Optimierung typischer Bildverarbeitungs-aufgaben mit Python

Benjamin Seppke
21.10.2013

Inhalt

- Motivation
- Wechsel der Software
  - Anderer Interpreter
  - Andere Programmiersprache
- Wechsel der Rechnerarchitektur
  - Threading
  - GPU
- Zusammenfassung

Inhalt

- Motivation
- Wechsel der Software
  - Anderer Interpreter
  - Andere Programmiersprache
- Wechsel der Rechnerarchitektur
  - Threading
  - GPU
- Zusammenfassung

Motivation (I)

- Wir benutzen bisher CPython
- Geschrieben in C – Interpretierende Sprache
- Geschwindigkeitsvorteil von C → NumPy für numerische, SciPy für weitere Algorithmen…
- Was, wenn sich der Algorithmus nicht mit „Bordmitteln“ lösen lässt?
- Elementar „nac programmieren“: for, if etc.

Motivation (II)

- Beispiel: `b = array + 1`
- Numpy: `b = array + 1`
- Ohne Numpy(3D-Array):

Timings für 256x256x256 Array:

177.223 ms

```
b = np.zeros((d,h,w))
for z in range(d):
  for y in range(h):
    for x in range(w):
      b[x,y,z] = a[x,y,z] + 1
```

48758.672 ms

Ca. 275-facher Geschwindigkeitsverlust!

Ziel des Vortrags

- Zurückholen der Performance (Numpy für eigene Algorithmen näher kommen)
- Wenig Komplexität in die Performance investieren
- Vorhandene Schnittstellen nutzen
- Vorschläge für eine Basis für weitere Optimierungen sammeln

Inhalt

- Motivation
- Wechsel der Software
  - Anderer Interpreter
  - Andere Programmiersprache
- Wechsel der Rechnerarchitektur
  - Threading
  - GPU
- Zusammenfassung

Software-Wechsel

- Problem an der Wurzel packen: dem Interpreter
- Zwei Möglichkeiten:
  - Anderen Interpreter verwenden
  - Interpreter beibehalten, aber problematisches Code-Fragment auslagern
- Klassischer Weg: CPython behalten und erweitern
- Durch neue Methoden (JIT etc.) auch neue Interpreter verfügbar!

Der PyPy-Interpreter (I)

