Einführung in Cython

Teil 1

Jonathan Helgert
jhelgert@mail.uni-mannheim.de

https://github.com/jhelgert/cython_einfuehrung

19. November 2020

Contents

Einstiegsbeispiel

Wir suchen die Anzahl an Pythagorastripeln, d.h. wie viele $(a, b, c) \in \mathbb{Z}^3$ mit $a^2 + b^2 = c^2$ und $0 < a < b < c \leq N$ gibt es für ein gegebenes N?

```cpp
// -- C & C++ --
int count_triples(const int N){
    int found = 0;
    for(int a = 1; a <= N; ++a){
        for(int b = a+1; b <= N; ++b){
            for(int c = b+1; c <= N; ++c){
                if(a*a + b*b == c*c){
                    found++;
                }
            }
        }
    }
    return found;
}
// N = 10 000, Timing: 12.2s
```

```python
# -- Python --
def count_triples(N):
    found = 0
    for a in range(1, N+1):
        for b in range(a+1, N+1):
            for c in range(b+1, N+1):
                if a*a + b*b == c*c:
                    found += 1
    return found

# N = 10 000, Timing: 21045s = 5h 50min
```

Takeaway 1. *Pythonschleifen sind im Vergleich zu low-level Sprachen (C, C++, Fortran) sehr langsam.*

Einstiegsbeispiel

Und mit numpy?

```python
# -- Python --
import numpy as np

def count_triples(N):
    a = np.arange(1, N+1, dtype=np.int32)
    b = np.arange(1, N+1, dtype=np.int32)
    c = np.arange(1, N+1, dtype=np.int32)
    ab = a[:, None]**2 + b[None, :]**2
    ab_updiag = ab[np.triu_indices_from(ab)]
    return np.sum(np.isin(ab_updiag, c**2))

# N = 10 000, Timing: 13.1s
```

Motivation

> *"What makes Python fast for development is what makes Python slow for code execution"*
>
> - Jake VanderPlas.

Python in a nutshell:

- High-Level Sprache: Programm wird von einem Interpreter (verbreitester: CPython) ausgeführt.
- Dynamisches Typsystem: Typ einer Variablen wird erst bei der Verwendung überprüft.
- *Dynamic Dispatching* für jede Operation und Funktionsaufruf.
- Umfangreiche Fehlerkontrolle.

Aber numpy, scipy, tensorflow etc. sind nicht langsam?

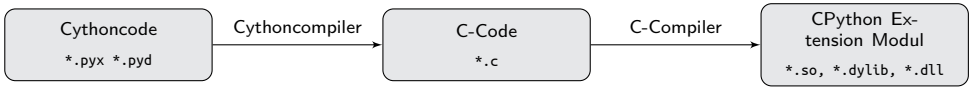
- Laufzeitkritischer Code wird in C/C++/Fortran geschrieben und eingebunden.
- Oder direkt in Cython geschrieben.

Einige Projekte, welche Cython verwenden: Sage, Scipy, Numpy, Pandas, scikit-learn, scikit-image, lxml, mpi4py, petsc4py, cyipopt, ...

Was ist Cython?

"Cython gives you the combined power of Python and C."
- cython.org

- Entwickler: Robert Bradshaw, Stefan Behnel, et al. Erscheinungsjahr: 2007.
- Open Source: `https://github.com/cython/cython`. Aktuellste Version 0.29.21. Neue Majorversion 3.x zu 92 % fertig.
- Eine Programmiersprache, die Python um das statische Typsystem von C und C++ erweitert.
- Ein "Compiler", der Cythonquellcode in effizienten C Code umwandelt, der wiederum in ein CPython Extension Modul kompiliert wird.
- Auch unveränderter Pythoncode kann vom Cythoncompiler kompiliert werden.



