkennt deutlich, wie die parallel verarbeiteten Vektor- / Arrayfunktionen das Ergebnis beschleunigen. Zum Berechnen nimmt man hier die Summenfunktion und die Modulofunktion aus der Numpy-Bibliothek.

In [12]:

```
print("Die geprüften Zahlen belegen als Array folgenden Speicherplatz:", sys.getsizeof(a),
```

```
Die geprüften Zahlen belegen als Array folgenden Speicherplatz: 4000096 Byte
s.
```

## Gemischte parallele Programmierung unter Python

Je nach Aufgabenstellung bietet bereits das Grundgerüst von Python ohne Bibliotheken genügend Funktionen, um Berechnungen parallel durchzuführen. Diese Erkenntnis kann dazu verleiten, Befehle aus dem Grundgerüst von Python, nämlich hier die Modulo-Funktion und die Summenfunktion, mit den Befehlen einer Bibliothek (hier Numpy) zu "vermischen".

In [13]:

```
import numpy as np
a = np.arange(1000000)
```

In [14]:

```
%%timeit
global count
global a
count = 0
b = ((a % 13)==0)
count = sum(b)
```

```
2.48 s ± 45.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [15]:

```
print(count," Zahlen zwischen 0 und 1.000.000 sind ganzzahlig durch 13 teilbar.")
```

```
76924  Zahlen zwischen 0 und 1.000.000 sind ganzzahlig durch 13 teilbar.
```

Man erkennt sehr deutlich, dass diese "Mischung" keinerlei Vorteile bringt, sondern - im Gegenteil sogar - merkliche zeitliche Nachteile in der Programmausführung. Eine Begründung hierfür ist, dass die Funktionen in Numpy unter dem Aspekt der Verarbeitbarkeit von Arrays und Listen, sowie unter dem Aspekt der Geschwindigkeit erstellt wurden und teilweise sehr prozessornah ablaufen.

In [16]:

```
print("Die geprüften Zahlen belegen als Array folgenden Speicherplatz:", sys.getsizeof(a),
```

```
Die geprüften Zahlen belegen als Array folgenden Speicherplatz: 4000096 Byte
s.
```

# Beispiel - ReLU-Funktion

Eine Funktion, die in bei der Erstellung neuronaler Netze als "Aktivierungsfunktion" sehr oft zum Einsatz kommt, ist die sogenannte ReLU-Funktion (ReLU = rectified linear unit; deutsch: gleichgerichtete lineare Einheitsfunktion). Sie ist definiert als f(x) = max(0,x), d.h. für alle x < 0 ergibt sich f(x) = 0 und für alle x >= 0 ergibt sich f(x) = x. Für die Berechnung dieser Funktion kann man verschiedene Ansätze wählen, die sich massiv in der Rechenzeit unterscheiden.

Zur besseren Vergleichbarkeit wird die Menge der x-Werte, aus denen dann f(x) berechnet werden soll, einmalig berechnet und für jede Variante zur Verfügung gestellt.

## Erzeugung der Zufallszahlen

Es werden 50 Dezimalzahlen im Wertebereich von -10 bis inkl. 10 erzeugt, die bis zu zwei Dezimalen besitzen können. Die Reihenfolge dieser Zahlen innerhalb des Arrays, in dem sie abgelegt werden, ist willkürlich.

In [17]:

```
import sys
import random as rnd
import numpy as np
import matplotlib.pyplot as plt
X = np.asarray(rnd.sample(range(-1000, 1001), 50)) / 100
```

In [18]:

```
print("Die erzeugten Zufallszahlen lauten:\n\n",X)
```

```
Die erzeugten Zufallszahlen lauten:

 [-0.71  4.56  8.74 -5.92 -7.75 -7.61  9.84 -6.64  2.75  2.63 -8.38 -4.15
 -1.33  9.34 -5.98  3.5  -9.66  3.36 -8.82  6.4   9.07  3.8   4.92 -0.16
 -4.25  1.7   9.94 -4.11 -0.1  -9.96  6.37 -3.06 -3.7   3.87  9.72 -1.48
  8.39 -9.81  5.61  1.52  1.56  9.82 -7.12 -7.76 -4.12 -0.01 -3.9  -1.84
  3.58  2.61]
```

In [19]:

```
print("Die erzeugten Zufallszahlen belegen als Array folgenden Speicherplatz:", sys.getsize
```

```
Die erzeugten Zufallszahlen belegen als Array folgenden Speicherplatz: 496 B
ytes.
```

## Serielle Programmierung

Die Funktion ReLU_seriell() erwartet, ein Array übergeben zu bekommen und arbeitet dann jedes einzelne Element dieses Arrays ab, bis das komplette Array durchlaufen ist. Ist das Element kleiner Null, so wird eine anfangs leere Liste y um eine Null ergänzt, andernfalls wird die Liste y um den Wert des betrachteten Elements ergänzt.

In [20]:

```
def ReLU_seriell(x):
    k = len(x)
    y = []
    for i in range(0,k):
        if (x[i] < 0):
            y.append(0)
        else:
            y.append(x[i])
    return y
```

In [21]:

```
%%timeit
global Y
Y = ReLU_seriell(X)
```

```
25.8 µs ± 1.38 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

In [22]:

```
print("Das Ergebnis lautet:\n",Y)
```

```
Das Ergebnis lautet:
 [0, 4.56, 8.74, 0, 0, 0, 9.84, 0, 2.75, 2.63, 0, 0, 0, 9.34, 0, 3.5, 0, 3.3
6, 0, 6.4, 9.07, 3.8, 4.92, 0, 0, 1.7, 9.94, 0, 0, 0, 6.37, 0, 0, 3.87, 9.7
2, 0, 8.39, 0, 5.61, 1.52, 1.56, 9.82, 0, 0, 0, 0, 0, 0, 3.58, 2.61]
```

In [23]:

```
print("Die erzeugten Funktionswerte der ReLU-Funktion belegen als Liste folgenden Speicherp
```

```
Die erzeugten Funktionswerte der ReLU-Funktion belegen als Liste folgenden S
peicherplatz: 528 Bytes.
```

## Serielle Programmierung in Numpy als Array

Bei der seriellen Programmierung in Numpy als Array wird zunächst ein Array y erzeugt, das nur Nullen enthält und genausoviele Elemente wie das übergebene Array x. Danach wird jedes Element in x geprüft; ist es kleiner als Null, wird an die gleiche Stelle in y eine Null geschrieben, andernfalls wird in y die betrachtete Zahl geschrieben.

In [24]:

```
def ReLU_seriell_array(x):
    k = len(x)
    y = np.zeros(k)
    for i in range(0,k):
        if (x[i] < 0):
            y[i] = 0
        else:
            y[i] = x[i]
    return y
```

In [25]:

```
%%timeit
global Y
Y = ReLU_seriell_array(X)
```

```
28.7 µs ± 334 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

In [26]:

```
print("Das Ergebnis lautet:\n",Y)
```

```
Das Ergebnis lautet:
 [0.   4.56 8.74 0.   0.   0.   9.84 0.   2.75 2.63 0.   0.   0.   9.34
 0.   3.5  0.   3.36 0.   6.4  9.07 3.8  4.92 0.   0.   1.7  9.94 0.
 0.   0.   6.37 0.   0.   3.87 9.72 0.   8.39 0.   5.61 1.52 1.56 9.82
 0.   0.   0.   0.   0.   0.   3.58 2.61]
```

In [27]:

```
print("Die erzeugten Funktionswerte der ReLU-Funktion belegen als Array folgenden Speicherp
```

