# Introduction to PyOpenCL

Alexei Strelchenko,

LinkSCEEM HPC winter school

Nicosia, 8 February, 2012

The Cyprus Institute

CaSToRC

# Plan:

- **Py(thon)OpenCL overview**
  - What is PyOpenCL?
  - Python example
  - Why are we interesting in PyOpenCL?
- **A brief introduction to programming with PyOpenCL**
  - General structure: the first look at PyOpenCL code
  - Platforms, Devices, Contexts
  - Command queues and events
  - Memory objects
  - Programs and Kernels
- **A walk-through example: pi.py**
- **How to start?**

# Py(thon)OpenCL overview

- **developed by** *Andreas Kloeckner* (see link in the end)
- **all what is needed for productive coding with OpenCL:**
  - supports nearly all OS's (Linux, OS X, Windows)
  - supports OpenCL 1.2 (latest release)
  - is *complete*
  - allows interactive mode
  - automatically manages resources
  - automatically checks for (and reports) errors
  - integrates with **NumPy** package

# Py(thon)OpenCL overview

- **What is Python?**
  - general purpose  HL language
  - multi-paradigm : Imperative, Object-oriented, Functional etc.
  - dynamically typed, automatic memory management

- **NumPy module:**
  - extension to Python for scientific computing
  - adds support for (large) multi-dimensional arrays
  - provides library of math functions
  - (almost) as fast as C

# Python example:

```
import numpy as np

def compute_summ (x , y):
    return x + y

A = np.array([1, 2, 3, 4, 6], dtype = np.float32)
B = np.pi / A

C = compute_summ(A, B)

for element in C:
    print   element
```

# Why are we interesting in PyOpenCL?

- **What is nice with OpenCL scripting:**
  - (in general) is highly readable code!
  - fits well to OpenCL paradigm (*host code* + *kernels*)
  - we do not care how slow the host code is, because ...
  - ... performance gain comes mostly from *kernels*

- **Python + OpenCL = PyOpenCL:**
  - wrap every OpenCL API construct into a Python class
  - kernels ('the device code') are given as a string
  - PyOpenCL objects are garbage collected
  - errors are translated into Python exceptions

# A brief introduction to programming with PyOpenCL

## The first look at PyOpenCL code :

```
import pyopencl as cl, numpy as np
source = """
kernel void demo(global int *res){
        res[get_global_id(0)] = get_local_id(0);
}"""
#Initialization phase:
ctx     = cl.create_some_context()
queue = cl.CommandQueue(ctx)
#Create mem object:
dout    = cl.Buffer(ctx, cl.mem_flags.WRITE_ONLY,
                                    size = (1024 * np.dtype('int32').itemsize))
```

# The first look (cont.):

```
#Compilation phase:
prg      = cl.Program(ctx, simple_kernel).build()
#Execution phase:
(event =)prg.demo(queue, (1024, ), None, dout)

hout = numpy.empty((1024,), dtype = numpy.uint32)

(event =)cl.enqueue_read_buffer(queue, dout, hout).wait()
#Now do something with data on the host:
for element in hout:
    print element
```

# Platforms

Available plaforms: 'Apple', 'AMD', 'Intel' , 'NVIDIA CUDA'

List of **Platfrom** instances:

*class* **pyopencl.get_platforms**()

Select  preferred platform (example):

```
for found_platform in pyopencl.get_platforms():
    if found_platform.name == 'NVIDIA CUDA':
        my_platform = found_platform
```

# Devices

OpenCL devices: ALL, CPU, GPU, DEFAULT, ACCELERATOR etc ..

List of devices within platform matching ***device_type***:

*class* **pyopencl.get_devices**(device_type = device_type.ALL)

Select  preferred device (example):

```
for devices in my_platform.get_devices():
    if pyopencl.device_type.to_string(devices.name) == 'GPU':
        device = found_device
```

# Contexts, Command Queues, Events

*context* = **pyopencl.Context**(*devices* = None | [dev1, dev2], *dev_type* = None )

*context* = **pyopencl.create_some_context**( *Interactive* = True )

- create from device type, or list of devices
- *device_type* : ALL, GPU, CPU etc.
- to create context 'somehow' use **pyopencl.create_some_context**

*queue* = **pyopencl.CommandQueue**(*ctx*, *dev* = None, *properties* = [(prop, value), ...])

