Portable Parallel Programs with Python and OpenCL

Massimo Di Pierro | DePaul University

Open Common Language (OpenCL) runs on multicore GPUs, as well as other architectures including ordinary CPUs and mobile devices. Combining OpenCL with numerical Python (numPy) and a new module—ocl, a Python-to-C converter that lets developers use Python to write OpenCL kernels—creates a powerful framework for developing efficient parallel programs for modern heterogeneous architectures.

In the past five years, consumer CPU clock speed has stalled, hitting a practical limit of 3 GHz. Beyond that limit, cooling the CPU becomes impractical. CPU computing power has nonetheless continued to grow thanks to an increased number of computing cores per CPU. At the same time, we've seen the emergence of heterogeneous architectures, in which the same device can host multiple CPUs and GPUs. To date, a state-of-the-art AMD Opteron 8380 hosts 32 computing cores and an Nvidia Tesla GPU K20X hosts 2,688 CUDA cores. Programming these devices can be a challenge, as traditional programming languages are not well equipped to deal with this level of concurrency.

In our guest editors' introduction for the November/December 2012 issue of *CiSE*, David Skinner and I gave examples of how some modern programming languages—such as Clojure, Erlang, and Haskell—tackle the problem of scaling on multiple cores.[1] Clojure uses transactional memory to simplify multithreaded programming. Erlang uses lightweight threads that can communicate only via message passing. Haskell uses a library of distributed data structures (a read-eval-print loop, or *REPL*) optimized for different types of architectures. However, these approaches aren't as fast as C code, and they aren't designed to work on GPUs. Nvidia developed the CUDA framework specifically for its GPUs.[2] CUDA consists of a C-like programming language that developers can use to write computing kernels, which are compiled and executed on the GPU cores in a multithreaded-like fashion (although CUDA threads are managed differently than regular threads). CUDA code can also target LLVM (formerly Low-Level Virtual Machine; see http://llvm.org), while MCuda[3] enables CUDA to run on multicore CPUs.

The Kronos Consortium—supported by Nvidia, Advanced Micro Devices (AMD), Intel, and ARM—has developed the Open Common Language framework. OpenCL[4] borrows many ideas from 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. OpenCL consists of a C99 programming dialect. Similar to CUDA, OpenCL programs define kernels that are queued to be executed in parallel on available devices. Kernels running on the same device have access to a shared memory area, as well as local memory areas. OpenCL provides an API that performs loading and saving operations between main memory and device memory. The operations of loading and saving data, and queuing and running kernels, are performed by a host program. This program is usually written in C/C++, but it's also possible to run CUDA and OpenCL kernels from a host program written in other languages, such as Erlang and Python.

Here, I focus on running OpenCL from Python using a library called *pyOpenCL*, created by Andreas Klöckner.[5] I'll also discuss a Python library, called *ocl*, that I developed, which allows just-in-time (JIT) conversion of Python code into OpenCL code. This approach has no overhead because, once the code is converted, it runs as native OpenCL code. Because the Python ocl module performs a conversion purely at the syntactical level—not at the semantic level—any handwritten OpenCL code can be expressed using the corresponding Python syntax without loss of performance. I don't include benchmarks here, because for any handwritten OpenCL code, I would be able to write Python code that generates the same exact target OpenCL code (including the same variable names). However, I have included examples from this article in the online GitHub repo for ocl so you can easily try them (see https://github.com/mdipierro/ocl).

Ocl isn't based—but takes inspiration from—the Cython library,[6] which coverts Python modules into C code.

A First Example with PyOpenCL

To better understand how it works, I should mention that pyOpenCL is built on top of the Python library for efficient array manipulations (that is, numerical Python, or *NumPy*) and OpenCL. It lets programmers embed OpenCL code into a Python program in the form of a string containing the kernel code. It also provides APIs to load a numPy array on a computing device (GPU or CPU), queue and run the kernel, and retrieve the computation results. In pyOpenCL, object cleanup is tied to the lifetime of objects, which makes it easier to write correct leak- and crash-free code. Moreover, OpenCL errors are automatically translated into Python exceptions.