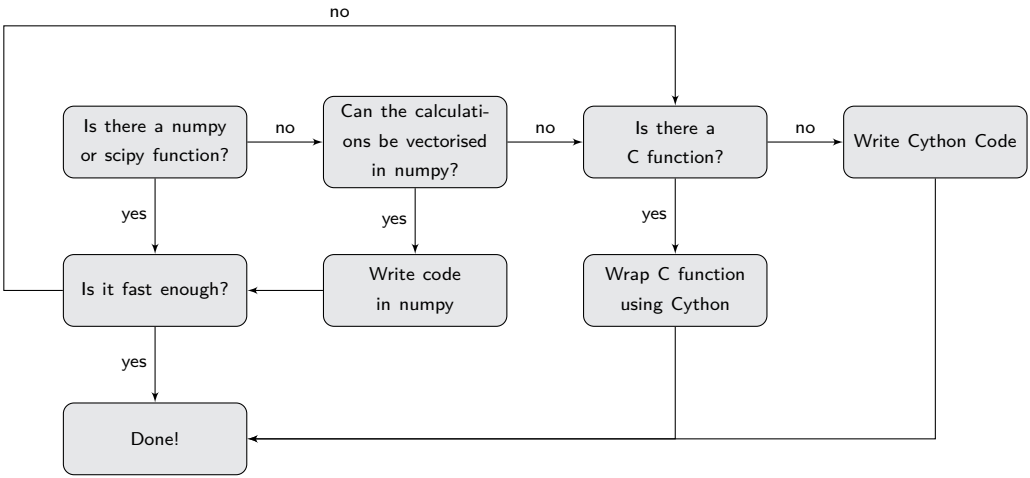
Anwendungsfälle:

1. Performanceoptimierung von Pythoncode.
2. *Write C without to have write C.*
3. *Wrapping* von C oder C++.

Python, Cython oder C?

"Premature optimization is the root of all evil."

- Donald Knuth.



Quelle: https://bit.ly/3hByQfr

Contents

Wie Code compilieren und ausführen?

Möglichkeit 1: Jupyter Notebook

- Innerhalb eines Jupyternotebooks `%load_ext cython` um bequem innerhalb eines Notebooks Cython verwenden zu können.
- Nun kann man das *Magiccommand* `%%cython` innerhalb eines Codebocks verwenden. Die im Codeblock enthaltenen Funktionen sind anschließend automatisch im aktuellen Notebook importiert.
- Tipp: Sehr nützlich ist die *annotate* option, d.h. `%%cython -a`.

Möglichkeit 2: Eigenes Package (Teil 2)

- Schreibe eine `helloworld.pyx` Datei mit dem Cythoncode.
- Schreibe eine `setup.py` Datei und nutze `setuptools`..
- Das Paket via `python3 setup.py build_ext --inplace` lokal im selben Verzeichnis installieren.
- Das Modul kann nun innerhalb des gleichen Verzeichnisses importiert werden.

Contents

Datentypen

Python besitzt nur die beiden reelen Zahlentypen `int` und `float`. Cython unterstützt alle C-Typen:

	Größe
`[unsigned] char`	8 Bit (1 Byte)
`[unsigned] short`	16 Bit (2 Byte)
`[unsigned] int`	32 Bit (4 Byte)
`[unsigned] long`	64 Bit (8 Byte)
`[unsigned] long long`	64 Bit (8 Byte)

Tabelle: Größe der ganzzahligen Datentypen.

	Größe
`float`	32 Bit (4 Byte)
`double`	64 Bit (8 Byte)
`long double`	min. 80 Bit (10 Byte)

Tabelle: Größe der Gleitkomma-Datentypen.

- Wertebereich für ganzzahligen Datentypen $[0, 2^n - 1]$ für vorzeichenlose (unsigned) bzw. $[-2^{n-1}, 2^{n-1} - 1]$ für vorzeichenbehaftet, wobei n die Anzahl der Bits ist.
- In Python sind `int`s unbeschränkt! Pythons `float` entspricht `double` in C.
- Der boolsche C-Datentyp `bool` heißt in Cython `bint`.

