Speeding up python

Mads M. Pedersen

1 Introduction

Python is normally considered as a "fast typing - slow executing" programming language. There is however several ways to speed up execution. This document describes the most important methods.

2 Algorithm

The efficiency of the algorithm is essential. Sorting a list of random numbers is a classical example which can be solved in O(n·log n) using an efficient algorithm, while a brute-force search algorithm requires O(n!). I.e. 100 numbers are sorted in negligible time using an efficient algorithm while the brute-force search would not finish within our life even if all computers of the world had been calculating since Big Bang.

As example consider a function that determines if a number is a prime. The first naive implementation, pcheck, checks if n is divisible with the numbers 2..n-1.

```
def pcheck(n):
    result = True
    for y in xrange(2,n):
        if n%y==0:
            result = False
    return result
```

The running time, both best and worst case, is linear to the size of n, O(n). In this example the best case execution time can be reduced to constant time, O(1), if the function returns false as soon `result` becomes false.
Moreover the worst case complexity can be altered to $O(n^{1/2})$ as there is no need to check values above $\sqrt{n}$:

```
def pcheck_opt(n):
    for y in xrange(2,int(math.sqrt(n))+1):
        if n%y==0:
            return False
    return True
```

It is possible to reduce the number of modulo operations to one third as seen in pcheck_opt2 or even more, but it may not be worth the cost in code complexity:

```
def pcheck_opt2(n):
    if n==2 or n==3: return True
    if n%2==0 or n%3==0: return False
    for y in xrange(6,int(math.sqrt(n))+2,6):
        if n%(y-1)==0 or n%(y+1)==0:
            return False
    return True
```

Table 1 shows execution times of the three functions. As expected pcheck_opt is roughly $\sqrt{n}$ times faster than pcheck, while pcheck_opt2 is three times faster than pcheck_opt in the last case.

	pcheck	pcheck_opt	pcheck_opt2
n = 179424691	11.410s	0.001s	0.001s
n = 179424692	11.477s	0.000s	0.000s
n = 32416190071	-	0.059s	0.020s

Table 1

3 Data structure

In some cases the data structure has a huge impact on the performance. As example consider two functions that takes a list of n random integers in the range 2..N and a list of all primes up to N and returns the prime elements from the random list.

In the first function, `list_structure`, the structure of the input primes list is preserved while the second, `set_structure`, converts the prime list to a dictionary.

```
def list_structure(lst, primes):
    result = []
    for n in lst:
        if n in primes:
            result.append(n)
    return result
```

```
def set_structure(lst, primes):
    primes = set(primes)
    result = []
    for n in lst:
        if n in primes:
            result.append(n)
    return result
```

The asymptotic running time for look-up is expected to be constant, O(1) in a dictionary and O(N) in the list, so the results in Table 2 is not surprising but nevertheless impressing.

n	N	list_structure	set_structure
1.000.000	1000	3.462s	0.073s
1.000.000	10000	25.203s	0.079s

Table 2

4 Loop structure

Use map or list comprehension instead of for-loops if possible. As example consider these three functions that all return the input list after applying the function f to all elements:

```
def for_loop(f, lst):
    res = []
    for x in lst:
        res.append(f(x))
    return res
```

```
def list_comprehension(f, lst):
  return [f(x) for x in lst]
```

```
def map_to_list(f,lst):
  return map(f,lst)
```

In Table 3 list_comprehension and map_to_list is seen to be significantly faster than for_loop

f	lst	for_loop	list_comprehension	map_to_list
`str`	`xrange(10**7)`	3.078s	2.240s	1.832s
`lambda x: str(x).upper()`	`xrange(10**7)`	5.605s	4.337s	4.278s

Table 3

5 String concatenation

Be carefull when using string concatenation. The '+' concatenation operator is very slow.
The following three functions return a string containing the string representation of all elements of the input list:

```
def str_for(lst):
    s = ""
    for x in lst:
        s += str(x)
    return s
```

```
def str_join(lst):
    s_lst = []
    for s in lst:
        s_lst.append(str(s))
    return "".join(s_lst)
```

```
def str_comp_join(lst):
    s_lst = [str(s) for s in lst]
    return "".join(s_lst)
```

In Table 4 `str_join` is seen to be remarkably faster than `str_for`, and `str_comp_join` is even faster.

lst	str_for	str_join	str_comp_join
xrange(10**6)	1.669s	0.317s	0.256s

Table 4

6 Compiled code

While writing Python code is fast and easy because of python's high level, execution is basically slow. The main reason is that python is interpreted and dynamically typed.

