Working with Numba

Numba is an accelerator library for Python, which just-in time compiles Python code into fast machine code. If used right, its performance can be close to optimized C code. Moreover, it supports offloading of kernels to GPU devices and shared memory parallelism.

The following example from the Numba homepage provides a very first idea of what Numba does.

```
from numba import njit, jit
from numpy import arange

# jit decorator tells Numba to compile this function.
# The argument types will be inferred by Numba when function is called.
@jit(nopython=True)
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i,j]
    return result

a = arange(9).reshape(3,3)
print(sum2d(a))
```

```
36.0
36.0
```

On its first call the `sum2d` function is just-in-time compiled into fast executable code and then executed. All that is needed is the decorator `@njit`.

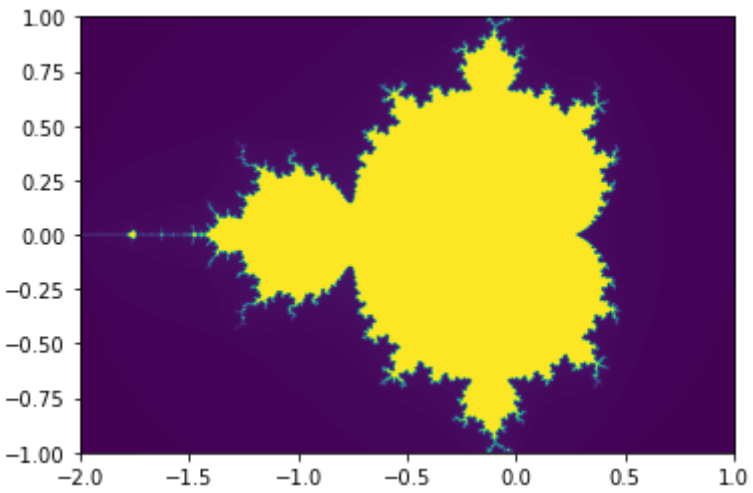
In the following we want to use Numba to compute the Mandelbrot fractal and measure its performance. First, we define a simple convenient timer in Python.

```
import time

class Timer:
    def __enter__(self):
        self.start = time.time()
        return self

    def __exit__(self, *args):
        self.end = time.time()
        self.interval = self.end - self.start
```

```
%matplotlib inline

#from __future__ import print_function, division, absolute_import
import numpy as np
from pylab import imshow, jet, show, ion

def mandel(x, y, max_iters):
    """
    Given the real and imaginary parts of a complex number,
    determine if it is a candidate for membership in the Mandelbrot
    set given a fixed number of iterations.
    """
    i = 0
    c = complex(x,y)
    z = 0.0j
    for i in range(max_iters):
        z = z*z + c
        if (z.real*z.real + z.imag*z.imag) >= 4:
            return i

    return 255

def create_fractal(min_x, max_x, min_y, max_y, image, iters):
    height = image.shape[0]
    width = image.shape[1]

    pixel_size_x = (max_x - min_x) / width
    pixel_size_y = (max_y - min_y) / height
    for x in range(width):
        real = min_x + x * pixel_size_x
        for y in range(height):
            imag = min_y + y * pixel_size_y
            color = mandel(real, imag, iters)
            image[y, x] = color

    return image

image = np.zeros((2000, 3000), dtype=np.uint8)

with Timer() as t:
    mandelbrot = create_fractal(-2.0, 1.0, -1.0, 1.0, image, 20)
print("Time to create fractal: {0}".format(t.interval))

imshow(mandelbrot, extent=[-2, 1, -1, 1])
```

```
Time to create fractal: 12.034635305404663
```

```
<matplotlib.image.AxesImage at 0x7f9adddedb80>
```