- attached (at least) to one device
- commands in a command queue can be executed synch. or asynchronously
- for asynch. use OUT_OF_ORDER_EXEC_MODE

# Contexts, Command Queues, Events (cont.)

All work is done through command queue(s) synchronously or async.:

- Computations
- Memory operations

Command format (always returns new event):

*new_event* = **pyopencl.enqueue_{*command*}** (*queue*, ...,
*wait_for* = [*event1*, *event2*, ....])

- command in a queue implicitly wait for completion of previous command(s)

Synchronization:

*new_event*.**wait**()

# Memory objects: Buffers

*buf* = **pyopencl.Buffer**(*context*, *flags* , *size* = 0, *hostbuf* = None)

- *flags* : {READ, WRITE}_ONLY, READ_WRITE, etc.
- *hostbuf* needs Python buffer interface (e.g **numpy.ndarray**)
- sparse resources can be explicitly freed: *buf*.**release**()

*event* = **pyopencl.enqueue_{*read/write*}_buffer**(*queue*, *buf*, *hostbuf*, *dev_offset* = 0, *wait_for* = None, *is_blocking* = True)

- for blocking/non-blocking memory transfer : True / False
- mapping memory in Host address space: **pyopencl.MemoryMap**
- WARNING: *is_blocking* defaults to True only in 2011.X versions!

# Memory objects: Images

*image* = **pyopencl.Image**(*context*, *flags* , *format*, *shape*, *pitches*, *hostbuf*)

- *shape* : 2D or 3D tuple
- sparse resources can be explicitly freed: *image*.**release**()

*event* = **pyopencl.enqueue_{*read/write*}_image**(*queue*, *buf*, *hostbuf*, ..... , *wait_for* = None, *is_blocking* = True)

- for blocking/non-blocking memory transfer : True / False

*event* = **pyopencl.enqueue_copy_image_to_buffer**(*queue*, *src*, *dst*, ...)
*event* = **pyopencl.enqueue_copy_buffer_to_image**(*queue*, *src*, *dst*, ...)

# Programs and Kernels

Program object:

*program* = **pyopencl.Program**(*context*, *source*)

- *source* : OpenCL device code

To build the program use:

*program*.**build**(*options* = “”, *devices* = None)

- *options* : cl-mad-enable etc.

Kernel object:

*kernel* = **pyopencl.Kernel**(*program*, '*kernel_name*')

# Execution:

Set kernel's arguments:

*kernel*.**set_args**(*args*)

*args:*

- buffers, images
- use None for null-pointers
- use numpy.dtype constructor for scalars (e.g. numpy.float64, numpy.uint32 etc.)
- use pyopencl.LocalMemory(*size*) *class* to pass __local memory arguments

Kernel execution:

*(event =)***pyopencl.enqueue_nd_range_kernel**(*queue*, *kernel*,
(Gx, ..), (Lx, ...), *offset* = 0, *wait_for* = None)

# A walk-through example : computation of pi

```
import pyopencl as cl,  numpy as np

#Available platform names:
AMD      = 'ATI Stream'
NVIDIA  = 'NVIDIA CUDA'

#Execution domain per device:
LOCAL DOMAIN_SIZE    = {'CPU' : 2;   'GPU' : 256}
GLOBAL_DOMAIN_SIZE = {'CPU': 8;    'GPU':  65536}

#Device source (not shown here!):
PI_KERNEL = """
__kernel pi_ocl(__global double *out, local double *lmem, double par1, int par2)
"""
```

# A walk-through example (cont.)

**Notes on source code:**

– *For double precision each vendor provides **different** extensions:*

```
#pragma OPENCL EXTENSION   cl_amd_fp64 : enable  (AMD)
#pragma OPENCL EXTENSION   cl_khr_fp64 : enable    (NVIDIA)
```