Table 5 illustrates the conceptual difference between executing a line of code in python and C and the practical consequences in terms of execution time.

Code	To do at runtime in highlighted line	Time for $N=10^7$

`for i in xrange(N):`
`  C[i] = A[i]+B[i]`
	• Interpret code line
	• Determine types of C[i] and A[i] and B[i]
	• Search for corresponding add method
	• Apply add method
	  ○ Load A[i] and B[i] into registers
	  ○ Add A[i] and B[i]
	  ○ Convert to appropiate data type in order to handle overflow
	  ○ Save C[i] into memory
		1.207 s

`for(i = 0;i < N ;i++){`
`  C[i] = A[i]+B[i];`
`}`
	Load A[i] and B[i] into register
	Add A[i] and B[i]
	Save C[i] into memory
		0.033 s

Table 5

Fortunately it is possible to use compiled code in Python. There are some different alternatives:

6.1 Numpy

The Numpy library, which is included in most Python distributions, provides compiled functions to deal with arrays and matrices.

If A and B are numpy arrays, then the vector sum can be computed directly using the '+' operator and the execution speed matches the speed of C.

Code (A and B must be numpy arrays)	Time for $\|A\|=\|B\|=10^7$
`C = A + B`	0.030 s

Table 6

Considering the prime check function again, it can be rewritten to utilize the numpy functions, e.g:

```
import numpy as np
def npcheck(n):
    y = np.arange(2,int(math.sqrt(n))+1)
    return np.all(n%y>0)
```

Table 7

This implementation speeds up the worst case execution time more than five times, but in contrast to `pcheck_opt2`, `npcheck` has the same best and worst case running time as the modulo and comparion operations are computed for all elements in y, before `np.all` starts to check the result for `False` values:

	pcheck_opt2	npcheck
n = 688846502588399	2.972s	0.551s
n = 688846502588398	0.000s	0.531s

Table 8

As seen above it is often possible to replace a python loop with numpy operations, but it is often a nontrivial task to figure out how.

Consider the task of finding the values of local maxima in an array:

		Example
A	The array	[0 2 4 2 0 2 5 5 2 -1]
B=A[1:]-A[:-1]	The gradient	[2 2 -2 -2 2 3 0 -3 -3]
C = (B[:-1]>=1) & (B[1:]<=-1)	Local max where negative gradient follows positive gradient	[F T F F F F F F]
D = np.r_[False, C, False]	Concatenate with False to align with A	[F F T F F F F F F F]
A[D]	Extract local maximum	[4]

Table 9

Using this procedure all one-element maximum peaks are found, but the two-element plateau with value 5 is not, and it is not possible to extend this procedure to find plateaus of arbitrary sizes with out sacrificing the performance.

In this case a for loop is required. Below three almost equal implementations are shown. maxima1 looks up the current and last value at every use, maxima2 iterates over the numpy array and keeps the last value in a variable, and maxima3 is equal to maxima2 except that it iterates over the python list generated by the tolist() function.

```
def maxima1(A):
  UP, DOWN = 0,1
  delta = UP
  last_v = A[0]
  maxima = []

  for i in xrange(1,len(A)):
    if A[i]-A[i-1]>0:
      delta=UP
    elif A[i]-A[i-1]<0:
      if delta==UP:
        maxima.append(A[i-1])
      delta=DOWN
```

```
def maxima2(A):
  UP, DOWN = 0,1
  delta = UP
  last_v = A[0]
  maxima = []

  for v in A[1:]:
    if v-last_v>0:
      delta=UP
    elif v-last_v<0:
      if delta==UP:
        maxima.append(last_v)
        delta=DOWN
```

```
def maxima3(A):
  UP, DOWN = 0,1
  delta = UP
  last_v = A[0]
  maxima = []

  for v in A[1:].tolist():
    if v-last_v>0:
      delta=UP
    elif v-last_v<0:
      if delta==UP:
        maxima.append(last_v)
      delta=DOWN
```

```
    return maxima
```

```
            last_v = v
    return maxima
```

```
            last_v = v
    return maxima
```