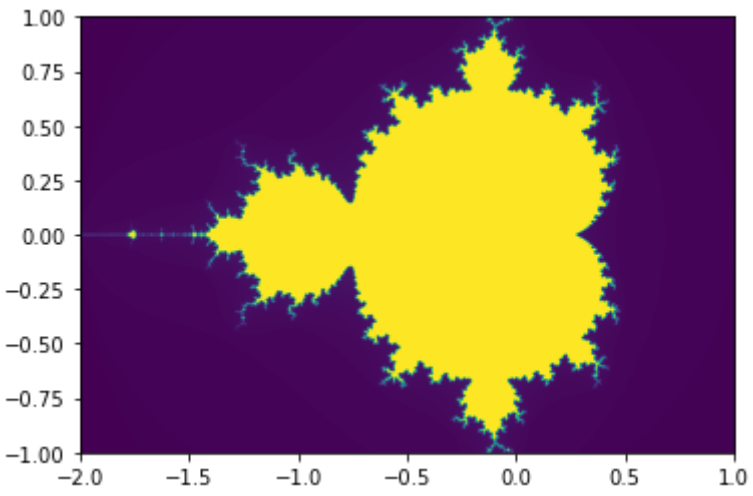


This is fairly slow. The problem is that we have three nested for-loops. In each inner iteration a call to the Python interpeter needs to be performed. Python is not designed for speedy handling of such loop constructs. However, we can improve it by enabling Just-In-Time compilation of the routines with Numba. This is done in the following code, where the `@njit` keyword was added.

```
%matplotlib inline

from numba import jit
import numpy as np
from pylab import imshow, jet, show, ion

@njit
def mandel(x, y, max_iters):
    """
    Given the real and imaginary parts of a complex number,
    determine if it is a candidate for membership in the Mandelbrot
    set given a fixed number of iterations.
    """
    i = 0
    c = complex(x,y)
    z = 0.0j
    for i in range(max_iters):
        z = z*z + c
        if (z.real*z.real + z.imag*z.imag) >= 4:
            return i

    return 255

@njit
def create_fractal(min_x, max_x, min_y, max_y, image, iters):
    height = image.shape[0]
    width = image.shape[1]

    pixel_size_x = (max_x - min_x) / width
    pixel_size_y = (max_y - min_y) / height
    for x in range(width):
        real = min_x + x * pixel_size_x
        for y in range(height):
            imag = min_y + y * pixel_size_y
            color = mandel(real, imag, iters)
            image[y, x] = color

    return image

image = np.zeros((2000, 3000), dtype=np.uint8)

with Timer() as t:
    mandelbrot = create_fractal(-2.0, 1.0, -1.0, 1.0, image, 20)
print("Time to create fractal: {0}".format(t.interval))

imshow(mandelbrot, extent=[-2, 1, -1, 1])
```

```
Time to create fractal: 0.3081510066986084
```

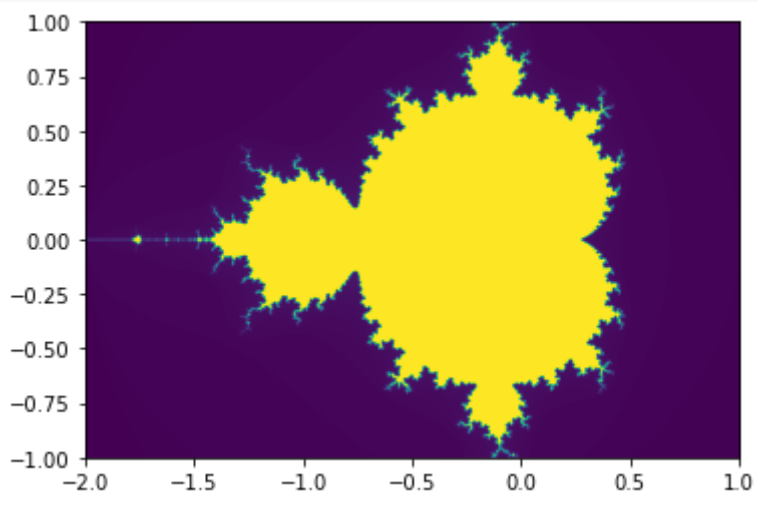
```
<matplotlib.image.AxesImage at 0x7f9ade196dc0>
```



```
%matplotlib inline

from numba import njit, prange
import numpy as np
from pylab import imshow, jet, show, ion

@njit
def mandel(x, y, max_iters):
    """
    Given the real and imaginary parts of a complex number,
    determine if it is a candidate for membership in the Mandelbrot
    set given a fixed number of iterations.
    """
    i = 0
    c = complex(x,y)
    z = 0.0j
    for i in range(max_iters):
        z = z*z + c
        if (z.real*z.real + z.imag*z.imag) >= 4:
            return i

    return 255

@njit(['uint8[:,:](float64, float64, float64, float64, uint8[:, :], uint8)'], parallel=True)
def create_fractal(min_x, max_x, min_y, max_y, image, iters):
    height = image.shape[0]
    width = image.shape[1]

    pixel_size_x = (max_x - min_x) / width
    pixel_size_y = (max_y - min_y) / height
    for x in prange(width):
        real = min_x + x * pixel_size_x
        for y in range(height):
            imag = min_y + y * pixel_size_y
            color = mandel(real, imag, iters)
            image[y, x] = color
    return image

image = np.zeros((2000, 3000), dtype=np.uint8)

with Timer() as t:
    mandelbrot = create_fractal(-2.0, 1.0, -1.0, 1.0, image, 20)
print("Time to create fractal: {0}".format(t.interval))

imshow(mandelbrot, extent=[-2, 1, -1, 1])
```

```
Time to create fractal: 0.02453899383544922
```

```
<matplotlib.image.AxesImage at 0x7f9ade00c640>
```