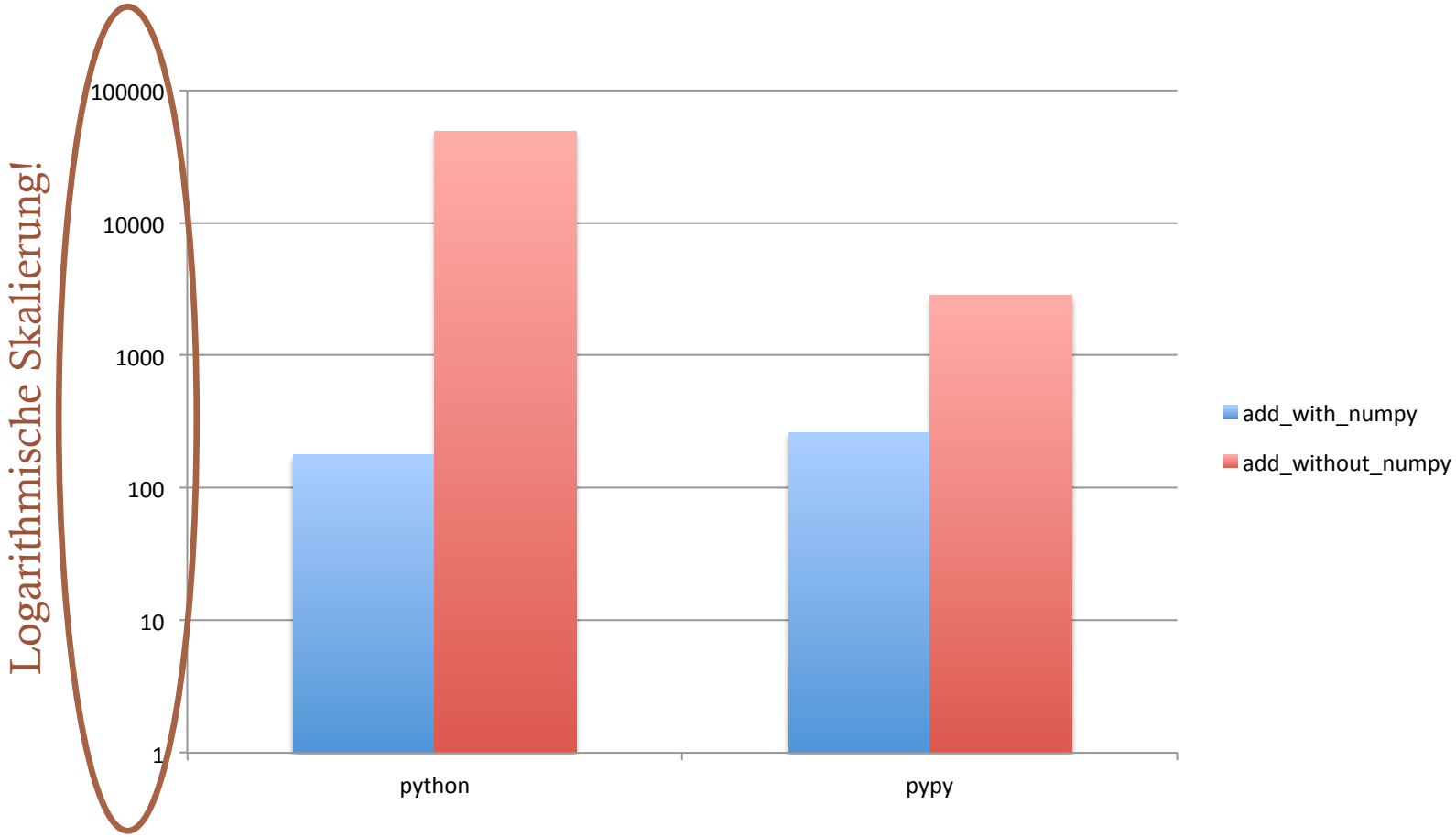
http://pypy.org/

- PyPy is a fast, compliant alternative implementation of the Python language (2.7.3 and 3.2.3). It has several advantages and distinct features:
- **Speed:** thanks to its Just-in-Time compiler, Python programs often run faster on PyPy. (What is a JIT compiler?)
- **Memory usage:** large, memory-hungry Python programs might end up taking less space than they do in CPython.
- **Compatibility:** PyPy is highly compatible with existing python code. It supports ctypes and can run popular python libraries like twisted and django.
- **Sandboxing:** PyPy provides the ability to run untrusted code in a fully secure way.
- **Stackless:** PyPy comes by default with support for stackless mode, providing micro-threads for massive concurrency.

Der PyPy-Interpreter (II)

- Vorteile:
  - Implementiert (irgendwann) vollständiges Python inklusive Python-Std-Libs
  - Tatsächlich of schneller als Cpython
- Nachteile:
  - Jedes weitere (C-)Modul muss angepasst und neu für Pypy kompiliert werden. Z.B. numpy....
  - Diese Anpassungen sind extrem aufwändig und teuer:
  - Bsp: `import numpypy`

Der Pypy-Interpreter (III)



Inhalt

- Motivation
- Wechsel der Software
  - Anderer Interpreter
  - Andere Programmiersprache
- Wechsel der Rechnerarchitektur
  - Threading
  - GPU
- Zusammenfassung

Von Python-Funktionen zu CPython-Erweiterungen

- Anstatt den Interpreter zu wechseln, wechselt nur der Algorithmus die Programmiersprache:
  - CPython: Python → C (oder ähnliche…)
  - Keine Angst, viele Hilfsmittel bereits vorhanden!
- Für diesen Vortrag ausgewählte Hilfsmittel:
  - weave/blitz
  - cython
  - ctypes
  - boost-python

weave/blitz (I)

- Teil von scipy: scipy.weave!

