Computer Science & Information Technology

David C. Wyld
Natarajan Meghanathan
Dhinaharan Nagamalai

Third International Conference on Computer Science, Engineering & Applications (ICCSEA 2013)

Delhi, India. May 2013.

Proceedings

Computer Science Conference Proceedings
AIRCC

OpenCL programming using Python syntax

Massimo Di Pierro

School of Computing, DePaul University, Chicago IL 60604, USA

ABSTRACT

We describe ocl, a Python library built on top of pyOpenCL and numpy. It allows programming GPU devices using Python. Python functions which are marked up using the provided decorator, are converted into C99/OpenCL and compiled using the JIT at runtime. This approach lowers the barrier to entry to programming GPU devices since it requires only Python syntax and no external compilation or linking steps. The resulting Python program runs even if a GPU is not available. As an example of application, we solve the problem of computing the covariance matrix for historical stock prices and determining the optimal portfolio according to Modern Portfolio Theory.

1. INTRODUCTION

Compatibly with Moore's Law, the number of transistors on integrated circuits continues to double every 18 months. This has resulted in modern CPUs having multiple computing cores. To date, a state of the art AMD Opteron 8380 hosts 32 computing cores. A Nvidia Tesla GPU K20X hosts 2688 CUDA cores. A modern desktop computer can host multiple CPUs and GPUs. Programming such heterogeneous architectures is a challenge. Traditional paradigms for programming parallel machines are not adequate to handle large number of cores.

Modern programming languages have proposed various solutions. For example Clojure uses transactional memory to simplify multi-threaded programming. Erlang and Go uses lightweight threads that communicate via message passing. Haskell uses a library of distributed data structures (REPL) optimized for different types of architectures. Yet they are not as fast as C code (except possibly for Go), scale but only up to a point, and they are not designed to work on GPUs. Nvidia has proposed the CUDA framework for programming GPUs and it remains the best solution for programming Nvidia devices. Unfortunately it does not work on regular CPUs.

The Kronos Consortium supported by Nvidia, AMD, Intel, and ARM, has developed the Open Common Language framework (OpenCL) which has many similarities with CUDA, but promises more portability and support for different architectures including Intel/AMD CPUs, Nvidia/ATI GPU, and ARM chips such as those used on mobile phones.

Both CUDA and OpenCL programs are divided into two parts. A host program usually written in C or C++ and one or more kernels written in a C-like language. The host program compiles the kernels and runs them on the available devices (the GPUs in the case of CUDA and GPUs or CPUs in the case of OpenCL). The host program also deals with loading and saving operations between the main memory and the computing devices. The OpenCL language is very close to

David C. Wyld (Eds) : ICCSEA, SPPR, CSIA, WimoA - 2013
pp. 57–66, 2013. © CS & IT-CSCP 2013

DOI : 10.5121/csit.2013.3506

C99 while the CUDA dialect is more powerful and, for example, supports templates.

A major complexity in writing CUDA and OpenCL code consists in having to write code inside code (the kernel managed by the host). This process can be somewhat simplified using a different language for the host. For example, the pyCUDA and pyOpenCL libraries created by Andreas Klöckner[3] allow compiling, queuing, and running kernels (in CUDA and OpenCL respectively) from inside a Python program. They are integrated with numpy[4], the Python numeric library.

In this paper we discuss a library called ocl which is built on top of pyOpenCL. It allows writing both the host program and the kernels using the Python languages without loss of performance compared to native OpenCL. ocl consists of two parts:

- A thin layer on top of pyOpenCL which encapsulates the device state (consisting of a context and a task queue).
- A function decorator which, at runtime, converts decorated Python functions into C99 code and uses the OpenCL JIT to compile them.

Python + numpy + pyOpenCL + ocl allow development of parallel programs which run everywhere (CPUs and GPUs), are written in a single language, and do not require any outside compilation, linking, or other building step.

ocl is similar to Cython[8] and Clyther[9] in the sense that it works by parsing the Python code into a Python Abstract Syntax Tree (AST) and then serializing the AST into C99 code. There are differences between Clyther and ocl: The implementation is different. The semantic used to build the kernel body is different and closer to C in the ocl case. Moreover, the ocl implementation is extensible and, in fact, it can be used to generate and compile C code (without the need for OpenCL, like Cython) but it can also be used to generate JavaScript code (to run, for example, in a browser). Javascript code generation is beyond the scope of this paper since its application is in web development, not high performance computing.