– *AMD Platform **allows** for* printf *in device kernels:*

```
#pragma OPENCL EXTENSION   cl_amd_printf : enable
```

– *WARNING:* __CPU__ *macro **defined** only for AMD platform!*

# A walk-through example (cont.)

```
#Initialization phase (select preferred platform and device type):

for found_platform in cl.get_platforms():
    if found_platform.name == NVIDIA:
        my_platform = found_platform
        print "Selected platform:", my_platform.name
        break

for device in my_platform.get_devices():
    dev_type = cl.device_type.to_string(device.type)
    if dev_type == 'GPU':
        dev = device
        print "Selected device: ", dev_type
```

# A walk-through example (cont.)

```
#create context on found list of devices:
context  = cl.Context([dev])

#create command queue:
queue   = cl.CommandQueue(context,

    properties=cl.command_queue_properties.PROFILING_ENABLE)

#Create and build program object for given device source:
program = cl.Program(context, PI_KERNEL).build()

#NOTE: build() may accept options (e.g., cl-mad-enable etc.)
```

# A walk-through example (cont.)

***#Memory initialization phase:***

```
#Create OpenCL buffers:
dout = cl.Buffer(context, cl.mem_flags.WRITE_ONLY,
                                          256 *
    np.dtype('float64').itemsize)

#Create __local memory argument for the kernel (dynamical
    allocation!):
lmem_arg  = cl.LocalMemory(256  * np.dtype('float64').itemsize)

#Create kernel constants:
num_steps  = pow(2, 30)
step       = 1.0 / num_steps
```

# A walk-through example (cont.)

```
#Create Kernel object (using kernel's entry point):
pi_kernel = cl.Program(program, 'pi_ocl')

#Execution phase:
event = pi_kernel(queue, (65536, ), (256, ), dout, lmem_arg,
                                        np.float64(step), np.int32(num_steps))

#Compute execution time (using event profiling):

event.wait()
elapsed = 1e-9 * (event.profile.end – event.profile.start)

print "Execution time: %g s" % elapsed
```

# A walk-through example (cont.)

## Notes on Execution phase:

– WARNING: Python **float** and **int** objects will NOT work:
scalar arguments must be an object acceptable to the **np.dtype** constructor : **np.int32() , np.float32()** etc.

– for dynamically allocated local memory use
**cl.LocalMemory**(size)

– global/local work size pass as tuples, (e.g. , (256, 64, 1) etc.)

# A walk-through example (cont.)

```
#Create host array (as numpy ndarray):
hout = np.zeros((256, ), dtype = np.float64)

#Now copy data:
event = cl.enqueue_read_buffer(queue, dout, hout)

#Wait until the data copy will be complete:
event.wait()

#Perform final reduction using sum function from numpy lib:
my_pi = (1.0 / num_steps) * np.sum(hout)

print "Error: ", abs(np.pi - my_pi)
```

# How to start?

- **What you need to begin with PyOpenCL:**
  - OpenCL toolkit (AMD APP SDK, NVIDIA CUDA)
  - Python (ver 2.4 and later)
  - NumPy package
  - OpenCL device available (for NVIDIA: NVIDIA GPU)
  - PyOpenCL package installed (see next slide)
- **Additional notes:**
  - cl_amd_printf extension may be **very** helpful
  - AMD OpenCL platform supports x86 architecture

# Links:

- **http://mathema.tician.de/software/pyopencl**
  - A. Kloekcner's web-page: PyOpenCL soft, documentations
- **www.khronos.org**
  - OpenCL documentations
- **http://developer.amd.com/gpu/AMDAPPSDK**
  - AMD APP SDK,  incl. OpenCL for x86 architecture
- **http://developer.nvidia.com/object/opencl.html**
  - NVIDIA OpenCL, documentations, forum

# Thank you

www.cyi.ac.cy

The Cyprus Institute

CaSToRC