The key to the parallelization is the `prange` command in the for-loop. This is similar to the parallel for-loop in low-level shared memory parallel libraries such as OpenMP and tells Numba to spread out the computation to multiple CPU cores. However, it is essential that Numba knows all data types, so that no Python calls will be performed during the parallel loop.

We can easily inspect the code that Numba generates. Consider the following simple function.

```
@njit
def mysum(a, b):
    return a + b

c = mysum(3, 4)
```

We can now inspect the generated LLVM code for this function.

```
for v, k in mysum.inspect_llvm().items():
    print(v, k)
```

```
(int64, int64) ; ModuleID = 'mysum'
source_filename = "<string>"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

@"_ZN08NumbaEnv8__main__10mysum$2414Exx" = common local_unnamed_addr global i8* null
@.const.mysum = internal constant [6 x i8] c"mysum\00"
@PyExc_RuntimeError = external global i8
@".const.missing Environment: _ZN08NumbaEnv8__main__10mysum$2414Exx" = internal constant [59
x i8] c"missing Environment: _ZN08NumbaEnv8__main__10mysum$2414Exx\00"

; Function Attrs: nofree norecurse nounwind writeonly
define i32 @"_ZN8__main__10mysum$2414Exx"(i64* noalias nocapture %retptr, { i8*, i32, i8* }**
noalias nocapture readnone %excinfo, i64 %arg.a, i64 %arg.b) local_unnamed_addr #0 {
entry:
  %.14 = add nsw i64 %arg.b, %arg.a
  store i64 %.14, i64* %retptr, align 8
  ret i32 0
}

define i8* @"_ZN7cpython8__main__10mysum$2414Exx"(i8* nocapture readnone %py_closure, i8*
%py_args, i8* nocapture readnone %py_kws) local_unnamed_addr {
entry:
  %.5 = alloca i8*, align 8
  %.6 = alloca i8*, align 8
  %.7 = call i32 (i8*, i8*, i64, i64, ...) @PyArg_UnpackTuple(i8* %py_args, i8* getelementptr
inbounds ([6 x i8], [6 x i8]* @.const.mysum, i64 0, i64 0), i64 2, i64 2, i8** nonnull %.5,
i8** nonnull %.6)
  %.8 = icmp eq i32 %.7, 0
  br i1 %.8, label %entry.if, label %entry.endif, !prof !0

entry.if:                                         ; preds =
%entry.endif.endif.endif.endif.endif, %entry.endif.endif.endif, %entry
  ret i8* null

entry.endif:                                      ; preds = %entry
  %.12 = load i8*, i8** @"_ZN08NumbaEnv8__main__10mysum$2414Exx", align 8
  %.17 = icmp eq i8* %.12, null
  br i1 %.17, label %entry.endif.if, label %entry.endif.endif, !prof !0

entry.endif.if:                                   ; preds = %entry.endif
  call void @PyErr_SetString(i8* nonnull @PyExc_RuntimeError, i8* getelementptr inbounds ([59
x i8], [59 x i8]* @".const.missing Environment: _ZN08NumbaEnv8__main__10mysum$2414Exx", i64
0, i64 0))
  ret i8* null

entry.endif.endif:                                ; preds = %entry.endif
  %.21 = load i8*, i8** %.5, align 8
  %.24 = call i8* @PyNumber_Long(i8* %.21)
  %.25 = icmp eq i8* %.24, null
  br i1 %.25, label %entry.endif.endif.endif, label %entry.endif.endif.if, !prof !0