„The scipy.weave (below just weave) package provides tools for including C/C++ code within in Python code. This offers both another level of optimization to those who need it [...]. Inlining C/C++ code within Python generally results in speed ups of 1.5x to 30x speed-up over algorithms written in pure Python (However, it is also possible to slow things down...).“

- Benutzt numpy.distutils, um C/C++-Compiler zu erkennen.
- Alles bleibt unter einem Dach: Python
- Bis zu 6x schneller als NumPy! (Nach dem Kompilieren!)
- Beispiele aus: http://docs.scipy.org/doc/scipy/reference/tutorial/weave.html

weave/blitz (II)

- inline-Funktion für C/C++-Code direkt in Python aus
- Beispiel 1:

```
>>> from scipy import weave
>>> a = 1
>>> weave.inline('printf("%d\\n",a);',['a'])
1
```

- Was folgt für umfangreichere Funktionen?
- Was ist mit komplexeren Datentypen?

weave/blitz (III)

- Beispiel 2: Binäre Suche (Python)

```
def binary_search(seq, t):
  min = 0;
  max = len(seq) - 1
  while 1:
    if max < min:
      return -1
    m = (min + max) / 2
    if seq[m] < t:
      min = m + 1
    elif seq[m] > t:
      max = m - 1
    else:
      return m
```

weave/blitz (III)

- Beispiel 2: Binäre Suche (mixed C/C++)

```
def c_int_binary_search(seq,t):
  code = """
       #line 29 "binary_search.py"
       int val, m, min = 0; int max = seq.length() - 1;
       PyObject *py_val;
       for(;;) {
         if (max < min ) {
           return_val = Py::new_reference_to(Py::Int(-1)); break;
         }
         m = (min + max) /2;
         val = py_to_int(PyList_GetItem(seq.ptr(),m),"val");
         if (val < t)
           min = m + 1;
         else if (val > t)
           max = m - 1;
         else {
            return_val = Py::new_reference_to(Py::Int(m)); break;
         }
       }
       """
  return inline(code,['seq','t'])
```

weave/blitz (IV)

- Beispiel: blitz++ mit NumPy-Array

```
from weave.blitz_tools import blitz_type_factories
from weave import scalar_spec
from weave import inline

def _cast_copy_transpose(type,a_2d):
    assert(len(shape(a_2d)) == 2)
    new_array = zeros(shape(a_2d),type)
    NumPy_type = scalar_spec.NumPy_to_blitz_type_mapping[type]
    code = \
    """
    for(int i = 0;i < _Na_2d[0]; i++)
        for(int j = 0;  j < _Na_2d[1]; j++)
            new_array(i,j) = (%s) a_2d(j,i);
    """ % NumPy_type
    inline(code,['new_array','a_2d'],
           type_factories = blitz_type_factories,compiler='gcc')
    return new_array
```

cython (I)

http://cython.org

Cython is an **optimising static compiler** for both the **Python** programming language and the extended Cython programming language (based on **Pyrex**). It makes writing C extensions for Python as easy as Python itself.

