High Performance computing with Python

ESRF | The European Synchrotron

Jérôme Kieffer
Online Data Analysis

Layout

0. No optimization without profiling
1. Cython

    binary Python extensions in Python

2. Programming massively parallel devices

    Example of GPU

3. Example with PyOpenCL

About exercises

- All exercises are about defining a mask:

  http://paulbourke.net/geometry/polygonmesh/

  - Consider an image of 1024x1024

```
L=1024; N=24
msk = numpy.zeros((L,L), dtype='uint8')
```

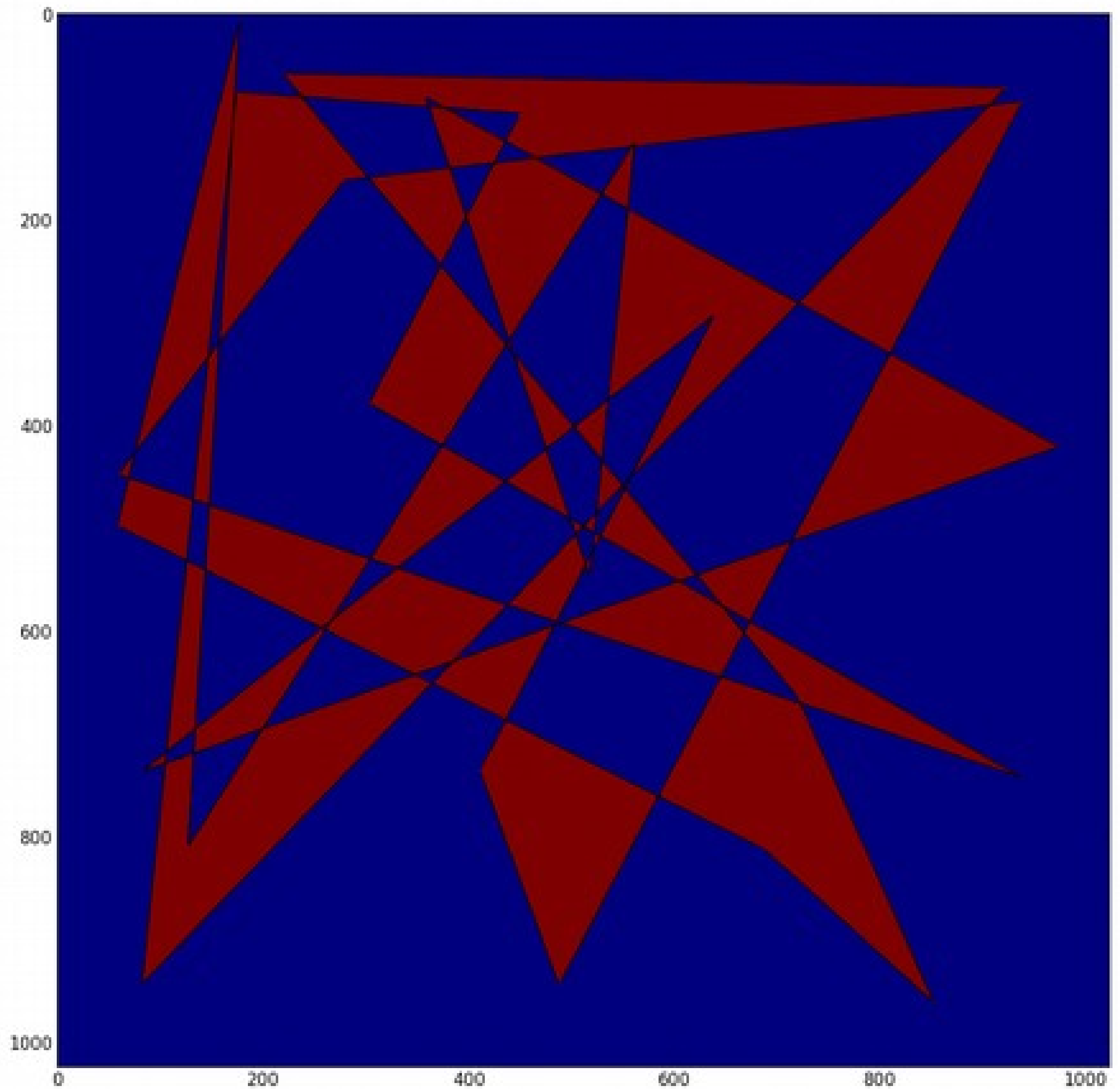
  - And a 24-edge polygon, randomly chosen:

```
vertices = [(random.randint(0, L), random.randint(0, L))
    for i in range(N)]
```

  - Define all pixel which are inside the polygon:

```
imshow(msk); last = vertices[-1]
for v in vertices:
    annotate("",xy=v,xytext=last,xycoords="data",
            arrowprops=dict(arrowstyle="-"))
    last=v
```

Result



Only optimize tested & profiled code !