entry.endif.endif.if:                             ; preds = %entry.endif.endif
  %.27 = call i64 @PyLong_AsLongLong(i8* nonnull %.24)
  call void @Py_DecRef(i8* nonnull %.24)
  br label %entry.endif.endif.endif

entry.endif.endif.endif:                          ; preds = %entry.endif.endif,
%entry.endif.endif.if
  %.22.0 = phi i64 [ %.27, %entry.endif.endif.if ], [ 0, %entry.endif.endif ]
  %.32 = call i8* @PyErr_Occurred()
  %.33 = icmp eq i8* %.32, null
  br i1 %.33, label %entry.endif.endif.endif.endif, label %entry.if, !prof !1

entry.endif.endif.endif.endif:                    ; preds = %entry.endif.endif.endif
  %.37 = load i8*, i8** %.6, align 8
  %.40 = call i8* @PyNumber_Long(i8* %.37)
  %.41 = icmp eq i8* %.40, null
  br i1 %.41, label %entry.endif.endif.endif.endif.endif, label
%entry.endif.endif.endif.endif.if, !prof !0

entry.endif.endif.endif.endif.if:                 ; preds = %entry.endif.endif.endif.endif
  %.43 = call i64 @PyLong_AsLongLong(i8* nonnull %.40)
  call void @Py_DecRef(i8* nonnull %.40)
  br label %entry.endif.endif.endif.endif.endif

entry.endif.endif.endif.endif.endif:              ; preds = %entry.endif.endif.endif.endif,
%entry.endif.endif.endif.endif.if
  %.38.0 = phi i64 [ %.43, %entry.endif.endif.endif.endif.if ], [ 0,
%entry.endif.endif.endif.endif ]
  %.48 = call i8* @PyErr_Occurred()
  %.49 = icmp eq i8* %.48, null
  br i1 %.49, label %entry.endif.endif.endif.endif.endif.endif, label %entry.if, !prof !1

entry.endif.endif.endif.endif.endif.endif:        ; preds =
%entry.endif.endif.endif.endif.endif
  %.14.i = add nsw i64 %.38.0, %.22.0
  %.74 = call i8* @PyLong_FromLongLong(i64 %.14.i)
```

```
  ret i8* %.74
}

declare i32 @PyArg_UnpackTuple(i8*, i8*, i64, i64, ...) local_unnamed_addr

declare void @PyErr_SetString(i8*, i8*) local_unnamed_addr

declare i8* @PyNumber_Long(i8*) local_unnamed_addr

declare i64 @PyLong_AsLongLong(i8*) local_unnamed_addr

declare void @Py_DecRef(i8*) local_unnamed_addr

declare i8* @PyErr_Occurred() local_unnamed_addr

declare i8* @PyLong_FromLongLong(i64) local_unnamed_addr

; Function Attrs: norecurse nounwind readnone
define i64 @"cfunc._ZN8__main__10mysum$2414Exx"(i64 %.1, i64 %.2) local_unnamed_addr #1 {
entry:
  %.14.i = add nsw i64 %.2, %.1
  ret i64 %.14.i
}

; Function Attrs: nounwind
declare void @llvm.stackprotector(i8*, i8**) #2

attributes #0 = { nofree norecurse nounwind writeonly }
attributes #1 = { norecurse nounwind readnone }
attributes #2 = { nounwind }

!0 = !{!"branch_weights", i32 1, i32 99}
!1 = !{!"branch_weights", i32 99, i32 1}
```

Numba has a number of features to not only target CPUs, but also GPU architectures. It is a fast moving project and widely used to speed up Python code to levels that were previously only known in languages such as C and Fortran, making it possible to combine the ease of Fortran with the performance of low-level languages.

By Timo Betcke