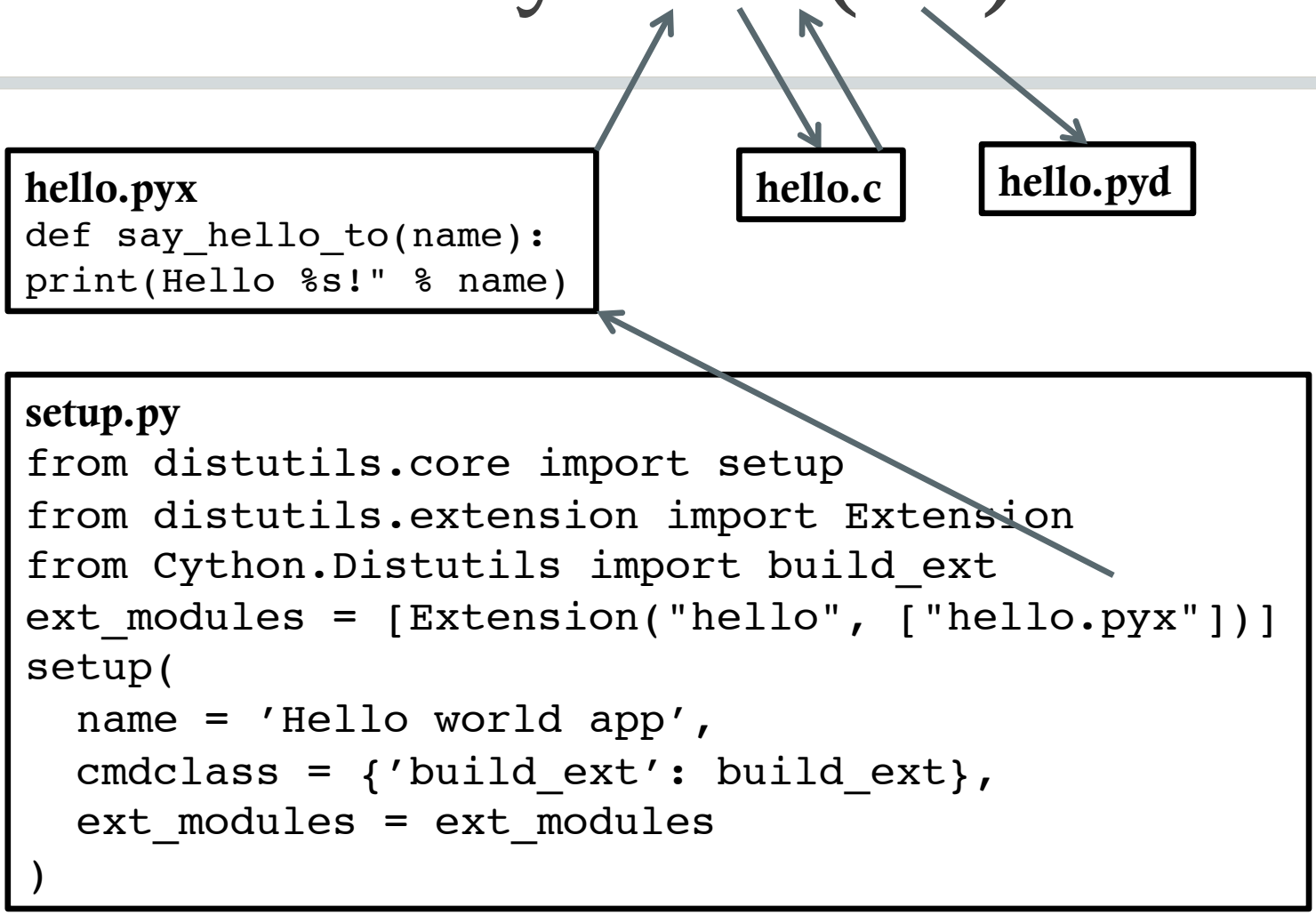
Cython gives you the combined power of Python and C to let you

- write Python code that calls back and forth from and to C or C++ code natively at any point.
- easily tune readable Python code into plain C performance by adding static type declarations.
- use combined source code level debugging to find bugs in your Python, Cython and C code.
- interact efficiently with large data sets, e.g. using multi-dimensional NumPy arrays.
- quickly build your applications within the large, mature and widely used CPython ecosystem.
- integrate natively with existing code and data from legacy, low-level or high-performance libraries and applications.

cython (II)

- Gute Quelle für weitere Recherche:
  S Behnel, R Bradshaw, D Seljebotn, *Cython tutorial* in *Proceedings of the 8th Python in Science conference (SciPy 2009)*, G Varoquaux, S van der Walt, J Millman (Eds.), pp. 4-14
  http://conference.scipy.org/proceedings/SciPy2009/paper_1/full_text.pdf
- Andere Herangehensweise:
  - Erweiterungsmodule in Python-ähnlicher Sprache schreiben
  - Transparent (durch Python) kompilieren
  - Einfache Re-Integration der kompilierten Funktionen
- Vor allem für größere Projekte geeignet!

