SSE tutorial using Python and Weave module

Contents

Goals of this tutorial

1. test connection to computer server for running this and future tutorials
2. launch and modify python scripts using NumPy module
3. Learn how to run C++ code inside python using SciPy.Weave module
4. Optimize C++ code using SSE instructions and Intrinsics

Connect to tutorial compute server

if you don't have a INRIA Rennes account

- Go to this page : Connect to parallel server soroban

Else

- Go to this page for connect to igrida computer node connect to igrida gpu node

Very little introduction to **Python** and **NumPy** module

Python (http://www.python.org/) is a programing language defined in http://docs.python.org /2/tutorial/index.html as this: *Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python’s elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.* Python seems to be well adapted for tutorials on parallel computing thanks to modules like Scipy.Weave, PyCuda and PyOpenCL. We will work with python 2.7 version.

Launch a Python console in computer server

Python come with an interpreter console, you can launch it just typing :

```
$ python
Python 2.7.2+ (default, Jul 20 2012, 22:15:08)
[GCC 4.6.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 5+7
12
>>> exit()
$
```

IPython

There is a very powerful and comfortable interactive console named Ipython. Ipython allows you to access to command history, command complexion using [TAB] touch, help about command or objects using "?" or debugging scripts facilities. I recommended you to work with it. To launch Ipython console just tape:

```
$ ipython
Python 2.7.2+ (default, Jul 20 2012, 22:15:08)
Type "copyright", "credits" or "license" for more information.

IPython 0.10.2 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.
In [1]:
```

Ask for in-line documentation

```
In [1]: range?
Type:           builtin_function_or_method
Base Class:     <type 'builtin_function_or_method'>
String Form:    <built-in function range>
Namespace:      Python builtin
Docstring:
    range([start,] stop[, step]) -> list of integers
    Return a list containing an arithmetic progression of integers.
    range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!) defaults to 0.
    When step is given, it specifies the increment (or decrement).
    For example, range(4) returns [0, 1, 2, 3].  The end point is omitted!
    These are exactly the valid indices for a list of 4 elements.
```

Completion

Completion using [TAB] touch:

```
In [2]: list= ra
raise      range      raw_input
```

Using a variable and lists

Define a variable **my_list** that contents a list of integers:

```
In [2]: my_list= range(10)

In [3]: my_list
Out[3]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]


In [4]: list.reverse?
Type:           builtin_function_or_method
Base Class:     <type 'builtin_function_or_method'>
String Form:    <built-in method reverse of list object at 0x1070200>
Namespace:      Interactive
Docstring:
    L.reverse() -- reverse *IN PLACE*


In [5]: my_list.reverse()

In [6]: my_list
Out[6]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

In [7]: exit()
Do you really want to exit ([y]/n)? y
$
```

Write a python program

You can edit a python program using a file editor like vim, or gedit (graphic mode only). Python files have usually ".py" extension.

```
$gedit my_test.py
```

By default python files are interpreted as ASCII to files. But you can set another character coding mode by adding this kind of instruction at the begin of file (here is UTF-8 character sets):

```
# -*- coding: utf-8 -*-
```

Thats allow you to type char with French accents if you want. You can define comments with "#" char to the end of line :

```
a= 5 # this is a comment
```

or multi-line comments (in fact, triple-quoted strings) :

```
'''
This is a multiline
comment.
'''
```

Block syntax

Unlike C or C++, python don't need semi-colons to separate instructions. You can just separate instructions with line-feeds:

```
a=5
b=8
print "a=", a
print "b=", b
print "a+b=", a+b
```

And unlike C or C++, in Python instructions blocks aren't delimited by "{}" but using indentation:

```
if a> 2:
  c=1
  print "a is bigger that 2"
else:
  print "a is smaller or equal to 2"
  c=2
print "c=",c
```

Functions

You can define functions with parameters, and parameters with default values:

```
def my_function(x, y=0):
  a= 5*y + x**2
  return a
def my_function(x, y=0):
  a= 5*y + x**2
  return a

w= my_function(3)
z= my_function(3,1)
y= my_function(y=2, x=3)

print "(w,z,y)=",(w,z,y)
```

Run your program

You can run your program with python:

```
$ python my_test.py
a= 5
b= 8
a+b= 13
a is bigger that 2
c= 1
(w,z,y)= (9, 14, 19)
$
```

Inside ipython you can run your program with:

```
In [1]: %run my_test.py
a= 5
b= 8
a+b= 13
a is bigger that 2
c= 1
(w,z,y)= (9, 14, 19)
```

After your program run, you can watch state of variable or call methods:

```
In [2]: b
Out[2]: 8

In [3]: my_function(4)
Out[3]: 16
```

NumPy

NumPy is a module that allow you to efficiently manipulate array of data in high level with a optimized operation. You can find more information here : http://docs.scipy.org/doc/numpy /user/index.html

You can access to Numpy functions by importing module:

```
import numpy
```

Manipulate arrays

Create an array of float32

An empty linear array of 1000 elements:

```
a= numpy.empty(1000,numpy.float32)
```