In order to explain the syntax of ocl we will consider first the simple example of adding two vectors and then a more complex financial example of computing the correlation matrix for arithmetic return of multiple time series and determine the optimal portfolio according to Modern Portfolio Theory.

2. Simple Example

In our first example we consider the following Python code which defines two arrays, fills them with random Gaussian floating numbers, and adds them using the available devices.

```
from ocl import Device
import numpy
import random

n = 1000000
a = numpy.zeros(n, dtype=numpy.float32)
b = numpy.zeros(n, dtype=numpy.float32)

for k in xrange(n):
    a[k] = random.gauss(0,1)
```

```
    b[k] = random.gauss(0,1)

device = Device()
a_buffer = device.buffer(source=a, mode=device.flags.READ_ONLY)
b_buffer = device.buffer(source=b)

@device.compiler.define_kernel(a='global:const:ptr_float',
                               b='global:ptr_float')
def add(a, b):
    k = new_int(get_global_id(0))
    b[k] = a[k]+b[k]

program = device.compile()
program.add(device.queue, [n], None, a_buffer, b_buffer)
b = device.retrieve(b_buffer,shape=(n,))
```

In this example:

- Lines 1-3 import the requires modules.
- Lines 6-7 define two numpy arrays (a and b) of size n in the main RAM memory and fill them with zeros.
- Lines 9-11 populate the arrays a and b with random Gaussian numbers with mean 0 and standard deviation 1.
- Line 13 determines the available device and incorporates its state.
- Lines 14-15 copies the arrays a and b into the memory of the device represented by a_buffer and b_buffer respectively.
- Lines 17-21 define a new kernel called "add".
- Line 17-18 decorate the "add" function to specify it is a kernel and needs to be converted into C99 code and declare the types of the function arguments.
- Lines 19-21 define the body of the kernel.
- Line 20 runs on the device (one instance per thread) and on each thread returns a different value for k.
- Line 23 converts and compiles the kernel, and loads it into the device.
- Line 24 calls the function "add". More precisely it queues [n] threads each calling the function "add" and determines that a_buffer and b_buffer are to be passed as arguments.
- Line 25 retrieves the output array b_buffer from the device and copies it into b.

The program above is run with a single Bash shell command:

```
python myprogram.py
```

Notice how the kernel function "add" is preceded by the decorator

```
@device.compiler.define_kernel(...)
```

This decorator performs introspection of the function at runtime, parses the body of the function into an Abstract Syntax Tree (AST) for later conversion to C99/OpenCL code. This allows us to write OpenCL code using Python syntax.

Supported control structures and commands include def, if..elif else, for ... range, while, break, and continue.

The decorated code is converted at runtime into the following Python AST

```
FunctionDef(name='add',
    args=arguments(args=[Name(id='a'), Name(id='b', ctx=Param())]),
    body=[Assign(targets=[Name(id='k')],
                 value=Call(func=Name(id='new_int'),
                            args=[Call(func=Name(id='get_global_id'),
                                       args=[Num(n=0)])]),
          Assign(targets=[Subscript(value=Name(id='b'),
                                    slice=Index(value=Name(id='k'))],
                 value=BinOp(left=Subscript(value=Name(id='a'),
                                    slice=Index(value=Name(id='k'))),
                             op=Add(),
                             right=Subscript(value=Name(id='b'),
                                  slice=Index(value=Name(id='k'))))),
          Return(value=Name(id='None'))])
```

(we simplified the AST by omitting irrelevant parameters).

The call to device.compile(...) converts the AST into the following code:

```
__kernel void add(__global const float* a, __global float* b) {
    int k;
    k = get_global_id(0);
    b[k] = (a[k] + b[k]);
}
```

__kernel is an OpenCL modifier which declares the function to be a kernel. __global and __local are used to declared the type of memory storage: global of the device or local of the core, respectively.