Takeaway 2. *Die C-Typen sind deutlich schneller, haben allerdings auch C-Semantik (Overflow bei ganzzahligen Datentypen und unterschiedliche Größe je nach Plattform)*

Datentypen

- Reguläre Pythonvariablen haben dynamischen Typ und können genau so in Cython verwendet werden.
- In Cython haben diese den Typ `object`.
- Mit dem Schlüsselwort `cdef` können wir innerhalb von Cython Variablen auf C-Level deklarieren.

```
# -- Cython --
a = 2                           # ein Python-int (unbeschränkt)
b = 2**31                       # = 2147483648
cdef int c = 2                  # ein C-int
cdef int d = 2**31              # = -2147483648, da Overflow :(
cdef long e = 2**31             # = 2147483648
cdef unsigned int g = 2**31     # = 2147483648

x = 0.1                         # Python float (64 Bit)
cdef float y = 0.1              # C float (32 Bit)
cdef double z = 0.1             # C double (64 Bit)
cdef long double q = 0.1*0.1    # C long double (min. 80 Bits)

cdef bint f = True              # boolscher Typ
```

Datencontainer

- Cython bietet auch Pythons Datencontainer `list`, `dict`, `tuple` für statisches Typing an.
- Zu Pythons Tuple gibt es ein effizientes `ctuple`. (Intern wird eine passende struct in C angelegt)
- Zudem können die C++ STL Container `std::deque`, `std::list`, `std::map`, `std::pair`, `std::queue`, `std::set`, `std::stack` und `std::vector` via `libcpp` benutzt werden (Teil2).

```
# -- Cython --
cdef (int, int) tup1                # tup1 ist ein ctuple bestehend aus zwei ints.
cdef (int, double) tup2 = (1, 2.0)  # ctuple aus int und double
cdef list l1 = ["hi", 2, 3.0]       # eine liste
cdef dict d1 = {"key" : 2}          # ein dictionary
cdef int[3] carr = [1, 2, 3]        # ein C-Array (andere Syntax als in C!)
```

Funktionen: def, cdef, cpdef

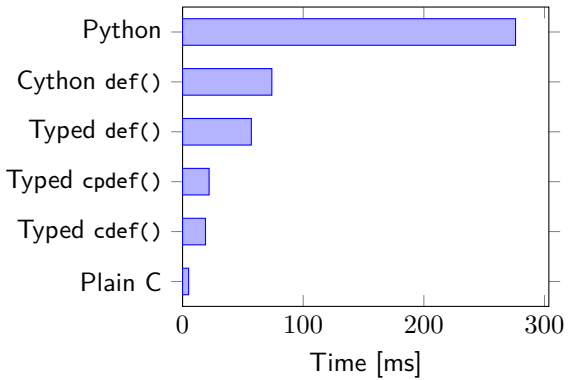
Es gibt in Cython drei mögliche Schlüsselwörter um eine Funktion zu definieren: `def`, `cdef` und `cpdef`.

- `def`: Normale Pythonfunktion.
- `cdef`: C-Funktion. Parameter und Rückgabewert sind C-Variablen oder Pythonobjekte. Vorteil: Kein Overhead. Nachteil: Nicht so flexibel wie Pythonfunktionen, können nicht außerhalb eines Cythonmoduls aufgerufen werden.
- `cpdef`: Es wird automatisch eine `cdef` und `def` Funktion erstellt. Kann von überall aufgerufen werden. Innerhalb von Cython wird immer die C Funktion aufgerufen.

```
def func1(x):           # Kann innerhalb von Python aufgerufen werden.
    cdef int y = x
    return y*y

cdef int func2(int x):  # Kann NICHT innerhalb von Python aufgerufen werden.
    return x*x

def func3(int x):       # Cython erstellt automatisch func1
    return x*x

cpdef func4(int x):     # Cython erstellt automatisch func1 und func2
    return x*x
```

Wie schnell sind `def`, `cdef`, bzw. `cpdef`?

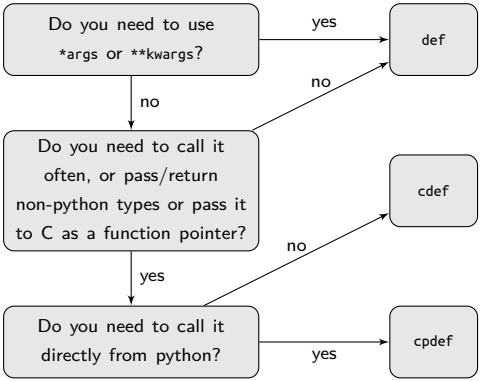
Trivialer Benchmark: Rekursive Implementierung der Fibonaccifolge. Aufruf `fib(30)`:



Takeaway 3. *Pythonfunktionen haben großen Overhead.*

```python
# -- Python / Cython --
def fib(N):
    if N < 2:
        return N
    return fib(N-2) + fib(N-1)

# -- Cython --
def int fib(int N):
    if N < 2:
        return N
    return fib(N-2) + fib(N-1)

# -- Cython --
cpdef int fib(int N):
    if N < 2:
        return N
    return fib(N-2) + fib(N-1)
```

Wann def, cdef oder cpdef?