- Use the profiler from python: `cProfile`

```
python -m cprofile -o log run_tuple.py
```

- Visualization :
  - Either using the *`pstats`* module
  - Or better : visualize with *`runsnake`*

```
pip install --user runsnakerun
sudo apt-get install runsnakerun
```

- Or simply use *`%timeit`* (within ipython/jupyter)

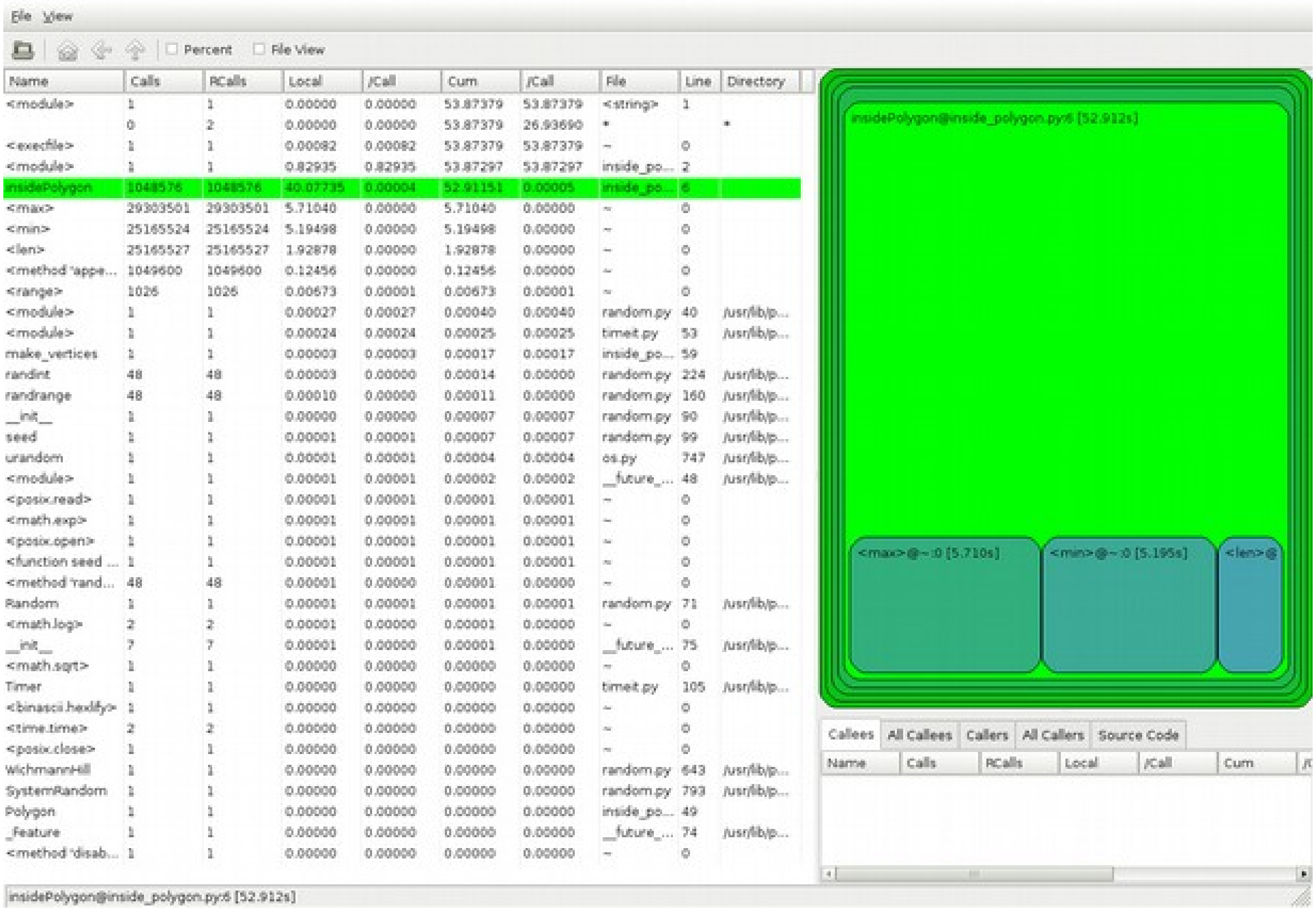
Exercise: Profile the code

- Check if a point is inside a polygon :
  - Code is in 0_Python:

    https://raw.github.com/kif/HPP/master/0_Python/inside_polygon.py
  - Profile the code considering :
    - 24-edges polygon
    - 1024x1024 pixels in image to check

Execution time on a bi-Xeon5520 (2.27GHz):
Based on tuples: 36s
Based on numpy: 450s

Visualization with runsnake



Name	Calls	RCalls	Local	/Call	Cum	/Call	File	Line	Directory
<module>	1	1	0.00000	0.00000	53.87379	53.87379	<string>	1	
	0	2	0.00000	0.00000	53.87379	26.93690	*		*
<execfile>	1	1	0.00082	0.00082	53.87379	53.87379	~	0	
<module>	1	1	0.82935	0.82935	53.87297	53.87297	inside_po...	2	
insidePolygon	1048576	1048576	40.07735	0.00004	52.91151	0.00005	inside_po...	6	
<max>	29303501	29303501	5.71040	0.00000	5.71040	0.00000	~	0	
<min>	25165524	25165524	5.19498	0.00000	5.19498	0.00000	~	0	
<len>	25165527	25165527	1.92878	0.00000	1.92878	0.00000	~	0	
<method 'appe...	1049600	1049600	0.12456	0.00000	0.12456	0.00000	~	0	
<range>	1026	1026	0.00673	0.00001	0.00673	0.00001	~	0	
<module>	1	1	0.00027	0.00027	0.00040	0.00040	random.py	40	/usr/lib/p...
<module>	1	1	0.00024	0.00024	0.00025	0.00025	timeit.py	53	/usr/lib/p...
make_vertices	1	1	0.00003	0.00003	0.00017	0.00017	inside_po...	59	
randint	48	48	0.00003	0.00000	0.00014	0.00000	random.py	224	/usr/lib/p...
randrange	48	48	0.00010	0.00000	0.00011	0.00000	random.py	160	/usr/lib/p...
__init__	1	1	0.00000	0.00000	0.00007	0.00007	random.py	90	/usr/lib/p...
seed	1	1	0.00001	0.00001	0.00007	0.00007	random.py	99	/usr/lib/p...
urandom	1	1	0.00001	0.00001	0.00004	0.00004	os.py	747	/usr/lib/p...
<module>	1	1	0.00001	0.00001	0.00002	0.00002	__future_...	48	/usr/lib/p...
<posix.read>	1	1	0.00001	0.00001	0.00001	0.00001	~	0	
<math.exp>	1	1	0.00001	0.00001	0.00001	0.00001	~	0	
<posix.open>	1	1	0.00001	0.00001	0.00001	0.00001	~	0	
<function seed ...	1	1	0.00001	0.00001	0.00001	0.00001	~	0	
<method 'rand...	48	48	0.00001	0.00000	0.00001	0.00000	~	0	
Random	1	1	0.00001	0.00001	0.00001	0.00001	random.py	71	/usr/lib/p...
<math.log>	2	2	0.00001	0.00000	0.00001	0.00000	~	0	
__init__	7	7	0.00001	0.00000	0.00001	0.00000	__future_...	75	/usr/lib/p...
<math.sqrt>	1	1	0.00000	0.00000	0.00000	0.00000	~	0	
Timer	1	1	0.00000	0.00000	0.00000	0.00000	timeit.py	105	/usr/lib/p...
<binascii.hexlify>	1	1	0.00000	0.00000	0.00000	0.00000	~	0	
<time.time>	2	2	0.00000	0.00000	0.00000	0.00000	~	0	
<posix.close>	1	1	0.00000	0.00000	0.00000	0.00000	~	0	
WichmannHill	1	1	0.00000	0.00000	0.00000	0.00000	random.py	643	/usr/lib/p...
SystemRandom	1	1	0.00000	0.00000	0.00000	0.00000	random.py	793	/usr/lib/p...
Polygon	1	1	0.00000	0.00000	0.00000	0.00000	inside_po...	49	
_Feature	1	1	0.00000	0.00000	0.00000	0.00000	__future_...	74	/usr/lib/p...
<method 'disab...	1	1	0.00000	0.00000	0.00000	0.00000	~	0	

Nota: running code in profiling mode slows it down ...