cython (III)


hello.pyx
def say_hello_to(name):
print(Hello %s!" % name)
hello.c
hello.pyd
setup.py
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
ext_modules = [Extension("hello", ["hello.pyx"])]
setup(
name = 'Hello world app',
cmdclass = {'build_ext': build_ext},
ext_modules = ext_modules
)

cython (IV)

- Auch Anbindungen für NumPy vorhanden: http://docs.cython.org/src/tutorial/numpy.html
- Besonderheit auch hier: Nur wenig Python-Code muss angepasst werden, um C-Geschwindigkeit zu erhalten.
- Aber: Weder C- noch Python als Sprache ist zu verwenden…

ctypes (I)

http://docs.python.org/2/library/ctypes.html

- Teil von Python + Numpy-Ext. numpy.ctypeslib
- Sehr elementarer (Low-Level-)Ansatz:
  - Numpy-Arrays → C-Pointer
  - Python-Variablen → C-Variablen
  - Einfaches Laden von dynamischen Bibliotheken
- Volle Kontrolle aber auch potentiell weniger Arbeitserleichterung

ctypes (II)

ctest.c

```
void cfun(const double* in_arr, int width, int height, double* out_arr)
{
  int i;
  for (y = 0; y < height; ++y) {
    for (x = 0; x < width; ++x, ++i) {
        outdata[i] = indata[i] +1;
    }
  }
}
```

gcc –fPIC –shared –o ctest.so ctest.c

ctest.py

```
from numpy.ctypeslib import ndpointer
lib = ctypes.cdll.LoadLibrary('./ctest.so')
fun = lib.cfun
fun.restype = None
fun.argtypes = [ndpointer(ctypes.c_double),
                ctypes.c_size_t,
                ctypes.c_size_t,
                ndpointer(ctypes.c_double)]
```

ctypes (III)

- Hübscher durch „dekorative“ Funktionen:

```
def my_add1(arr):
  assert(np.dtype(arr) == np.double)
  out_arr = np.zeros_like(arr)
  shp =  arr.shape
  fun(arr, shp[0], shp[1],out_arr)
  return out_arr
```

- Aufruf mit:

```
my_add1(.....)
```

boost-python (I)
http://www.boost.org/libs/python

*„[…] **Boost.Python**, a C++ library which enables seamless interoperability between C++ and the Python programming language. […] flexible interface, and many new capabilities, including support for:*

- References and Pointers
- Globally Registered Type Coercions
- Automatic Cross-Module Type Conversions
- Efficient Function Overloading
- C++ to Python Exception Translation … and many more

boost-python (II)

- Idee: Dem Entwickler auf der C/C++-Seite volle Kontrolle erlauben, aber ihn in der Anbindung zu Python bestmöglich unterstützen
- Ziel: Deutlich komfortableres ctypes!
- Vor allem geeignet für C++-Anbindungen und größere Projekte
  - VIGRA benutzt z.B. Boost-Python zur Realisierung der vigranumpy-Anbindung
- Beispiel entnommen aus: http://www.boost.org/doc/libs/1_54_0/libs/python/doc/tutorial

boost-python (III)

```
#include <boost/python.hpp>

char const* greet()
{
   return "hello, world";
}

BOOST_PYTHON_MODULE(hello_ext)
{
    using namespace boost::python;
    def("greet", greet);
}
```

Kompilieren und installieren in Pythons site-packages Ordner

```
>>> import hello_ext
>>> print hello_ext.greet()
hello, world
```

boost-python (IV)

- Build-Prozess ebenfalls automatisiert:
  - „Jamroot“-Datei gibt Projektparameter an
  - „user-config.jam“ gibt Benutzerdefinitionen an (z.B. Ort der der Python Installation etc.)
- Im Idealfall: Kompilieren = ein Aufruf von `bjam`
- Auch andere Build-Tools können verwendet werden (z.B. CMake)

Inhalt

- Motivation
- Wechsel der Software
  - Anderer Interpreter
  - Andere Programmiersprache
- Wechsel der Rechnerarchitektur
  - Threading
  - GPU
- Zusammenfassung