Quelle: https://bit.ly/334YioK

Takeaway 4. *Sobald eine Funktion aufgerufen wurde, besteht für den Code innerhalb der Funktion kein Performanceunterschied zwische* `def`, `cdef` *und* `cpdef`.

Fused Types

Angelehnt an C++-Templates, bietet Cython *fused types* für Funktionsparameter.

- Das Erstellen der passenden Funktion (*Dispatching*) findet zur Compilezeit statt, nicht zur Laufzeit.
- Einschränkung: Wird derselbe Fused Type mehrmals in den Funktionsargumenten verwendet, müssen diese Argumente auch denselben Typ haben.
- Cython bietet drei vordefinierte Fused Types: `integral`, `floating` und `numeric`.

```
# -- Cython --
# Deklarationen von fused types
ctypedef fused T1:
    int
    double

ctypedef fused T2:
    T1
    float

from cython cimport floating
```

```
# Fused Types für Funktionsargumente:
cdef f(T1 arg1, T1 arg2, T2 arg3):
    # do something

# Auch für Memoryviews
def g(floating[:, ::1] A, floating[::1] b):
    # ...

f(1, 1.0, 1.0) # kompiliert nicht!
f(1, 1, 1.0)   # f(int, int, double)
f(1.0, 1.0, 1) # f(double, double, int)
```

Takeaway 5. *Verwende Fused Types für generischen Cythoncode.*

cimport vs import

Mit Hilfe von `cimport` können wir C-Funktionen, C-structs, C-variablen und Extension Types einbinden.

- Ein `cimport` bindet das Modul und die darin enthaltenen Funktion über dessen Cythonschnittstelle auf C-Level ein statt auf Pythonlevel.
- Es gibt inzwischen immer mehr Libraries, die Cythonmodule anbieten.
- Vorteil: Diese Funktionen haben deutlich weniger Overhead.

```
import numpy as np   # Benutzt numpy über Python
cimport numpy as np  # Benutzt direkt die C-API für numpy
from scipy.linalg.cython_blas cimport dgemv     # Matrixvektorprodukt
from scipy.linalg.cython_lapack cimport dsgesv # LGS-Loeser
from scipy.linalg.cython_optimize cimport bisect # Nullstelle via bisektion
from scipy.special.cython_special cimport gamma # Gammafunktion
```

cimport vs import

Beispiel: In C gibt es standardisierte ganzzahlige Typen innerhalb von `stdint.h`. Diese stehen in Cython innerhalb von `libc.stdint` zur Verfügung.

```
# -- Cython --
from libc.stdint cimport *

cdef int8_t   a = 1 #   signed char,  8 Bit
cdef int16_t  b = 1 #   signed short, 16 Bit
cdef int32_t  c = 1 #   signed int, 32 Bit
cdef int64_t  d = 1 #   signed long long, 64 Bit
cdef uint8_t  e = 1 # unsigned char,  8 Bit
cdef uint16_t f = 1 # unsigned short, 16 Bit
cdef uint32_t g = 1 # unsigned int, 32 Bit
cdef uint64_t h = 1 # unsigned long long, 64 Bit
```

Takeaway 6. *Verwende die standardisierten ganzzahligen Typen aus `libc.stdint`.*

Contents

Code Profiling via Line Profiler

Jetzt: Wie können wir genau herausfinden, wie viel Zeit welche Abschnitte unseres Codes in Anspruch nehmen?