1. Cython

- Python binary extension written in Python
  - Translates python code to C/C++
  - Provides speed but needs compilation
  - Built-in support of numpy arrays
  - Can interface with C/C++ libraries (wrapping)
- Cython is getting mainstream for many projects:
  - Lxml, pyzmq, h5py, scipy, scikit image/learn, mpi4py …

Extension to Python

- Cython modules have `.pyx` extensions
  - Just copy your python file with `.pyx` extension
- Convert your code to C using
  `cython -a inside_polygon.pyx`
  - option `-a` produces a webpage with annotation: open it with a web browser.
  - It will be used later for further optimization
  - Note that a `.c` file is created (possibly erasing sources)

Building a Cython module

- Using a *setup.py* is recommended:

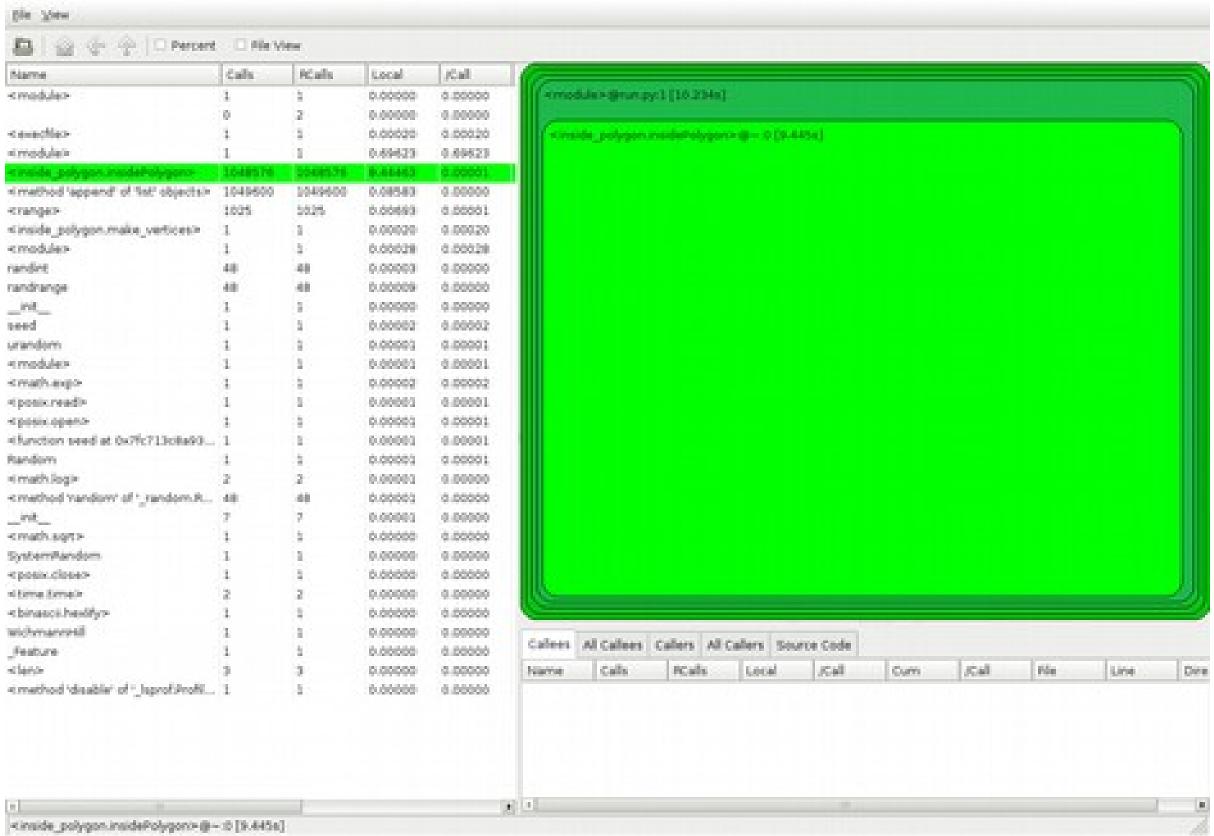
```
#!/usr/bin/python
from distutils.core import setup
from Cython.Distutils import build_ext
from distutils.extension import Extension

cy_mod=Extension("inside_polygon",
    sources=   ["inside_polygon.pyx"])

setup(ext_modules = [cy_mod],
    cmdclass = {'build_ext': build_ext})
```