Rechnerarchitekturen

- Bisher: Ein-dimensionale Kommandostruktur:
  Eingabe → Verarbeitung → Ausgabe
- Erhöhung der Performanz durch Parallelisierung
  - 1-Prozessor
    →Multiprozessor/Multithreading
  - Falls nötig:
    Multiprozessor → Massive Multiprozessoren
    Also CPU → GPU

Single-Threading in Python

- Aufbauend auf Threading-Präsentation (P. Vergain)
  http://de.slideshare.net/pvergain/multiprocessing-with-python-presentation
  - Primzahlen berechnen
  - Vergleich der Effizienz Threads vs. Prozesse
- Beispiel: Primzahlen summieren

```
def sum_primes(n):
  """Calculates sum of all primes below n"""
  return sum([x for x in xrange(2,n) if isprime(x)])
```

- Eigentliches Programm (Single-Threading)

```
for i in xrange(100000,5000000, 100000):
  print sum_primes(i)
```

Multi-Threading in Python (III)

- Worker Thread

```
def do_work(q):
  while True:
    try:
      x = q.get(block=False)
      print sum_primes(x)
    except Empty:
      break
```

- Eigentliches Programm (Multi-Threading):

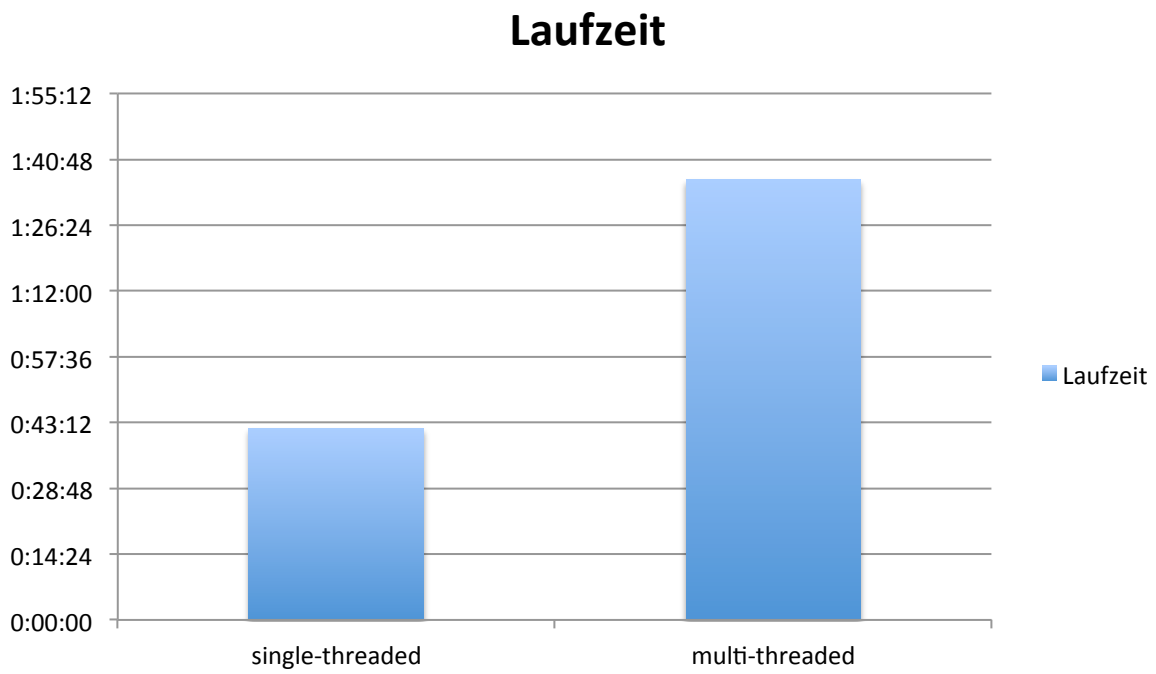
```
work_queue = Queue()
for i in xrange(100000,5000000, 100000):
  work_queue.put(i)

threads = [Thread(target=do_work, args=(work_queue,) for i in range(8))]

for t in threads:
  t.start()
for t in threads:
  t.join()
```