In the following examples, I won't use pyOpenCL directly, but rather the abstraction layer provided by the ocl library. Here's an example of a simple Python program that we want to parallelize:

```
import numpy

n = 50000
a = numpy.random.rand(n).astype(numpy.float32)
b = numpy.random.rand(n).astype(numpy.float32)
c = numpy.zeros(n,dtype=numpy.float32)

for i in range(0, n):
      c[i] = a[i] + b[i];

assert numpy.linalg.norm(c - (a + b)) == 0
```

```
from ocl import Device
import numpy

n = 50000
a = numpy.random.rand(n).astype(numpy.float32)
b = numpy.random.rand(n).astype(numpy.float32)

device = Device()
a_buffer = device.buffer(source=a)
b_buffer = device.buffer(source=b)
c_buffer = device.buffer(size=b.nbytes)

kernels = device.compile("""
 __kernel void sum(__global const float *a, /* a_buffer */
                   __global const float *b, /* b_buffer */
                   __global float *c) {  /* c_buffer */
 int i = get_global_id(0); /* thread id */
 c[i] = a[i] + b[i];
 }
 """)

kernels.sum(device.queue,[n],None,a_buffer,b_buffer,c_buffer)
c = device.retrieve(c_buffer)

assert numpy.linalg.norm(c - (a+b)) == 0
```

Figure 1. Scalar product using pyOpenCL.

The program creates three numPy arrays (`a,b,c`), and fills the first two with random numbers and the latter with zeros. It then loops and adds `a[i]+b[i]` into `c[i]`. Finally, it uses `numpy.linalg` (the linear algebra module) to verify that the norm of `c-(a+b)` is zero.

The "computing intensive" or kernel part of this program is the `for` loop, which we want to parallelize into a kernel (see Figure 1).

This program is similar to the previous one, and they have many lines in common; however, it also does several new things:

- defines a device object, which encapsulates the OpenCL device context and the OpenCL queue;
- defines device buffers and copies `a` into `buffer a`, and `b` into `buffer b`;
- declares and compiles the kernels (kernels = device.compile(...));
- runs the kernel function sum (kernels.sum(...,[n],...)); and
- retrieves the array `c` from the device.

Notice how the kernel body consists of C99 code, but it's decorated using special macros: "`__kernel`," which indicates that the function is indeed a kernel; and "`__global`," which indicates that the arrays to be passed as arguments to the function are in the

```
import numpy
import pyopencl as pcl

class Device(object):
    flags = pcl.mem_flags
    def __init__ (self):
        self.ctx = pcl.create_some_context()
        self.queue = pcl.CommandQueue(self.ctx)
    def buffer(self,source=None,size=0,mode=pcl.mem_flags.READ_WRITE):
        if source is not None: mode = mode|pcl.mem_flags.COPY_HOST_PTR
        buffer = pcl.Buffer(self.ctx,mode, size=size, hostbuf=source)
        return buffer
    def retrieve(self,buffer,shape=None,dtype=numpy.float32):
        output = numpy.zeros(shape or buffer.size/4,dtype=dtype)
        pcl.enqueue_copy(self.queue, output, buffer)
        return output
    def compile(self,kernel):
        return pcl.Program(self.ctx,kernel).build()
```

Figure 2. Part of the ocl library, the Device class creates an abstration layer on top of pyOpenCL.

device's global memory. The function `get_global_id(0)` returns the ID of the running kernel. The number of running kernels is specified by the `[n]` argument of the call to `kernels.sum`. The last three arguments of this call are passed to the kernel function `sum`.

The `Device` class is defined in the ocl.py file and, in terms of the lower level pyOpenCL APIs, it looks like Figure 2.

Here, `self.ctx` is the device context and `self.queue` is the device queue. The functions `buffer`, `retrieve`, and `compile` map into the corresponding pyOpenCL functions `Buffer`, `enqueue copy`, and `Program`, but use a more compact syntax. For more details, see the official pyOpenCL documentation (http://documen.tician.de/pyopencl).