A linear array of 1000 elements with all values set to **zero** :

```
a= numpy.zeros(1000,numpy.float32)
```

A linear array of 1000 elements with all values set to **one** :

```
a= numpy.ones(1000,numpy.float32)
```

A linear array of 1000 from with random values between 0 and 1:

```
a=numpy.random.rand(1000).astype(numpy.float32)
```

A linear array of 1000 from with linear from values 0 to 1000 :

```
a= numpy.arange(1000, dtype=numpy.float32)
```

Access to elements

Get the first element:

```
In [3]: a[0]
Out[3]: 0.0
```

Get the last:

```
In [4]: a[999]
Out[4]: 999.0
```

But in general way get the last element of any linear array:

```
In [5]: a[-1]
Out[5]: 999.0
```

Copy and reference of an array

Assignation by reference:

```
In [3]: b=a

In [4]: a[0]
Out[4]: 0.0

In [5]: b[0]=5

In [6]: b[0]
Out[6]: 5.0

In [7]: a[0]
Out[7]: 5.0
```

To do a copy (physical copy in memory)

```
In [2]: a=numpy.arange(1000, dtype=numpy.float32)

In [3]: b=a.copy()

In [4]: b[0]=5

In [5]: a[0]
Out[5]: 0.0
```

Views and slicing

Numpy allows you to get different views of an array using *slicing*. An Slice is create with index manipulation. For example:

```
In [2]: a=numpy.arange(10, dtype=numpy.float32)
In [3]: a
Out[3]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.], dtype=float32)

In [4]: a[0:9:2]
Out[4]: array([ 0.,  2.,  4.,  6.,  8.], dtype=float32)
```

Index 0:9:2 indicate de begin = 0 to the end = 9 with a step of 2.

Reverse Indexing

Step can be a negative integer:

```
In [5]: a[9:0:-2]
Out[5]: array([ 9.,  7.,  5.,  3.,  1.], dtype=float32)
```

in this case a[9:0:-2] is equivalent to a[-1:0:-2] Slice is a view of existent array. Is is not a copy of data.

Default Index

It is possible to take default values for begin or end index:

```
In [6]: a[::-2]
Out[6]: array([ 9.,  7.,  5.,  3.,  1.], dtype=float32)
```

Or for step:

```
In [7]: a[:]
Out[7]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.], dtype=float32)
```

```
In [8]: a[5:]
Out[8]: array([ 5.,  6.,  7.,  8.,  9.], dtype=float32)
```

```
In [9]: a[:5]
Out[9]: array([ 0.,  1.,  2.,  3.,  4.], dtype=float32)
```

Array Operators

Scalar operation

Scalar operation with a array is equivalent to apply the operation between the scalar and every element of the array:

```
In [1]: import numpy

In [2]: a=numpy.arange(10, dtype=numpy.float32)

In [3]: a*4
Out[3]: array([  0.,   4.,   8.,  12.,  16.,  20.,  24.,  28.,  32.,  36.], dtype=float32)

In [4]: 5+a
Out[4]: array([  5.,   6.,   7.,   8.,   9.,  10.,  11.,  12.,  13.,  14.], dtype=float32)
```

Array Vs. Array operation

Binary operations like "*" "+" are do in element-wise way :

```
In [1]: import numpy

In [2]: a=numpy.arange(10, dtype=numpy.float32)

In [3]: a*4
Out[3]: array([  0.,   4.,   8.,  12.,  16.,  20.,  24.,  28.,  32.,  36.], dtype=float32)

In [4]: 5+a
Out[4]: array([  5.,   6.,   7.,   8.,   9.,  10.,  11.,  12.,  13.,  14.], dtype=float32)

In [5]: a+a
Out[5]: array([  0.,   2.,   4.,   6.,   8.,  10.,  12.,  14.,  16.,  18.], dtype=float32)

In [6]: a*a
Out[6]: array([  0.,   1.,   4.,   9.,  16.,  25.,  36.,  49.,  64.,  81.], dtype=float32)

In [7]: c=-a

In [8]: a*c
Out[8]: array([ -0.,  -1.,  -4.,  -9., -16., -25., -36., -49., -64., -81.], dtype=float32)
```

Matrix dot product must be called explicitly :

```
In [15]: numpy.dot(a,a)
Out[15]: 285.0
```

Using SciPy.Weave to integrate C/C++ code in python

SciPy.Weave is a module that allow in-line inclusion of C/C++ code in python using code generation and compilation on the fly. see https://github.com/scipy/scipy/blob/master/scipy/weave/doc/tutorial.txt for more information.

Hello world with Weave

Imagine a C++ function that take a string parameter "text" and print a message :

```
void Hello( std::string text)
{
 std::cout << "Hello "<< text << std::endl;
}
```

We can call a similar function in python with this code:

```
In [1]: import scipy.weave

In [2]: text='World'

In [3]: scipy.weave.inline('std::cout << "Hello "<< text << std::endl;',['text'])
Hello World
```

At first call, generator and compiler run a produce a library that will be call on future calls

First argument of scipy.weave is a string that contents de code to be compiled, equivalent to inside par of a function. This code can be a multi-line string like:

code = std::cout << "Hello "<< text << std::endl; float a=0; std::cout << "a=" << a << std::endl;

When run:

```
In [5]: scipy.weave.inline(code,['text'])
Hello World
a=0
```

Second argument is a list of arguments names for inline function.