Table 10

The difference in execution speed however is surprising as maxima3 is more than 6 times faster than maxima2 and more than 8 times faster than maxima1:

	maxima1	maxima2	maxima3
|A| = 1000000	1.787s	1.393s	0.210s

Table 11

6.2 cython

Another way to utilize compiled code is to create a compiled library and invoke a function from that. Normally this approach would sacrifice two main reasons for using Python, namely the high-level language and cross-platform code.

Cython however provides a solution where normal python code can be translated to C-code, which can be compiled on the target machine and imported as a normal library.

The process requires the cython package, a setup script and a compiler.

The cython package is included in Winpython and Anaconda and accessible from the homepage http://cython.org/ or the Python Package Index.

The setup script can be generated automatically by the 'import_cython' script, see **Fejl! Henvisningskilde ikke fundet.**. Using this script the functions in 'cycheck.py' can be translated, compiled and imported by the following code:

```
from import_cython import import_cython
from cycheck import *
import_cython("cycheck")
```

Table 12

Obviously it takes a little time to compile a module the first it is imported, but afterward the 'import_cython' script just imports the compiled library.

If for some reason the compilation fails, 'import_cython' just imports the orgininal standard python module.

The speed up of obtained by compiling pure python code is not fantastic. The main reason is that the C-code still needs to determine the type of all python values before they can be stored in an appropriate type variable.

Therefore some type declaration is required in order to obtain execution speeds comparable to normal compiled programs. Fortunately cython provides a python compatible way to declare the type of local variables, such that the code can still be run as a normal python script in case the compilation fails.

Below two functions are shown. cycheck is similar to pcheckopt2 while three decorator lines are added above cycheck_opt.

The first line, `@cython.ccall`, speeds up the function call when invoked from other compiled functions by using the faster C calling conventions, while it is still callable from Python. (Especially important for sub functions)

The second line, `@cython.locals`, declares the type of the local variables. For possible types see http://docs.cython.org/src/tutorial/pure.html.

The third line, `@cython.returns` declares the type of the returned variable

```
def cycheck(n):
  if n==2 or n==3: return True
  if n%2==0 or n%3==0: return False
  for y in xrange(6,int(math.sqrt(n))+2,6):
    if n%(y-1)==0 or n%(y+1)==0:
      return False
  return True
```

```
@cython.ccall
@cython.locals(y=cython.int,n=cython.ulonglong)
@cython.returns(cython.bint)
def cycheck_opt(n):
  if n==2 or n==3: return True
  if n%2==0 or n%3==0: return False
  for y in xrange(6,int(math.sqrt(n))+2,6):
    if n%(y-1)==0 or n%(y+1)==0:
      return False
  return True
```

Table 13

As seen in the table below these three lines speeds up the execution about 40 times.

	pcheck_opt2	npcheck	cycheck	cycheck_opt	ccheck
n = 688846502588399	2.972s	0.551s	2.341s	0.059s	0.086
n = 688846502588398	0.000s	0.531s	0.000s	0.000s	0.000

Table 14

From the last column, it is seen that the cython translation and compilation of cycheck_opt is even faster than if the function was written directly in C:

```
int ccheck(long long n){
  int y;
  if (n==2 || n==3) return 1;
  if (n%2==0 || n%3==0) return 0;

  for (y=6; y<sqrt(n)+2; y+=6){
    if (n%(y-1)==0 || n%(y+1)==0){
      return 0;
    }
  }
  return 1;
}
```

Table 15

7 CPU parallelization

The maximum CPU speed has been almost constant around 3GHz for some years now. The problem is that the theoretical distance an electric signal can propagate in one clock cycle on a 3GHz computer is around, 10 cm, so all components must be packed into a very small area. This means that the heat dissipation from transistors changing state leads to a huge head concentration.