Laplace Solver

We can now implement a more complex algorithm. Specifically, given a 2D discretized scalar field q as input, we want to solve the following 2D Laplace equation in u:

$$(\partial_x^2 + \partial_y^2)u(x, y) = q(x, y). \qquad (1)$$

If we discretize the second derivatives

$$\partial_x^2 u(x, y) = (u(x-h, y) - 2u(x, y) + u(x+h, y)) / h^2$$

$$\partial_y^2 u(x, y) = (u(x, y-h) - 2u(x, y) + u(x, y+h)) / h^2$$

substitute in Equation 1, and solve in $u(x, y)$, we obtain:

$$u(x, y) = 1/4(u(x + h, y) + u(x - h, y) + u(x, y + h) + u(x, y - h) - h^2\, q(x, y)). \qquad (2)$$

We can thus solve Equation 1 by iterating Equation 2 until convergence. Here, q is the input, and the initial value of u is irrelevant because it will decorrelate during the convergence process.

We can now write a solver for this problem using `pyOpenCL+ocl` as shown in Figure 3.

This code differs from the previous code in many ways:

- First, `u`, `w`, and `q` are 2D arrays (`w` does not appear in the equations, but it will be used as temporary storage).
- Next, `q` is initialized by setting its value at +1 or −1 at some random sites.
- The kernel `solve` runs on a 2D grid, and therefore each running instance is identified by two numbers, x and y, determined by the calls to `get_global_id(0)` and `get_global_id(1)`, respectively.
- The kernel is compiled at runtime, therefore the value of n is passed to the kernel via string replacement. This is a trick, but it illustrates the possibility of dynamically generating OpenCL code at runtime.
- In the kernel, `XY` is a variable that stores the 1D index corresponding to the x, y coordinates.

```
from ocl import Device
from canvas import Canvas
from random import randint, choice
import numpy

n = 300
q = numpy.zeros((n,n), dtype=numpy.float32)
u = numpy.zeros((n,n), dtype=numpy.float32)
w = numpy.zeros((n,n), dtype=numpy.float32)

for k in range(n):
    q[randint(1, n-1),randint(1, n-1)] = choice((-1,+1))

device = Device()
q_buffer = device.buffer(source=q, mode=device.flags.READ_ONLY)
u_buffer = device.buffer(source=u)
w_buffer = device.buffer(source=w)


kernels = device.compile("""
       __kernel void solve(__global float *w,
                            __global const float *u,
                            __global const float *q) {
           int x = get_global_id(0);
           int y = get_global_id(1);
           int xy = y*WIDTH + x, up, down, left, right;
           if(y!=0 && y!=WIDTH-1 && x!=0 && x!=WIDTH-1) {
              up=xy+WIDTH; down=xy-WIDTH; left=xy-1; right=xy+1;
              w[xy] = 1.0/4.0*(u[up]+u[down]+u[left]+u[right] - q[xy]);
           }
       }
    """.replace('WIDTH',str(n)))

for k in range(1000):
    kernels.solve(device.queue, [n,n], None, w_buffer, u_buffer, q_buffer)
    (u_buffer, w_buffer) = (w_buffer, u_buffer)

u = device.retrieve(u_buffer,shape=(n,n))

Canvas().imshow(u).save(filename='plot.png')
```

Figure 3. Laplace solver with pyOpenCL using the Device class.

Notice that, from the device's viewpoint, all buffers are 1D C-style arrays.

- Finally, the `solve` kernel is called within the loop in `k`. The `[n,n]` argument specifies that we want to run one kernel for each x,y, where $x = 0 \ldots n - 1$ and $y = 0 \ldots n - 1$. Because the kernel uses `u` as input and as output, to avoid concurrency issues, we store the output in a temporary array `w` and swap `u` with `w` after each iteration. For didactic purposes, we don't check convergence; we simply iterate Equation 2 a fixed number of times.

We use the Canvas library (https://github.com/mdipierro/canvas), built on top of the matplotlib plotting library, to generate an 2D image of the output field u. Figure 4 shows the output of a random input. In this implementation, we assume a choice of units such that $h \equiv 1$.