One complication is that C99/OpenCL is a statically typed language while Python is only strongly typed. Therefore one needs a way to declare the type of variables using Python syntax. For the function arguments this is done in the decorator arguments. For example:

```
a = "global:const:ptr_float"
```

is converted into

```
__global const float *a
```

The type of local variables is defined using the *pseudo-casting* operators new_int, new_float, etc. For example:

```
k = new_int(get_global_id(0))
```

is converted into

```
int k;
k = get_global_id(0);
```

Variable types must be declared when first assigned. The converter checks that this is the case to

prevent further and more obscure compile time errors.

In this example, get_global_id(0) is an OpenCL function user to identify the rank of the current running instance of the kernel. If one queues [n] kernel tasks, it will return all values from 0 to $n-1$, a different value for each running task. The order in which queued tasks are executed is not guaranteed.

The return type of a function is determined automatically but one must return a variable (or None), not an expression.

Other variable types and pointers can be defined and manipulated using the syntax shown in the following table:

ocl	C99/OpenCL
x = new_type(...)	type x = ...;
x = new_prt_type(...)	type *x = ...;
x = new_prt_prt_type(...)	type **x = ...;
ADDR(x)	&x
REFD(x)	*x
CAST(prt_type,x)	(type*)x

OpenCL provides special types of variables to take advantages of vector registries. For example float4 can store four floating point numbers called x, y, z, and w. These special vector types can be used from ocl as follows:

```
q = new_float4((0.0,0.0,0.0,0.0))
q.x = 1.0
```

Notice that the goal of ocl is not to translate arbitrary Python code into C99/OpenCL but to allow the use of Python syntax to write OpenCL kernels and functions. Only C99/OpenCL variable types are allowed, not Pyhton types such as lists and dictionaries. This may be supported in future versions of ocl. Moreover all functions called in the Python code which constitutes the body of the add function are intended to be OpenCL functions not Python functions. While Python code is normally garbage collected, the decorated code runs on the device as C code and it is not garbage collected therefore explicit memory management may be necessary:

```
@device.compiler.define_kernel(...)
def some_kernel(...):
    d = new_ptr_float(malloc(size))
    ...
    free(d)
```

which would be converted into:

```
__kernel void some_kernel(...):
    float* d;
    d = (float*)malloc(size);
    ...
```

```
    free(d);
}
```

One can define multiple kernels within the same program and call them when needed. One can use the @device.compiler.define(...) decorator to define other functions to be executed on the device. They will be visible and callable by the kernels, but they will not be callable from the host program.

One can pass constants from the Python context to the OpenCL context:

```
device.compile(..., constants = dict(n = m))
```

where n is a constant in the kernel and m is a variable in the host program.

Additional C files can be included using the syntax

```
device.compile(..., inlcudes = ['#include <math.h>'])
```

where includes is a list of #include statements.

3. FINANCIAL APPLICATION EXAMPLE

In the following example, we solve the typical financial problem of analyzing multiple time series such as stock prices, p[k,t] (simulated stock prices). For each time series k we compute the arithmetic daily returns, r[k,t], and the average returns, mu[k]. We then compute the covariance matrix, cov[i,j], and the correlation matrix, cor[i,j]. We use different kernels for each part of the computation.

Finally, to make the application more practical, we use Modern Portfolio Theory [10] to compute a tangency portfolio which maximizes the Sharpe ratio, *i.e.* return/risk under the assumption of normal returns:

Here μ is the vector of average returns (mu), Σ is the covariance matrix (cov), and is the input risk free interest rate. The tangency portfolio is identified by the vector x (aka array x in the code) whose terms indicate the amount to be invested in each stock (must add up to one dollar). We perform this maximization on the CPU to demonstrate integration with the numpy Linear Algebra package.

We use the symbols i and j to identify the stock time series, and the symbol t for time (for daily data t is a day). n is the number of stocks and m is the number of trading days.