The solution has been to increase the number of CPU's, and utilize them in parallel.

CPU parallelization may be a good solution to speed up you programs, but most often it is not, so a good advice is to reconsider the previous mentioned methods as well as the number of cores in your computer (and your operating system).

Parallelization must be used with care as it critical issues, such as race conditions, deadlocks and synchronization overhead are easily introduced.

Subprocess

In python a subprocess can be created in which a new interpreter can be started. This is however an inelegant solution and there is no smart way to handle synchronization and data sharing.

Multiprocessing

`multiprocessing` is a package included in the standard python distribution. It provides an API that allows the user to create and use processes analogue to the use of threads in the `threading` module, see Table 16. It is easy to use and contains functionality to synchronize and share data between processes. It uses the efficient fork call on Unix systems and the much slower subprocess module on Windows which in practical tests uses about 0.2 seconds to create a process, see Figure 1, and processes cannot be reused for new tasks.

Parallel Python

Parallel Python, PP, is a light and easy integrable module that provides mechanisms for executing python code in parallel on SMP systems and clusters.

Parallel Python automatically detects computational resources and distributes jobs at runtime to benefit from dynamic load balancing. In addition it is fault-tolerant, i.e. if one node fails, tasks are rescheduled on other nodes.

It is easy to use for simple tasks, Table 16, but it may be painful to use for complicated tasks as all used modules must be stated manually.

Creating the job-server takes some time, but in contrast to the processes in `multiprocessing`, it only need to be created once.

PP must be installed manually as it is not included in the standard distribution, and moreover it has not been ported to Python 3 yet.

Comparison

Table 16 shows the time required to execute a time consuming CPU-bound method, `task`, two times on a dual core Windows pc. As seen Python 3.2 executes much faster than Python 2.7, and the context switching overhead of the threading module in Python 2.7 is totally avoided in Python 3.2.

In this example the speed-up achieved by two processes is worth the initialization cost and as seen the combination of `multiprocessing` and Python 3.2 provides the fastest solution while PP is fastest in Python 2.7.

Execution time, sec	Sequential	CPython threads	multiprocessing	Parallel Python, PP

	`def task():`
	`  n = xrange(5000000)`
	`  for i in n`
	`  sum = (i*i)`
	
	`def task_x2():`
	`  task()`
	`  task()`
		`from threading import Thread`
		`class TaskThread(Thread):`
		`  def run(self):`
		`  task()`
		
		`def taskThread_x_2 ():`
		`  t1 = taskThread()`
		`  t2 = taskThread()`
		`  t1.start(); t2.start()`
		`  t1.join();  t2.join()`
			`from multiprocessing import Process`
			
			`def multi_processing():`
			`  p1 = Process(target=task)`
			`  p2 = Process(target=task)`
			`  p1.start(); p2.start()`
			`  p1.join();  p2.join()`
				`import pp`
				`s = pp.Server()`
				
				`def taskPP_P():`
				`  t1=s.submit(task)`
				`  t2=s.submit(task)`
				`  t1()`
				`  t2()`

Python 2.7	5.84s	8.94s	3.46s	3.12s
Python 3.2	3.90s	3.89s	2.34s	Not supported

Table 16 – Time required for executing `task` two times

Figure 1 shows the executing times obtained by running 2 and 24 subtasks respectively, on a Unix system with 24 cores and Python 2.6. The experiment was repeated 20 times with increasing task sizes. In this simple example, the speed-up is amazing, but it requires task that are suited for parallelisation.