- Build module using:

```
$ python setup.py build_ext -i
```

  – This builds the binary module in-place

Exercise: Compile a binary module



Speed-up: 4x just by compiling !
Problem: not much to be seen inside → use annotation file

Cython optimization

- Benchmark your code !!!
- Inspect the annotated page (option -a)
  - Remove the yellow in the critical part
- Use `cdef` to enforce the type of variables

```
cdef int i, n
```

  - Will prevent the type checking which slows down Python
  - Use Numpy containers as they enforce regular data-types (slows down much with pure Python !)

Annotated file

Generated by Cython 0.20dev on Tue Feb 4 10:30:22 2014

Raw output: [inside_polygon.c]()

```
#!/usr/bin/python


def insidePolygon(vertices, point, border_value=True):
    """
    Return True/False is a pixel is inside a polygon.

    @param vertices:
     @param point: 2-tuple of integers or list
     @param border_value: boolean
     """
     counter = 0
     for i,polypoint1 in enumerate(vertices):
         if (polypoint1[0] == point[0]) and (polypoint1[1] == point[1]):
             return border_value
         polypoint2 = vertices[(i+1)%len(vertices)]
         if (point[1] > min(polypoint1[1], polypoint2[1])):
             if (point[1] <= max(polypoint1[1], polypoint2[1])):
                 if (point[0] <= max(polypoint1[0], polypoint2[0])):
                     if (polypoint1[1] != polypoint2[1]):
                         xinters = (point[1]-polypoint1[1])*(polypoint2[0]-polypoint1[0])/(polypoint2[1]-polypoint1[1])+po
                         if (polypoint1[0] == polypoint2[0]) or (point[0] <= xinters):
                             counter+=1
     if counter % 2 == 0:
         return False
     else:
         return True
```

The lighter, the better & faster
Click on a line to see the generated code

Tell cython to be less pythonic

```
cimport cython
```

- Use C division instead of python's bullet-proof division:

```
@cython.cdivision(True)
```

- Do not wrap around for array[-1]:

```
@cython.wraparound(False)
```

- Do not check boundaries of arrays:

```
@cython.boundscheck(False)
```

Be aware you get exposed to segmentation-faults !!!

Wrapping C library

- C libraries can declared:

```
cdef extern from "InsidePolygonWithBounds.h":
   void PointsInsidePolygon(double *, int , \
                            double *, int , int ,\
     unsigned char *) nogil
```

- Cython offers a set of `cimport` libraries:

```
from libc.stdlib cimport malloc, free
from libc.string cimport memset, memcpy
```