- Die Syntax innerhalb eines Notebooks ist

  ```
  %lprun -f name_der_funktion name_der_funktion(*args)
  ```

- Um Cythoncode *profilen* zu können, müssen wir ihn mit entsprechenden Directives und Flags kompilieren.
- Innerhalb eines Notebooks muss dazu

  ```
  # cython: linetrace=True, binding=True
  # distutils: define_macros=CYTHON_TRACE_NOGIL=1
  ```

  in die obersten Zeilen der entsprechende Zelle mit Cythoncode.

Takeaway 7. *Kompilieren mit Profilingflags verlangsamt den Code deutlich. Daher niemals für Endcode verwenden.*

Zurück zum Einstiegsbeispiel

Wir suchen die Anzahl an Pythagorastripeln, d.h. wie viele $(a, b, c) \in \mathbb{Z}^3$ mit $a^2 + b^2 = c^2$ und $0 < a < b < c \leq N$ gibt es für ein gegebenes N?

```python
# -- Python --
def count_triples(N):
    found = 0
    for a in range(1, N+1):
        for b in range(a+1, N+1):
            for c in range(b+1, N+1):
                if a*a + b*b == c*c:
                    found += 1
    return found
```

```python
# -- Cython --
def count_triples(int N):
    cdef int a, b, c
    cdef int found = 0
    for a in range(1, N+1):
        for b in range(a+1, N+1):
            for c in range(b+1, N+1):
                if a*a + b*b == c*c:
                    found += 1
    return found
```

Contents

Cython, Numpy und Typed Memoryviews

Ein *Typed Memoryview* ermöglicht uns effizienten Zugriff auf Datenbuffer.

- Unterstützt unter anderem `np.arrays`, `cython.arrays` und C-Arrays.
- Indizieren, Kopieren, Transponieren und Broadcasting wird unterstützt.
- Beinhaltet zudem einige Attribute mit Informationen über den Datenbuffer, z.B. `shape`, `size`, `nbytes` oder `base`.

Wir beschränken uns auf Memoryviews auf `np.arrays`:

```
# -- Cython --
cdef int[:] a = np.arange(20, dtype=np.int32)                # a.shape: (20,)
cdef float[:, :] A = np.random.rand(10, 9, dtype=np.float32) # A.Shape: (10, 9)
cdef double[:, :, :] C = np.random.rand(10, 3, 5)            # C.Shape: (10, 3, 5)
```

Takeaway 9. *Verwende Typed Memoryviews für effizienten Zugriff auf `np.arrays`.*

Takeaway 9. *Schleifen über `np.arrays` innerhalb Python möglichst vermeiden.*

Cython, Numpy und Typed Memoryviews

Numpy Typ	C stdint	C Typ
`np.int8`	`int8_t`	`char`
`np.int16`	`int16_t`	`short`
`np.int32`	`int32_t`	`int`
`np.int64`	`int64_t`	`long [long]`
`np.uint8`	`uint8_t`	`unsigned char`
`np.uint16`	`uint16_t`	`unsigned short`
`np.uint32`	`uint32_t`	`unsigned int`
`np.uint64`	`uint64_t`	`unsigned long [long]`
`np.float32`		`float`
`np.float64`		`double`

Tabelle: Übersicht der reelen Numpy `dtypes`.

- Für ganzzahlige Arrays ist der Defaulttyp `np.int64`, für Arrays mit Gleitkommazahlen dagegen `np.float64`.

Takeaway 10. *Verwende für `np.arrays` den richtigen `dtype`. Tipp: `.astype()` Methode für Umwandlungen.*

Cython, Numpy und Typed Memoryviews

```
cdef double[:, :] mv1 = np.random.rand(10, 20)
cdef double[:, :] y1, y2, y3, y4

# Zuweisungen und Kopien:
y1 = mv1              # ACHTUNG: y ist ein View auf mv1 (d.h. teilen sich den Datenbuffer)
y2[:, :] = mv1        # Kopiert Werte von mv1 in y
y3[:, :] = mv1.copy() # Kopiert ebenfalls
y4[:, :] = 1.0        # funktioniert auch für Skalare

# Transponieren
cdef double[:, :] mv2 = mv1.T # Transponierte, Achtung: View (genau wie in numpy)

# Slicing über einen Index/eine Dimension via start:end:step
cdef double[:] mv3 = mv1[:, 0]         # View auf Spalte 0 von mv1
cdef double[:] mv4 = mv1[:, 0].copy()  # Kopie der Spalte 0 von mv1
cdef double[:] mv5 = mv1[3, 0:20:2]    # View auf jedes 2. Element aus Zeile 3

# Broadcasting
cdef double[:] mv6 = np.ones((10, ))
cdef double[:, :] mv7 = mv6[None, :].copy()  # shape (1, 10)
```