We use the canvas[6] library, based on the Python matplotlib library, to display one of the stock price series and the resulting correlation matrix. Below is the complete code. The output from the code can be seen in fig. 1.

```
from ocl import Device
from canvas import Canvas
import random
```

```
import numpy
from math import exp

n = 1000 # number of time series
m = 250  # number of trading days for time series
p = numpy.zeros((n,m), dtype=numpy.float32)
r = numpy.zeros((n,m), dtype=numpy.float32)
mu = numpy.zeros(n, dtype=numpy.float32)
cov = numpy.zeros((n,n), dtype=numpy.float32)
cor = numpy.zeros((n,n), dtype=numpy.float32)

for k in xrange(n):
    p[k,0] = 100.0
    for t in xrange(1,m):
        c = 1.0 if k==0 else (p[k-1,t]/p[k-1,t-1])
        p[k,t] = p[k,t-1]*exp(random.gauss(0.0001,0.10))*c

device = Device()
p_buffer = device.buffer(source=p, mode=device.flags.READ_ONLY)
r_buffer = device.buffer(source=r)
mu_buffer = device.buffer(source=mu)
cov_buffer = device.buffer(source=cov)
cor_buffer = device.buffer(source=cor)

@device.compiler.define_kernel(p='global:const:ptr_float',
                               r='global:ptr_float')
def compute_r(p, r):
    i = new_int(get_global_id(0))
    for t in range(0,m-1):
        r[i*m+t] = p[i*m+t+1]/p[i*m+t] - 1.0

@device.compiler.define_kernel(r='global:ptr_float',
                               mu='global:ptr_float')
def compute_mu(r, mu):
    i = new_int(get_global_id(0))
    sum = new_float(0.0)
    for t in range(0,m-1):
        sum = sum + r[i*m+t]
    mu[i] = sum/(m-1)

@device.compiler.define_kernel(r='global:ptr_float',
      mu='global:ptr_float', cov='global:ptr_float')
def compute_cov(r, mu, cov):
    i = new_int(get_global_id(0))
    j = new_int(get_global_id(1))
    sum = new_float(0.0)
    for t in range(0,m-1):
        sum = sum + r[i*m+t]*r[j*m+t]
    cov[i*n+j] = sum/(m-1)-mu[i]*mu[j]
```

```
@device.compiler.define_kernel(cov='global:ptr_float',
                               cor='global:ptr_float')
def compute_cor(cov, cor):
    i = new_int(get_global_id(0))
    j = new_int(get_global_id(1))
    cor[i*n+j] = cov[i*n+j] / sqrt(cov[i*n+i]*cov[j*n+j])

program = device.compile(constants=dict(n=n,m=m))

q = device.queue
program.compute_r(q, [n], None, p_buffer, r_buffer)
program.compute_mu(q, [n], None, r_buffer, mu_buffer)
program.compute_cov(q, [n,n], None, r_buffer, mu_buffer, cov_buffer)
program.compute_cor(q, [n,n], None, cov_buffer, cor_buffer)

r = device.retrieve(r_buffer,shape=(n,m))
mu = device.retrieve(mu_buffer,shape=(n,))
cov = device.retrieve(cov_buffer,shape=(n,n))
cor = device.retrieve(cor_buffer,shape=(n,n))

points = [(x,y) for (x,y) in enumerate(p[0])]
Canvas(title='Price').plot(points).save(filename='price.png')
Canvas(title='Correlations').imshow(cor).save(filename='cor.png')

rf = 0.05/m # input daily risk free interest rate
x = numpy.linalg.solve(cov,mu-rf) # cov*x = (mu-rf)
x *= 1.00/sum(x) # assumes 1.00 dollars in total investment
open('optimal_portfolio','w').write(repr(x))
```

Notice how the memory buffers are always 1-dimensional therefore the i,j indexes have to be mapped into a 1D index i*n+j. Also notice that while kernels compute_r and compute_mu are called [n] times (once per stock k), kernels compute_cov and compute_cor are called [n,n] times, once per each couple of stocks i,j. The values of i,j are retrieved by get_global_id(0) and (1) respectively.

In this program we have defined multiple kernels and complied them at once. We call one kernel at the time to make sure that the call to the previous kernel is completed before running the next one.