- Include files can be merged into a `.pxd`

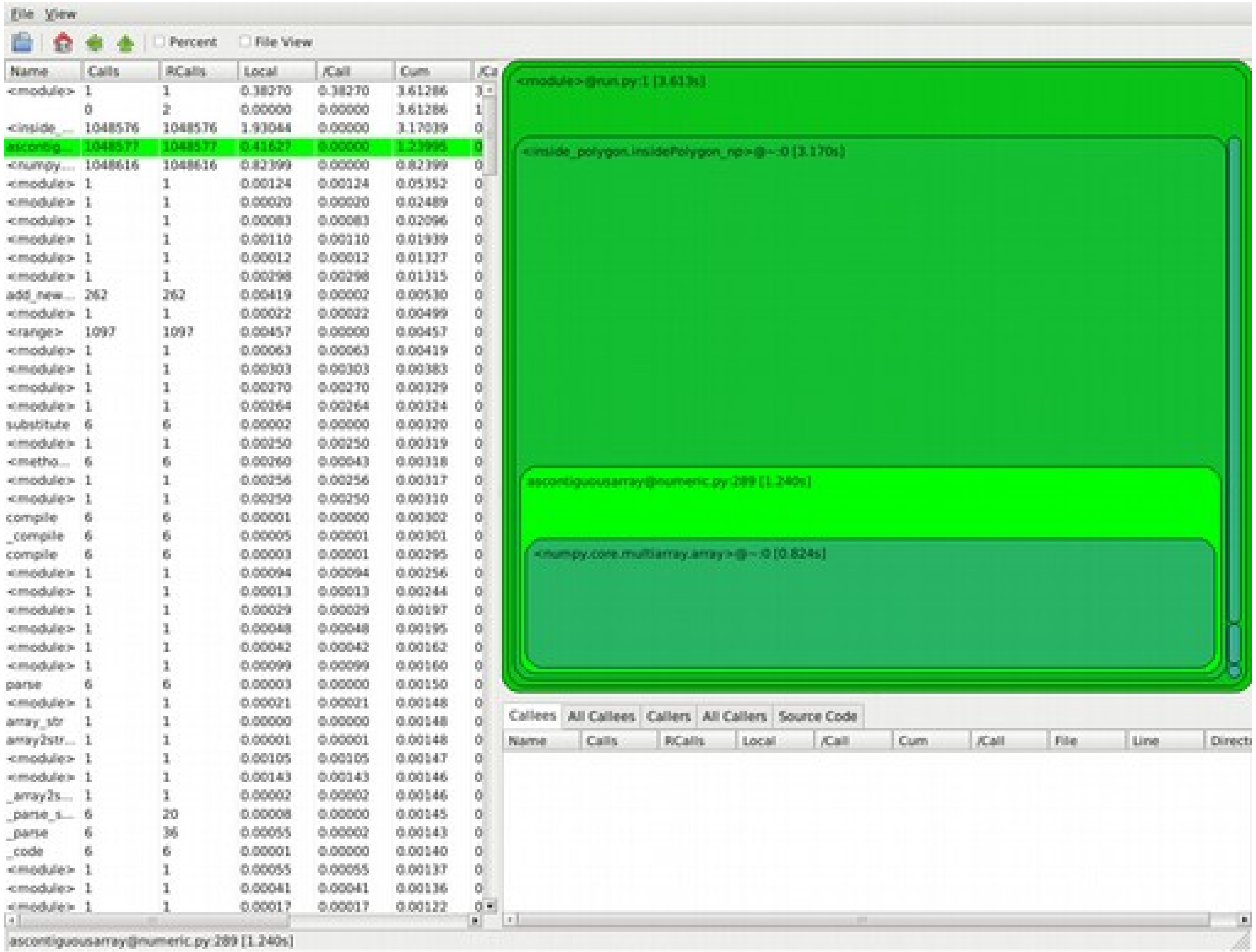
```
from InsidePolygon cimport PointsInsidePolygon
```

- Append your '.c' files to the setup.py to have files compiled:

```
cy_mod = Extension("inside_polygon",
    sources=["inside_polygon.pyx",
             'InsidePolygonWithBounds.c'],
    language="c")
```

Execution time: 0.270s

Analyse the numpy overhead ...



To get ride of this overhead, we need to keep the vertices
In the “c” space ... i.e. use Cython classes.

Cython classes

- Unlike Python classes, Cython classes:
  - Declare all attributes:
    - All instance variable live in C space:
      - instance variable are not accessible from Python
      - Accessors are needed to change them from Python
    - No monkey patching
  - Can have a C-constructor: __cinit__
    - Then need a specific destructor
- Cython classes can be translated into C or C++

Exercise:

- Define a cython-class Polygon containing:
  - A memory view on a set of vertices
  - The length of those vertices.
- Benchmark & profile the code ...

Execution time: 0.118s

Wrap C++ library

- Cython offers all stdc++ as pxd imports:
  - Deque, list, vector, pair, set, string, map, queue, stack, …
- Need to specify the language C++ in setup.py:

```
cy_mod = Extension("inside_polygon",
    sources=["inside_polygon.pyx"],
    language="c++")
```

- Cython needs the option --cplus to generate C++ code
- But Cython classes cannot inherit from C++ classes (yet)

Few words about the GIL

- This is the Global Interpreter Lock:
  - Core of the C-Python implementation
    - Other flavors of python are GIL-free (i.e. Jython)
  - Prevent multiple threads from simultaneously accessing a single Python-object
  - Only Python-free code can actually run in parallel
  - This is why C-function are annotated with “nogil”
  - Cython can chose to release the GIL (or not)

def, cdef and cpdef function

Multiple types of function exist in Cython:

- `def` functions are Python functions
- `cdef` functions are C-only functions
  - They need to declare their arguments
  - They need to declare their return types
  - They can be `nogil` hence run in //
- `cpdef` functions both C&Python functions

Use OpenMP to parallelize

- Cython provides tools to easily parallelize code using OpenMP:

```
from cython.parallel import prange
for i in prange(n, nogil=True):
    do_some_gil_free_stuff()
```

- One needs to link the code with OpenMP:

```
cy_mod = Extension("inside_polygon",
        sources=["inside_polygon.pyx"],
        extra_compile_args=['-fopenmp'],
        extra_link_args=['-fopenmp'],
        language="c")
```

Execution time: 0.020s

Summary of speed-ups

Method	Execution time (s)	Speed-up
Python with tuples	36	1
Python with numpy	450	0.08
Cython with tuples	9	4
Cython with numpy+Opt	3.6	10
Cython + C library	0.270	133
Cython-class	0.118	305
Cython with OpenMP	0.020	1800

Measured on a dual-quadcore @2.27 GHz Xeon 5520

Conclusion on Cython

- Cython should be used when the limits of NumPy are reached
- Cython provides a systematic way to make algorithms run faster
- Cython provides an elegant way of wrapping C/C++ libraries
- Cython, unlike SWIG, targets only Python

Conclusion on profiling

- Most numerical algorithms deserve NumPy vectorization
- In this case, consider the vectorization of the 2 outer most loops (loops over positions)
- Look at the implementation:

  inside_polygon.polygon_vec in 0_Python

  Execution time: 238ms !

Summary of speed-ups

Method	Execution time (s)	Speed-up
Python with tuples	36	1
Python with numpy	450	0.08
Cython with tuples	9	4
Cython with numpy+Opt	3.6	10
Cython + C library	0.270	133
Cython-class	0.118	305
Cython with OpenMP	0.020	1800
Vectorized Numpy	0.238	151

Measured on a dual-quadcore @2.27 GHz Xeon 5520

Premature optimization is the root of all evil

- In Donald Knuth's paper: "Structured Programming With GoTo Statements", he wrote:

  Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3%."

2. OpenCL

- OpenCL for an heterogeneous world
- Concepts in OpenCL
- Programming GPU using PyOpenCL
- Excercise & benchmarking
- Further training

Most slides are stolen from:
https://github.com/HandsOnOpenCL/Lecture-Slides/releases

It's a Heterogeneous world

A modern computing platform includes:

- One or more CPUs
- One of more GPUs
- DSP processors
- Accelerators
- … other?

E.g. Samsung® Exynos 5:

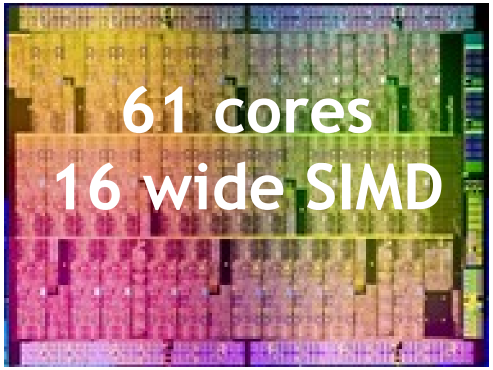
- Dual core ARM A15 1.7GHz, Mali T604 GPU

E.g. Intel i7 4950HQ with IRIS

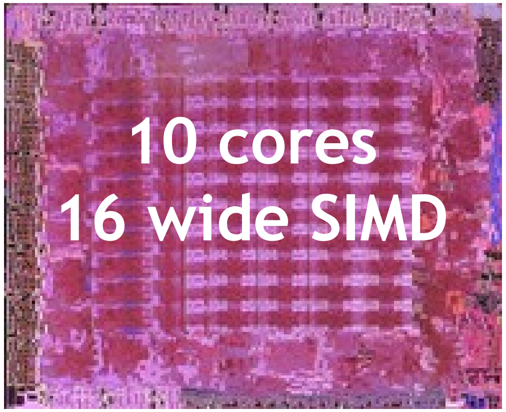
OpenCL lets Programmers write a single portable program that uses ALL resources in the heterogeneous platform