Multi-Threading: Effizienz

- Geschwindigkeitsvergleich:



Multi-Threading in Python (II)

- Problem: CPython benutzt zwar echte (OS-)Threads, aber nur einer zur Zeit darf Python-Bytcode ausführen.
  → Global Interpreter Lock (GIL)
- Vereinfacht vieles auf der Entwurfsseite
- Führt uns nicht zum Ziel!
- Echtes paralleles Multithreading dennoch möglich:
  - Auf C/C++-Seite (aber hier nicht gezeigt) – z.B. mit Boost-Threads oder QT-Threads
  - Mit anderem Interpreter (z.B. Jython ooder IronPython)

Multi-Processing in Python (I)

- Im Gegensatz zu Threads relativ schwergewichtig:
  - Eigener Speicher,
  - eigene Ausführungseinheit
- Laufen unter Python (wirklich) parallel, da kein Widerspruch zum GIL
- Syntaktisch wenig Unterschied zu Multi-Threading

Multi-Processing in Python (II)

- Worker Process (unverändert)

```
def do_work(q):
  while True:
    try:
      x = q.get(block=False)
      print sum_primes(x)
    except Empty:
      break
```

- Eigentliches Programm (Multi-Processing):

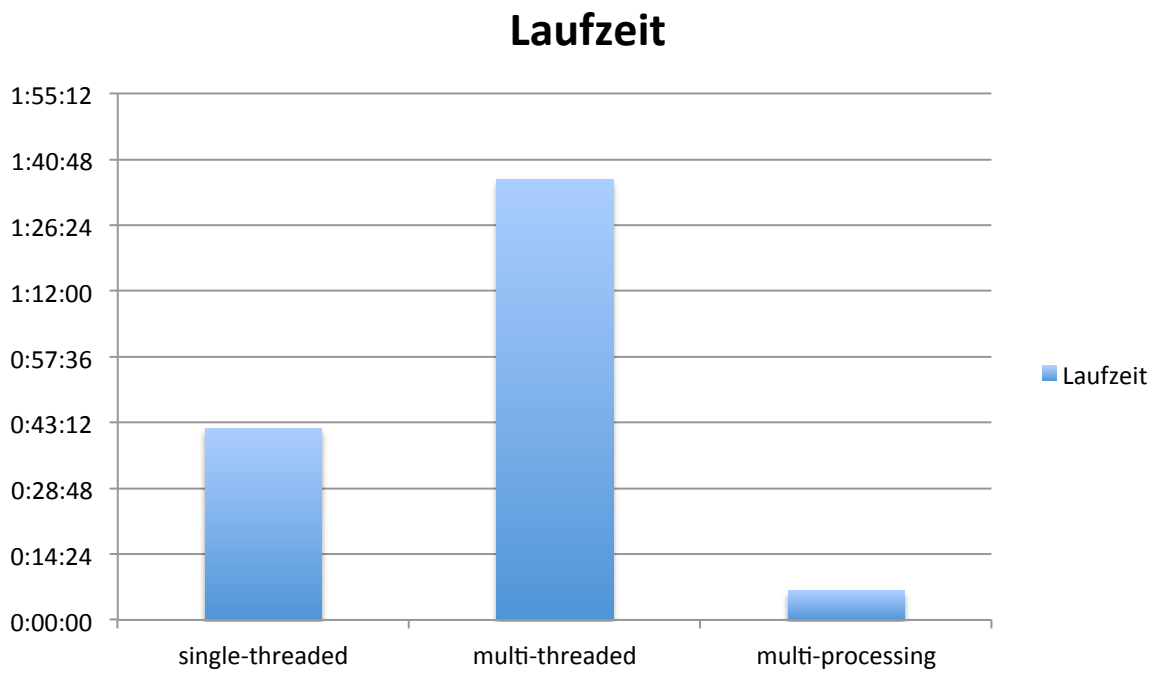
```
work_queue = Queue()
for i in xrange(100000,5000000, 100000):
  work_queue.put(i)

processes = [Process(target=do_work, args=(work_queue,) for i in range(8))]

for p in processes:
  p.start()
for p in processes:
  p.join()
```