Laplace Solver with ocl

Mixing Python syntax with C syntax can make programs difficult to read and error prone (a syntax error in the kernel might result in an obscure runtime error, for example). A possible solution to this problem consists of coding the kernels themselves in Python. There are two libraries for doing this: ocl and Clyther (https://github.com/srossross/Clyther/) both are based on the Meta decompiler.

Table 1. Converting ocl to C99/OpenCL.

Ocl	C99/OpenCL
`a = new_type(...)`	`type a = ...;`
`a = new_prt_type(...)`	`type *a = ...;`
`a = new_prt_prt_type(...)`	`type **a = ...;`
`None`	`Null`
`ADDR(x)`	`&x`
`REFD(x)`	`*x`
`CAST(prt_type,x)`	`(type*)x`

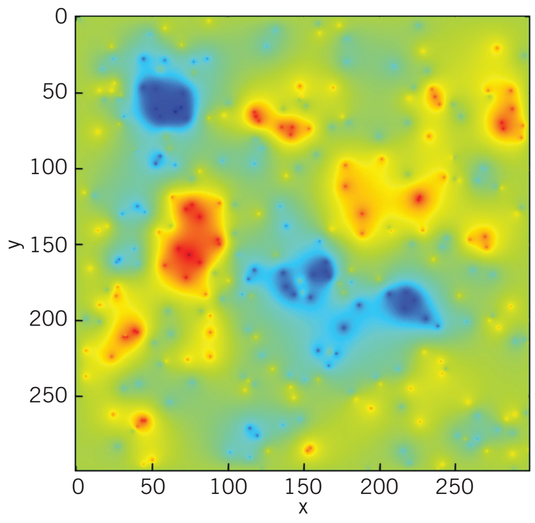

Figure 4. The output of the Laplace program. This output represents the 2D electrostatic potential for a random charge distribution.

Here, we use ocl, which I specifically designed to work with pyOpenCL.

We rewrite the Laplace solver as Figure 5 shows.

Most of the code is the same as in Figure 3, except for the `solve` kernel; here, the kernel is a Python function "decorated" with `@device.compiler.define_kernel(...)`. This decorator performs introspection of the function at runtime, converts the body of the function into an Abstract Syntax Tree (AST), serializes the AST into C99/OpenCL code, and compiles it using pyOpenCL. This lets us write OpenCL code using Python syntax.

One complication is that C99/OpenCL is a statically typed language, while Python is only strongly typed. We thus need a way to declare the type for variables in Python. For the function arguments, we can do this in the decorator arguments. For example:

```
w = "global:const:ptr_float"
```

is converted to

```
__global const float *w.
```

For local variables, we do this through the pseudo-casting operators, such as `new_int` and `new_float`. For example:

```
xy = new_int(x*n+y)
```

is converted into

```
int xy = x*n+y;.
```

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. The return type is determined automatically, but you must return a variable (or `None`), not an expression. Table 1 shows how other types are converted.

The goal of ocl isn't to translate arbitrary Python code into C99/OpenCL, but rather to allow the use of Python syntax to write OpenCL. This means that only C99/OpenCL variable types are allowed; Python types, such as lists and dictionaries, aren't. Moreover, all functions called in the Python code that constitutes the body of the `solve` function are intended to be OpenCL functions, not Python functions.

With ocl, you can define multiple kernels within the same program and call them when needed. You can also 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 won't be callable from the host program.

We can pass constants from the Python context to the OpenCL context as follows: `device.compile(..., constants = dict(n = n))`. Additional C files can be included using the syntax, `device.compile(..., includes = ['#include <math.h>'])`, where `includes` is a list of `#include` statements.