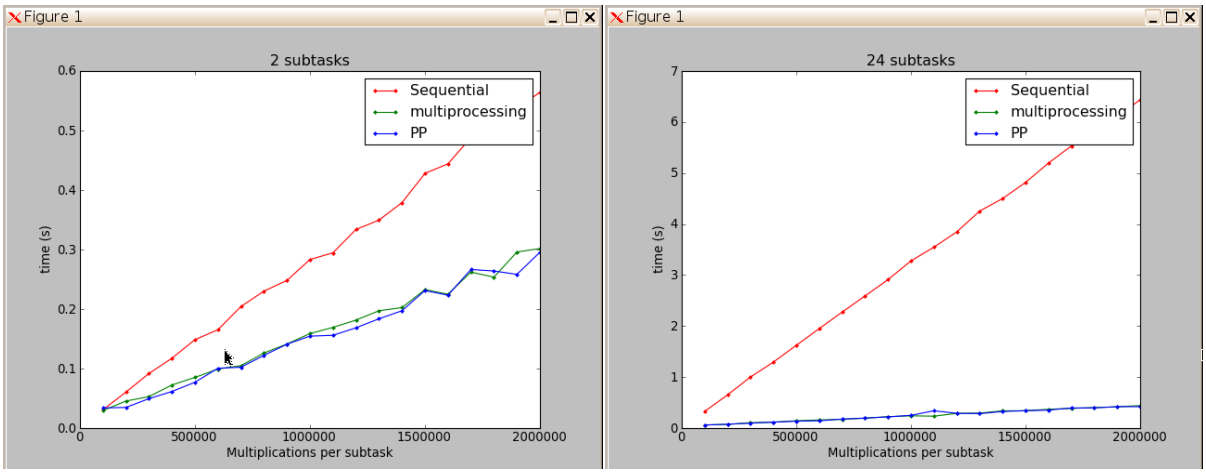

Figure 1 - Executing times obtained by running 2 and 24 subtasks respectively on a Unix system with 24 cores

8 GPU utilisation

Another way to accelerate execution is to utilize Graphic Processing Units, GPU. Modern graphic cards have plenty of computational resources and are well suited for executing hundreds of threads in parallel. Two frameworks exists, CUDA and OpenCL.

8.1 CUDA

CUDA, Compute Unified Device Architecture, is developed by Nvidia and can only be used on Nvidia graphic cards. CUDA is accessible in Python via the wrapper, PyCuda.

It is fairly easy to use, see Figure 2, where two random arrays of 1024 elements are multiplied. In order to utilize the GPU a C function, which multiplies one element of the arrays, is build and invoked by 1024 threads on the GPU and finally the result is compared to the standard numpy CPU result.

```
import numpy
import pycuda.autoinit
import pycuda.driver as drv
from pycuda.compiler import SourceModule

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

multiply_them = mod.get_function("multiply_them")


a = numpy.random.randn(1024).astype(numpy.float32)
b = numpy.random.randn(1024).astype(numpy.float32)
c = numpy.zeros_like(a)

multiply_them(drv.Out(c), drv.In(a), drv.In(b), block=(1024,1,1))

assert numpy.array_equal(c,(a*b))
```

Figure 2 – Multiplying two numpy arrays using PyCuda

In this case however, the GPU parallelisation speed-up is wasted on data transfer. In a test, the GPU version appeared to be 270 times slower than the CPU version.

The speed-up mainly depends on the number and types of computations, the size of input and output data, i.e. the amount of data to transfer, and of course the resources at the GPU.

Figure 3 shows the result of an experiment in which three different GPUs have been used to execute 1-30 multiplications(top row) and 1-30 trigonometric functions (bottom row) on each element of two arrays containing 16384 and 65536 random elements.

As seen the GPUs are faster than the CPU for more than 10 multiplications on the bigger array and a speed-up is seen on both arrays for a task consisting of just a few trigonometric functions. As trigonometric functions are heavily used at GPUs, they often contain dedicated hardware for these functions.