Multi-Processing: Effizienz

- Geschwindigkeitsvergleich:



Inhalt

- Motivation
- Wechsel der Software
  - Anderer Interpreter
  - Andere Programmiersprache
- Wechsel der Rechnerarchitektur
  - Threading
  - GPU
- Zusammenfassung

PyOpenCL (I)
http://mathema.tician.de/software/pyopencl/

- Idee von OpenCL: Führe Algorithmen auf allen (passenden) zur Verfügung stehenden Prozessoren aus:
  - CPUs
  - VSXs
  - GPUs
- Erfordert OpenCL-Installation und OpenCL-fähige Treiber der jew. Hardwarekomponenten
- PyOpenCL ermöglicht direkte Verwendung der OpenCL ohne C/C++ benutzen zu müssen.

PyOpenCL (II)

- Beispiel (schnelle Array-Addition):

```
import pyopencl as cl
import numpy as np
import numpy.linalg as la

a = np.random.rand(50000).astype(np.float32)
b = np.random.rand(50000).astype(np.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)

prg.sum(queue, a.shape, None, a_buf, b_buf, dest_buf)

a_plus_b = np.empty_like(a)
cl.enqueue_copy(queue, a_plus_b, dest_buf)

print(la.norm(a_plus_b - (a+b)), la.norm(a_plus_b))
```

```
prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
    """).build()
```

PyOpenCL (III)

- Vorteile
  - Sobald Code auf der GPU ausgeführt werden kann, enorme Beschleunigung möglich
  - Unterstützung nicht Hersteller-abhängig
  - Viele Tutorials: http://documen.tician.de/pyopencl/
- Nachteile:
  - Speicher-Optimierung oft nötig!
  - Treiber muss verfügbar sein
  - Noch eine neue Sprache bzw. C-Dialekt…

PyCUDA (I)

http://mathema.tician.de/software/pycuda/

- CUDA: Programmierschnittstelle zu *nvidia*-Grafik- und dedizierten. Rechenkarten
- Bereits große Verbreitung im Bereich des wiss. Rechnens
- Zwei verbreitete Wege unter Python:
  - Mumbapro (soll viel können, kostet aber auch viel)
  - PyCUDA („nur“ ein CUDA-Wrapper, aber umsonst)
- Daher: Vorstellung von PyCUDA

PyCUDA (II)

- Beispiel: Multiplikation von Arrays

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy

from pycuda.compiler import SourceModule

mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1), grid=(1,1))

print dest-a*b
```

PyCUDA (III)

- Oder einfacher (für viele Operationen):

```
import pycuda.gpuarray as gpuarray
import pycuda.driver as cuda
import pycuda.autoinit
import numpy

a_gpu = gpuarray.to_gpu(numpy.random.randn(4,4).astype(numpy.float32))
a_doubled = (2*a_gpu).get()
print a_doubled
print a_gpu
```

- Nachteil auch hier: Neue Sprachumgebung muss erlernt werden
- Aber: Große CUDA-Community

Inhalt

- Motivation
- Wechsel der Software
  - Anderer Interpreter
  - Andere Programmiersprache
- Wechsel der Rechnerarchitektur
  - Threading
  - GPU
- Zusammenfassung

Zusammenfassung

- Schnelles wissenschaftliches Rechnen auch jenseits von NumPy/SciPy-Funktionen
- Viele Wege führen zum Ziel:
  - Wahl des Interpreters (CPython vs. PyPy)
  - Wahl der Programmiersprache (Python vs. C/C++)
  - Wahl der Rechnerarchitektur (CPU vs. GPU, single- vs. multi-threaded)
- Für alle diese Wege existieren (mehrere) Python-Lösungen.

… Zeit, diese auszuprobieren!

Vielen Dank für die Aufmerksamkeit!

Zeit für Fragen!

Dieser Vortrag wird in Kürze auf der Projekt-Homepage veröffentlicht.