The ocl library also contains a decorator to convert a Python function into C99 and compile it at runtime without using pyOpenCL, as well as a decorator to convert a Python function into a JavaScript function (which is beyond this article's scope).

```
from ocl import Device
from canvas import Canvas
from random import randint, choice
import numpy

n = 300
q = numpy.zeros((n,n), dtype=numpy.float32)
u = numpy.zeros((n,n), dtype=numpy.float32)
w = numpy.zeros((n,n), dtype=numpy.float32)

for k in range(n):
    q[randint(1, n-1),randint(1, n-1)] = choice((-1,+1))

device = Device()
q_buffer = device.buffer(source=q, mode=device.flags.READ_ONLY)
u_buffer = device.buffer(source=u)
w_buffer = device.buffer(source=w)

@device.compiler.define_kernel(
    w='global:ptr_float',
    u='global:const:ptr_float',
    q='global:const:ptr_float')
def solve(w,u,q):
    x = new_int(get_global_id(0))
    y = new_int(get_global_id(1))
    xy = new_int(x*n+y)
    if y!=0 and y!=n-1 and x!=0 and x!=n-1:
        up = new_int(xy-n)
        down = new_int(xy+n)
        left = new_int(xy-1)
        right = new_int(xy+1)
        w[xy] = 1.0/4*(u[up]+u[down]+u[left]+u[right] - q[xy])

kernels = device.compile(constants=dict(n=n))

for k in range(1000):
    kernels.solve(device.queue, [n,n], None, w_buffer, u_buffer, q_buffer)
    (u_buffer, w_buffer) = (w_buffer, u_buffer)

u = device.retrieve(u_buffer,shape=(n,n))

Canvas().imshow(u).save(filename='plot.png')
```

Figure 5. Laplace solver with ocl.

Often, the addition of a new layer adds complexity and makes debugging more difficult. That isn't the case here. Although I use Python syntax to generate OpenCL code, the generated code is statement-by-statement equivalent to Python's source code, including variable names. The conversion layer helps debugging, because it prevents typical typographical errors, such as unbalanced brackets, missing semicolons, and the use of undefined symbols. If the Python code is syntactically valid (and that's easier to check than the OpenCL code), then the generated OpenCL is also syntactically valid. Logic errors are still possible, as are misspellings of external function names. The generated code is saved and it can be debugged as native OpenCL. Mapping the generated code back into the source Python code is trivial.

OpenCL is the only framework for writing portable applications that work on CPU, GPUs, and mobile devices. Yet programming into native OpenCL code embedded into C/C++ code is very complex and is a major barrier to entry for scientists. Further, while Python—thanks to the numPy libraries—has established itself as a new standard for scientific computing, it can't take advantage of modern hardware because the interpreter serializes all running code (including multithreaded code). Fortunately, pyOpenCL solves this by letting scientists combine the best of both worlds: using Python for the overall workflow, input/output, and plotting; and using OpenCL for the code's computing-intensive parts. Here, I took this a step further with ocl, which converts Python code into OpenCL code at runtime. ■

Acknowledgments

I thank Andreas Klöckner for pyOpenCL and the excellent documentation, and Alan Etkin and Jonathan Gemmell for proofreading this article.

References

1. M. Di Pierro and D. Skinner, "Concurrency in Modern Programming Languages," *Computing in Science & Engineering*, vol. 14, no. 6, 2012, pp. 8–12.
2. Nvidia Developer Zone, *CUDA C Programming Guide*, Nvidia; http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
3. J.A. Stratton et al., "MCUDA: An Efficient Implementation of CUDA Kernels for Multi-Core CPUs," *Proc. Int'l Workshop Languages and Compilers for Parallel Computing,* LNCS 5335, 2008, pp. 16–30.
4. J.E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science & Engineering*, vol. 12, no. 3, 2010, pp. 66–73.
5. A. Klöckner et al., "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation," *Parallel Computing,* vol. 38, no. 3, 2012, pp. 157–174.
6. S. Behnel et al., "Cython: The Best of Both Worlds," *Computing in Science & Eng.*, vol. 13, no. 2, 2011, pp. 31–39.

Massimo Di Pierro is an associate professor at DePaul University's School of Computing, where he manages the master's program in computational finance and teaches courses in scientific computing, Monte Carlo simulation, parallel programming, and Web development. Di Pierro has a PhD in high-energy theoretical physics from the University of Southampton, UK. Contact him at mdipierro@cs.depaul.edu.