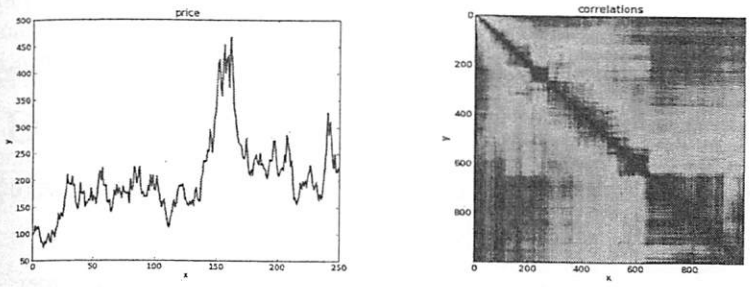

Figure 1: The image on the left shows one of the randomly generated stock price histories. The image on the right represents the computed correlation matrix. Rows and columns corresponds to stock returns and the color at the intersection is their correlation (red for high correlation and blue for no correlation). The resulting shape is an artifact of the algorithm used to generate random data.

4. OCL WITHOUT OpenCL

ocl can be used to generate C99 code, compile it, and run it without OpenCL. In this case the code always runs on the CPU and not on the GPU. Here is a simple example:

```
from ocl import Compiler
c99 = Compiler()

@c99.define(n='int')
def factorial(n):
    output = 1
    for k in range(1, n + 1):
        output = output * k
    return output
compiled = c99.compile()
print(compiled.factorial(10))
assert compiled.factorial(10) == factorial(10)
```

In this example, we have replaced device.compiler with c99 = Compiler() since "device" is an OpenCL specific concept. We have been careful to engineer the function "factorial" so that it can run both in Python (factorial) and compiled in C99 (compiled.factorial). This is not always possible but it possible for functions that only call mathematical libraries, such as our factorial function. Notice that c99.compile() requires the presence of gcc installed instead of the OpenCL JIT.

5. CONCLUSIONS

In this paper we provide an introduction to ocl, a library that allows writing OpenCL programs and kernels using exclusively Python syntax. ocl is built on top of pyOpenCL[2], meta[7], and numpy[4] (the Python Scientific Libraries). The Python code is converted to C99/OpenCL and compiled at run-time. It can run on any OpenCL compatible device including ordinary Intel/AMD CPUs, Nvidia/ATI GPUs, and ARM CPUs with the same performance as native OpenCL code.

Our approach does not necessarily simplifies the algorithms but it makes them more readable, portable, and eliminates external compilation and linking steps thus lowering the barrier of entry to GPU programming. It does not eliminate the need to understand OpenCL semantic but allows the use of a simpler syntax.

As an example of application we considered the computation of the covariance and correlation matrices from financial data series, in order determine the optimal portfolio according to Model Portfolio Theory [10].

We plan to extend ocl to support a richer syntax, generate CUDA as well as OpenCL code, and provide better debugging information. Moreover, since Python code can easily be serialized and deserialized at runtime, we plan to improve the library to allow distributed computations where the code is written once (in Python), distributed, then compiled and ran on different worker nodes using heterogenous devices.

ACKNOWLEDGEMENTS

I thank Andreas Klöckner for pyOpenCL and the excellent documentation.

REFERENCES

[1] Lindholm, Erik, et al. "NVIDIA Tesla: A unified graphics and computing architecture." Micro, IEEE 28.2 (2008): 39-55.
[2] Munshi, Aaftab. "OpenCL: Parallel Computing on the GPU and CPU." SIGGRAPH, Tutorial (2008).
[3] Klöckner, Andreas, et al. "Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation." Parallel Computing 38.3 (2012): 157-174.
[4] Oliphant, Travis E. A Guide to NumPy. Vol. 1. USA: Trelgol Publishing, 2006.
[5] https://github.com/mdipierro/ocl
[6] https://github.com/mdipierro/canvas
[7] http://srossross.github.com/Meta/html/index.html
[8] Behnel, Stefan, et al. "Cython: The best of both worlds." Computing in Science & Engineering 13.2 (2011): 31-39.
[9] http://srossross.github.com/Clyther/
[10] Markowitz, Harry M. "Foundations of portfolio theory." The Journal of Finance 46.2 (2012): 469-477.