Cython, Numpy und Typed Memoryviews

- **Achtung**: Ein Memoryview ist kein np.array. Man muss es also ausdrücklich in ein `np.array` umwandeln bei der Rückgabe an Python.
- Die Funktion `np.asarray` erkennt dann automatisch `shape` und Datentyp des Memoryviews.

```
# -- Cython --
cimport numpy as np
import numpy as np

def cy_ones(N):
    cdef double[:] B = np.empty(N)
    B[:] = 1.0
    return np.asarray(B)  # <--- Wichtig, shape (N,)

def cy_one_matrix(N, N):
    cdef double[:, :] B = np.empty((N, N))
    B[:, :] = 1.0
    return np.asarray(B)  # <--- Wichtig, shape (N, N)
```

Memorylayout

Arrays werden in C/C++ zeilenweise ohne Lücken gespeichert. In anderen Sprachen wie Fortran, Julia oder Matlab dagegen spaltenweise. Anschaulich:

$$A_{\text{row-major}} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}, \qquad B_{\text{column-major}} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Im Speicher sieht das jeweils so aus:

$$A_{\text{row-major}} : \ldots \; 1 \; 2 \; 3 \; 4 \; 5 \; 6 \; 7 \; 8 \; 9 \; \ldots$$
$$B_{\text{column-major}} : \ldots \; 1 \; 4 \; 7 \; 2 \; 5 \; 8 \; 3 \; 6 \; 9 \; \ldots$$

Setzen wir `::1` bei der Deklaration eines Memoryviews am letzten Index, ist das Memorylayout des Memoryviews row-major. Setzen wir es am ersten Index, ist es column-major:

```
# -- Cython --
cdef long[:, ::1] A = np.array(np.arange(1, 10).reshape(3,3))            # row-major (default)
cdef long[::1, :] B = np.array(np.arange(1, 10).reshape(3,3), order="F") # column-major
```

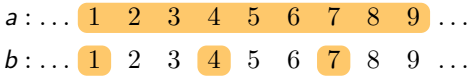
Takeaway 11. *Die Daten in einem np.array sind standardmäßig zeilenweise gespeichert.*

Memorylayout

Achtung: ein np.array muss nicht zwingend lückenlos als C-contiguous gespeichert sein:

```
# -- Cython --
cdef long[::1] a = np.arange(1, 10) # row- & columnmajor
cdef long[::1] b = a[0:10:3]        # Slicing: jedes dritte Element von a, KOMPILIERT NICHT!
```

Im Speicher:



Das array `b` ist ein View auf `a` und offensichtlich nicht c-contiguous. Also:

```
# -- Cython --
cdef long[:] b = a[0:10:3]
# oder:
cdef long[::1] b = np.ascontiguousarray(a[0:10:3])
```

Memorylayout

- Nützliche Funktionen: `np.ascontiguousarray` und `np.asfortranarray`.
- Das `.flags` Attribut liefert Infos über ein np.array

```
# .flags eines np.array
C_CONTIGUOUS : True        # row-major?
F_CONTIGUOUS : False       # column-major?
OWNDATA : True             # View ja/nein ?
WRITEABLE : True           # Schreibrechte
ALIGNED : True             # Memory alignment
WRITEBACKIFCOPY : False    # ...
UPDATEIFCOPY : False       # ...
```

Takeaway 12. *Bei der Deklaration das Memorylayout möglichst angeben und Speicher immer in der angelegten Reihenfolge durchgehen (Cache-friendly Code).*

Contents

Compilerdirectives

	Directive	default-Wert
Prüfen der Indices?	`boundscheck`	`True`
Negative Indices erlauben?	`wraparound`	`True`
C-Division verwenden?	`cdivision`	`False`
Prüft bei Zugriff auf Memoryviews, ob diese initialisiert sind	`initializedcheck`	`True`
Gibt einen Fehler zurück bei Integer-Overflow	`overflowcheck`	`False`

Tabelle: Nützliche Directives für den Cythoncompiler. Mehr unter: `https://bit.ly/3luL7EK`

Innerhalb eines Notebooks können die Directives für eine komplette Zelle via

```
# cython: directive1=value, directive2=value, directive3=value
```

gesetzt werden.