```
Die erzeugten Funktionswerte der ReLU-Funktion belegen als Array folgenden S
peicherplatz: 496 Bytes.
```

## Serielle Programmierung in Numpy als Array - Variante 2

Bei der zweiten Variante der seriellen Programmierung in Numpy als Array wird zunächst ein Array y erzeugt, das nur Nullen enthält und genausoviele Elemente wie das übergebene Array x. Danach wird jedes Element in

x geprüft; ist es größer als Null, wird an die gleiche Stelle in y der Wert dieses Elementes geschrieben; andernfalls wird in y nichts geschrieben, denn der Wert der ReLU-Funktion ist dann Null, das Array y wurde aber bereits zu Beginn komplett mit Nullen gefüllt. Somit wird der Schreibvorgang in y gespart, wenn die Werte in x kleiner oder gleich Null sind.

In [28]:

```
def ReLU_seriell_array(x):
    k = len(x)
    y = np.zeros(k)
    for i in range(0,k):
        if (x[i] > 0):
            y[i] = x[i]
    return y
```

In [29]:

```
%%timeit
global Y
Y = ReLU_seriell_array(X)
```

```
25.1 µs ± 462 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

In [30]:

```
print("Das Ergebnis lautet:\n",Y)
```

```
Das Ergebnis lautet:
 [0.   4.56 8.74 0.   0.   0.   9.84 0.   2.75 2.63 0.   0.   0.   9.34
 0.   3.5  0.   3.36 0.   6.4  9.07 3.8  4.92 0.   0.   1.7  9.94 0.
 0.   0.   6.37 0.   0.   3.87 9.72 0.   8.39 0.   5.61 1.52 1.56 9.82
 0.   0.   0.   0.   0.   0.   3.58 2.61]
```

In [31]:

```
print("Die erzeugten Funktionswerte der ReLU-Funktion belegen als Array folgenden Speicherp
```

```
Die erzeugten Funktionswerte der ReLU-Funktion belegen als Array folgenden S
peicherplatz: 496 Bytes.
```

## Serielle Programmierung mit Boolschen Operatoren

Bei der seriellen Programmierung mit Boolschen Operatoren wird auch jedes Element des übergebenen Arrays x einzeln betrachtet und mit einem Boolschen Operator multipliziert. Dieser ist True, also gleich 1, wenn das einzelne Element größer oder gleich Null ist, andernfalls ist der Operator False, also gleich Null. Das Produkt des betrachteten Elementes mit dem Boolschen Operator wird einer anfangs leeren Liste angehängt.

Bemerkenswert ist hierbei, dass Python einen internen Rundungsfehler mit durchzieht und es daher zu Werten -0.0 kommen kann.

In [32]:

```
def ReLU_seriell_bool(x):
    k = len(x)
    y = []
    for i in range(0,k):
        y.append(x[i] * (x[i] >= 0))
    return y
```

In [33]:

```
%%timeit
global Y
Y = ReLU_seriell_bool(X)
```

```
157 µs ± 1.3 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

In [34]:

```
print("Das Ergebnis lautet:\n",Y)
```

```
Das Ergebnis lautet:
 [-0.0, 4.56, 8.74, -0.0, -0.0, -0.0, 9.84, -0.0, 2.75, 2.63, -0.0, -0.0, -
0.0, 9.34, -0.0, 3.5, -0.0, 3.36, -0.0, 6.4, 9.07, 3.8, 4.92, -0.0, -0.0, 1.
7, 9.94, -0.0, -0.0, -0.0, 6.37, -0.0, -0.0, 3.87, 9.72, -0.0, 8.39, -0.0,
5.61, 1.52, 1.56, 9.82, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0, 3.58, 2.61]
```

In [35]:

```
print("Die erzeugten Funktionswerte der ReLU-Funktion belegen als Liste folgenden Speicherp
```