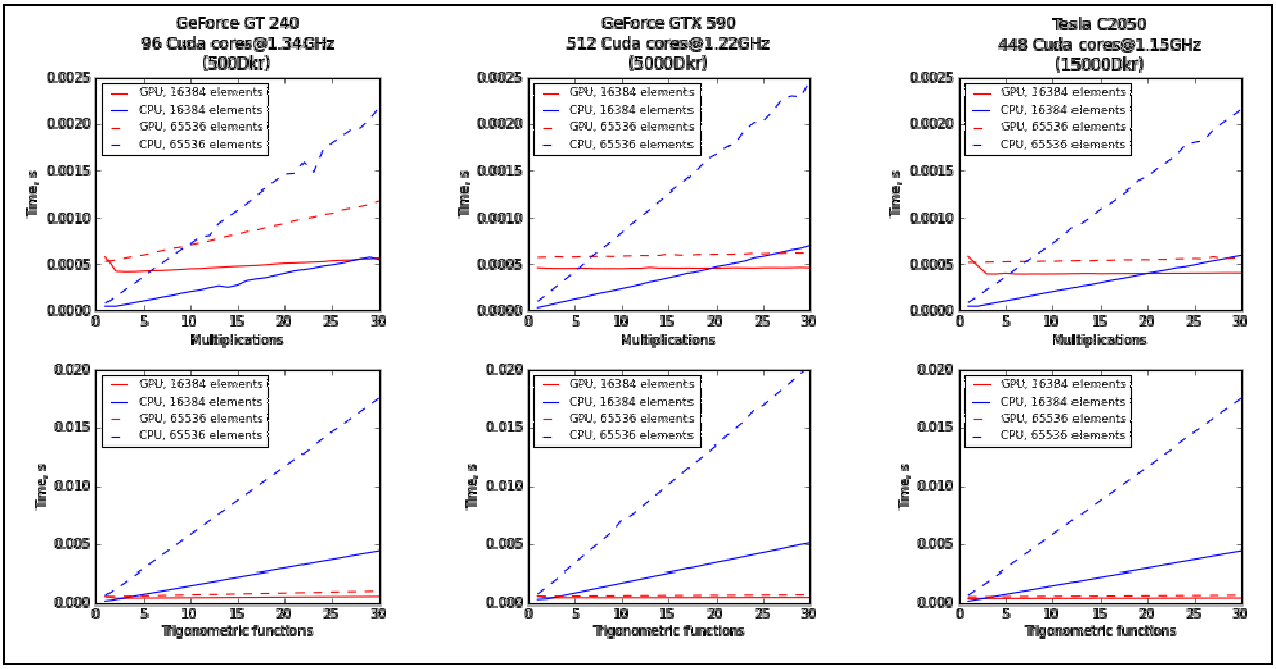

Figure 3 – Time used by three different CPU and GPUs to compute 1-30 multiplications(top row) or trigonometric functions (bottom row) on each element in two arrays of 16384 and 65536 random elements.

As seen it is possible to accelerate execution of a theoretical example using GPUs. It requires however a newer powerful graphic card, a parallelisation suited task which is computational complex and requires little input/output data and finally PyCuda depends on software that must be present on the target machine.

8.2 OpenCL

Open Computing Language, OpenCL, is a framework for writing programs for heterogeneous platforms, e.g. CPU and GPU. It is supported by Nvidia and AMD graphic cards. In addition OpenCL programs can execute on Intel, AMD and Apple CPUs. Similar to CUDA, OpenCL is available in Python via the PyOpenCL wrapper. Figure 4 shows and PyOpenCL implementation of the example in Figure 2.

```
import pyopencl as cl
import numpy

a = numpy.random.randn(1024).astype(numpy.float32)
b = numpy.random.randn(1024).astype(numpy.float32)
c = numpy.empty_like(a)

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)
c_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)

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

prg.mul(queue, a.shape, None, a_buf, b_buf, c_buf)
cl.enqueue_copy(queue, c, c_buf)
assert numpy.array_equal(c,(a*b))
```

Figure 4 - – Multiplying two numpy arrays using PyOpenCL

As seen the principle is similar to PyCuda and the capabilities, pros and cons are similar too, except that PyOpenCL is also supported by AMD/ATI GPUs.