Compilerflags

	gcc & clang*	MVSC
Aktiviert Codeoptimierungen	`-O3`	`/O2`
Verwenden aller verfügbaren CPU-Features der aktuellen Maschine	`-march=native`	`/arch:AVX2` `/arch:AVX512`
Beschleunigung von Gleitkommaoperationen	`-ffast-math`	`/fp:fast`
Feedback bzgl. SIMD-Autovektorisierung	`-fopt-info-vec-optimized` `-Rpass=loop-vectorize`*	`/Qvec-report:1`
Für Parallelisierung via OpenMP	`-fopenmp`	`/openmp`

Tabelle: Nützliche Compilerflags für die drei gängigsten C-Compiler.

Die Flags werden innerhalb einer Notebookzelle via `%%cython -c=flag1 -c=flag2` gesetzt und dann automatisch an den C-Compiler übergeben.

Global Interpreter Lock

- CPythons *Garbage Collector* kümmert sich automatisch um das Speichermanagement.
- Der *Global Interpreter Lock* stellt sicher, dass stets nur ein Thread den Referenzzähler des Garbage Collectors verändern kann.
- In Cython können wir den GIL mit einem `with nogil:`-Block deaktivieren.
- Cythonfunktionen, die den GIL nicht benötigen, können wir mit `nogil` deklarieren.

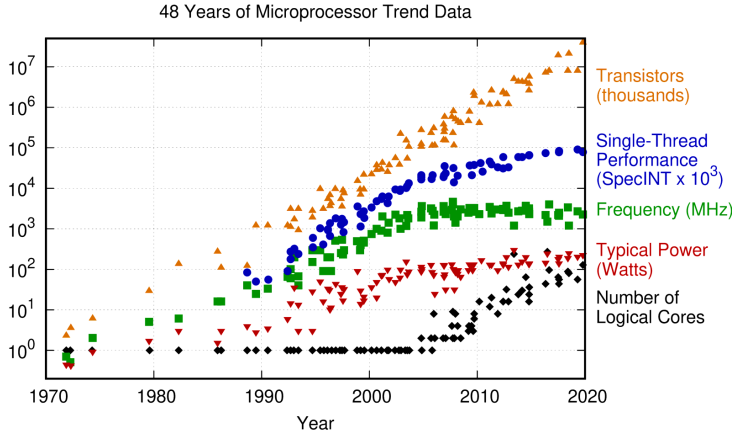
```
cdef winnetou() nogil: # Diese Funktion kann ohne den GIL verwendet werden
    # ...body...

cdef our_func():
    # ...do something in python...
    with nogil:
        winnetou()     # GIL-freier Block. Hier sind keine Pythonoperationen mehr möglich!
    # ...do something in python...
```

Takeaway 13. *Deklariere Cythonfunktionen, welche keine Pythonobjekte oder -operationen verwenden, mit* `nogil`.

Parallelisierung

"AMD Ryzen Threadripper 3990X: Konkurrenzlos an die Spitze. Mit dem 64-Kerner dringt AMD in neue Leistungssphären vor, sofern die Software damit zurecht kommt."

- heise.de 02/2020



Quelle: https://www.karlrupp.net/

Parallelisierung

Cython unterstützt threadbasierte Parallelisierung via OpenMP.