Function with two arguments and a return value

See next example:

```
code = '''
return_val=sqrt(x*x+y*y);
'''
```

And run with:

```
In [17]: x=2.0

In [18]: y=3.0

In [19]: scipy.weave.inline(code,['x','y'])
Out[19]: 3.605551242828369
```

Here the specific "return_val" term replace the function return.

Passing NumPy Arrays as arguments in read/write mode

Passing a NumPy array to a inline code in C/C++ is equivalent to pass pointers of array in C++:

```
import scipy.weave
import numpy
size=10
a=numpy.arange(size, dtype=numpy.float32)
b=numpy.arange(size, dtype=numpy.float32)
c=numpy.empty_like(a)
code = '''
for(unsigned int i=0; i < size ; i++)
{
  c[i]=a[i]+b[i];
}'''

scipy.weave.inline(code,['a','b','c','size'])
```

The result is :

```
In [12]: scipy.weave.inline(code,['a','b','c','size'])

In [13]: c
Out[13]: array([  0.,   2.,   4.,   6.,   8.,  10.,  12.,  14.,  16.,  18.], dtype=float32)
```

Here inside the C++/C code 'c' variable is a type float*. Also 'a' and 'b' are float*

Optimization of a C++ code with SSE instructions

It's time to integrate NumPy and ScyPy.Weave to have program a benchmark for optimization tests.

Benchmark code

This code define a benchmark for test a classical procedure in linear algebra packages : SAXPY in a particular case. This code use others optional arguments in call of scipy.weave :

- **headers** : integrate additional includes
- **extra_compile_args** : Compilation arguments to allow to use SSE and OpenMP instructions
- **extra_link_args** : link arguments to allow to use SSE and OpenMP instructions
- **verbose** : to display compilation complete command
- **support_code**: to add additional functions used in principal code

```
import numpy
from time import time
from scipy import weave
sizeX = 1000000
numberIterations =1000
X = numpy.random.rand(sizeX).astype(numpy.float32)
Y = numpy.empty(sizeX).astype(numpy.float32)

# C++ reference code
referenceCode="""
#line 11 "saxpy.py" // helpful for debug
void saxpy(int n, float alpha, float *X, float *Y)
{
        int i;
        for (i=0; i<n; i++)
                Y[i] += alpha * X[i];
}
"""
mainCode="""

for(int j=0; j< numberIterations;j++)
  saxpy(sizeX, 0.001f, X, Y);
"""

def BenchmarkCode(name, mainCode, saxpyCode,X,Y):
        # init
        Y[:]=0.5
        X[:]=1.0
        # start chronometer
        start_time = time()
        #use this to compile directly C++ code.
        # fix for bug http://projects.scipy.org/scipy/ticket/1143 adding support code as comment in mai
        weave.inline(mainCode+"/* " + saxpyCode + " */" , ['Y','X','sizeX', 'numberIterations'],
                                 compiler = 'gcc', support_code=saxpyCode,
                                 verbose=2,
                         extra_compile_args=['-msse2 -fopenmp'],
                         extra_link_args=['-msse2 -fopenmp'],
                         headers=['<xmmintrin.h>'])
        # stop chronometer
        stop_time = time()
        execution_time= stop_time - start_time
        print("execution time for "+name+" code = "+ str(execution_time))
        return execution_time


referenceTime=BenchmarkCode('Reference', mainCode, referenceCode,X,Y)

SSECode="""
#line 49 "saxpy.py" // helpful for debug
void saxpy(int n, float alpha, float *X, float *Y)
{
        int i;
        for (i=0; i<n; i++)
                Y[i] += alpha * X[i];
}
"""

SSETime=BenchmarkCode('SSE', mainCode, SSECode,X,Y)

print("speed up for SSE = " + str(referenceTime/SSETime))
```

SAXPY

SAXPY is a classical linear algebra routine that take **X** and **Y** arrays of float in parameters an produce an a output Y = Y + alpha *X. Where **alpha** is a scalar input parameter. In this tutorial we are interesting in computation performance of SAXPY in case where we need to cumulate 1000 iterative call of this function:

```
for(int j=0; j< numberIterations;j++)
  saxpy(sizeX, 0.001f, X, Y);
```

SSE version of SAXPY

In benchmark, we have a reference code of the function SAXPY in **referenceCode** string and a code to be changed in **SSECode**.

1. Modify SSECode to compute SAXPY using SSE instructions
2. test this version and compare speed ratio between reference code and SSECode
3. Draw a schema of memory transfer at every call of SSE code and global call in main code
4. Modify main code to integrate for loop for(int j=0; j< numberIterations;j++) inside SAXPY implementation
5. Change order of loops to determine best memory schema access
6. Measure corresponding acceleration (Speed-Up) in relation with reference code
7. Verify that results are identical for reference function and your code.

Go back to Parallel Computing Tutorials