```
Die erzeugten Funktionswerte der ReLU-Funktion belegen als Liste folgenden S
peicherplatz: 528 Bytes.
```

## Parallele Programmierung in Numpy als Vektor

Bei der parallelen Programmierung in Numpy als Vektor (Array) wird zunächst ein Array y mit Nullen erzeugt, das genausoviele Elemente enthält wie das übergebene Array x. Danach wird mittels eines parallelen Rechenzuges jedes Element aus x mit dem Boolschen Operator multipliziert, der True ist, wenn das Element größer oder gleich Null ist, und der andernfalls False ist; diese Ergebnisarray wird in das Array y geschrieben, dessen Speicher bereits dadurch reserviert war, dass man es mit Nullen gefüllt hatte.

In [36]:

```
def ReLU_parallel(x):
    k = len(x)
    y = np.zeros(k)
    y = (x >= 0) * x
    return y
```

In [37]:

```
%%timeit
global Y
Y = ReLU_parallel(X)
```

```
3.65 µs ± 41.9 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

In [38]:

```
print("Das Ergebnis lautet:\n",Y)
```

```
Das Ergebnis lautet:
 [-0.    4.56  8.74 -0.   -0.   -0.    9.84 -0.    2.75  2.63 -0.   -0.
 -0.    9.34 -0.    3.5  -0.    3.36 -0.    6.4   9.07  3.8   4.92 -0.
 -0.    1.7   9.94 -0.   -0.   -0.    6.37 -0.   -0.    3.87  9.72 -0.
  8.39 -0.    5.61  1.52  1.56  9.82 -0.   -0.   -0.   -0.   -0.   -0.
  3.58  2.61]
```

In [39]:

```
print("Die erzeugten Funktionswerte der ReLU-Funktion belegen als Array folgenden Speicherp
```

```
Die erzeugten Funktionswerte der ReLU-Funktion belegen als Array folgenden S
peicherplatz: 496 Bytes.
```

## Parallele Programmierung in Numpy als Vektor ohne Speicherreservierung

Bei der parallelen Programmierung in Numpy als Vektor (Array) ohne Speicherreservierung wird fast identisch wie in der Berechnung zuvor vorgegangen, jedoch wird zu Beginn keine Matrix y erzeugt, die nur Nullen enthält. Damit muss Python nach dem parallelen Rechnen einen Speicherplatz finden, wo die Ergebnismatrix y hineinpasst.

In [40]:

```
def ReLU_parallel_ohne_zeros(x):
    y = (x >= 0) * x
    return y
```

In [41]:

```
%%timeit
global Y
Y = ReLU_parallel_ohne_zeros(X)
```

```
2.58 µs ± 28.6 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

In [42]:

```
print("Das Ergebnis lautet:\n",Y)
```

```
Das Ergebnis lautet:
 [-0.    4.56  8.74 -0.   -0.   -0.    9.84 -0.    2.75  2.63 -0.   -0.
 -0.    9.34 -0.    3.5  -0.    3.36 -0.    6.4   9.07  3.8   4.92 -0.
 -0.    1.7   9.94 -0.   -0.   -0.    6.37 -0.   -0.    3.87  9.72 -0.
  8.39 -0.    5.61  1.52  1.56  9.82 -0.   -0.   -0.   -0.   -0.   -0.
  3.58  2.61]
```

In [43]:

```
print("Die erzeugten Funktionswerte der ReLU-Funktion belegen als Array folgenden Speicherp
```

```
Die erzeugten Funktionswerte der ReLU-Funktion belegen als Array folgenden S
peicherplatz: 496 Bytes.
```

## Grafische Ausgabe der ReLU-Funktion

In [44]:

```
x2 = np.linspace(-12,12,240)
y2 = ReLU_seriell(x2)
```

In [45]:

```
plt.figure(0,figsize = (8,8))
plt.title("Verlauf der ReLU-Funktion", size = 16)
plt.plot(x2,y2,linestyle = "dashed", color = "orange", label = "Funktionsgraph")
plt.scatter(X,Y, label = "Datenpunkte")
plt.grid(True)
plt.legend(loc = "upper left", fontsize = 14)
plt.show()
```