- Cython bietet die `prange` Funktion für parallele Schleifen an, welche automatisch via OpenMP Threads startet.
- Die Anzahl der Threads ist standardmäßig die Anzahl der verfügbaren CPU-Cores.
- Thread-Lokalität und `reductions` für Variablen werden automatisch erkannt.
- Der Code muss mit OpenMP-Compilerflag kompiliert und gelinkt werden.

```cython
# -- Cython --
from cython.parallel cimport prange

# Möglichkeit 1
cdef func1(int N):
    cdef int a
    with nogil:
        for a in prange(1, N):
            # do something (parallel)
```

```cython
# -- Cython --
from cython.parallel cimport prange

# Möglichkeit 2
cdef func2(int N):
    cdef int a
    for a in prange(1, N, nogil=True):
        # do something (parallel)
```

Takeaway 14. *Für parallelen Cythoncode muss der GIL deaktiviert werden.*

Parallelisierung

Wir können via `schedule` und `chunksize` steuern, wie die Arbeit unter den Threads aufgeteilt wird:

schedule	default chunksize	Overhead	Anwendungskriterium
`static`	$\frac{N}{P}$	gering	Das Problem lässt sich in ungefähr gleich große Chunks mit gleicher Laufzeit aufteilen.
`dynamic`	$\frac{N}{P}$	mittel	Falls Laufzeit und Größe der einzelnen Chunks unterschiedlich und im Voraus nicht bekannt ist

Tabelle: N Anzahl der Gesamtiterationen, P Anzahl der Threads.

Eine anschauliche Erklärung unter: `http://ppc.cs.aalto.fi/ch3/schedule/`.

```
# -- Cython --
for i in prange(N, nogil=True, schedule='static', chunksize=1):
    # ...
```

Takeaway 15. *Verwende für `prange` geeignetes scheduling und chunksize, um die Arbeit optimal unter den Threads aufzuteilen.*

Contents

Casestudy: Warmup

Gegeben ist folgender Cythoncode. Was bremst den Code aus? Was könnte man besser machen?

```
# image ist ein np.array mit shape (nx, ny, 3) und dtype=np.float32
def cython_color2gray(image):
    cdef int x, y, z
    cdef double grey
    for x in range(len(image)):
        for y in range(len(image[x])):
            for z in range(len(image[x][y])):
                if z == 0:
                    grey = image[x][y][0] * 0.21
                elif z == 1:
                    grey += image[x][y][1] * 0.07
                elif z == 2:
                    grey += image[x][y][2] * 0.72
            image[x][y][0] = grey
            image[x][y][1] = grey
            image[x][y][2] = grey
    return image
```

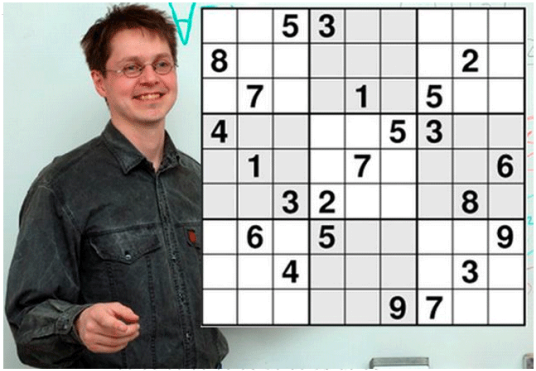
Casestudy: Solving world's hardest sudoku

The Telegraph

HOME » NEWS » SCIENCE » SCIENCE NEWS

World's hardest sudoku: can you crack it?

Readers who spend hours grappling in vain with the Telegraph's daily sudoku puzzles should look away now.



The Everest of numerical games was devised by Arto Inkala, a Finnish mathematician, and is specifically designed to be unsolvable to all but the sharpest minds.

```python
# -- Python --
def solve(puzzle):
    row, col = find_empty_cell(puzzle)
    if row == -1 and col == -1:
        return True

    for num in range(1,10):
        if valid(puzzle, num, (row, col)):
            puzzle[row, col] = num
            if solve(puzzle):
                return True
            else:
                puzzle[row, col] = 0
    return False
```

Quelle: https://bit.ly/32rBeAo

Aufgabe: Beschleunige den gegebenen Pythoncode mit Hilfe von Cython.

Ausblick

Cython bietet noch einiges mehr:

- C-Strings und C++-Strings.
- C-structs und C-unions.
- Klassen bzw. *Extension Types*.
- Pythran als numpy backend.
- Großteil der C++ STL.
- Support für C++.