Microprocessor trends

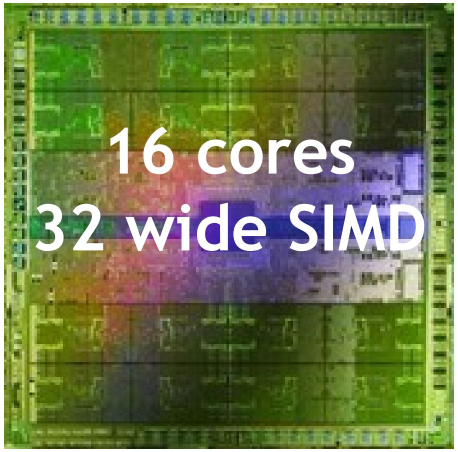
Individual processors have many (possibly heterogeneous) cores.



Intel® Xeon Phi™ coprocessor



ATI™ RV770

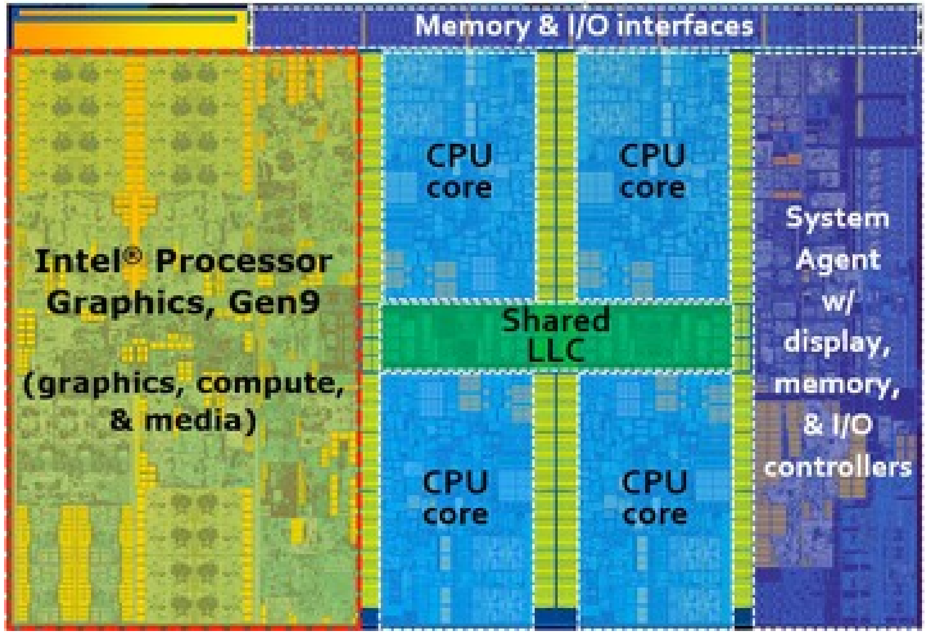


NVIDIA® Tesla® C2090

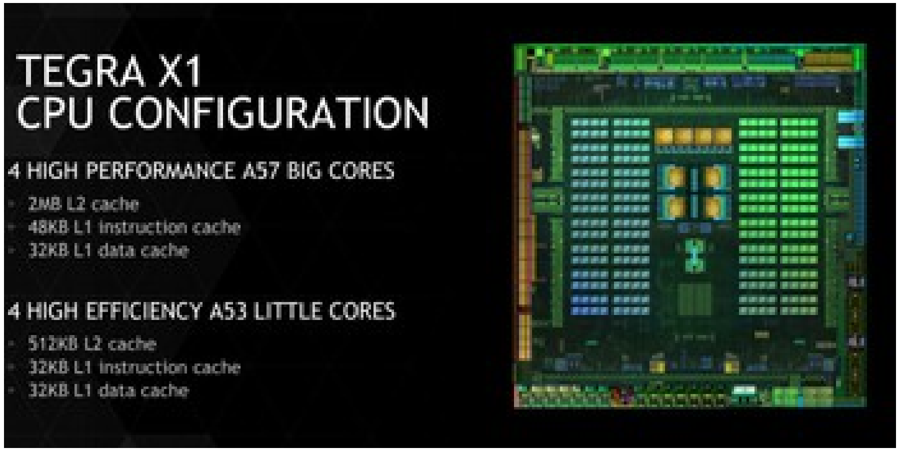
The Heterogeneous many-core challenge:

How are we to build a software ecosystem for the Heterogeneous many core platform?

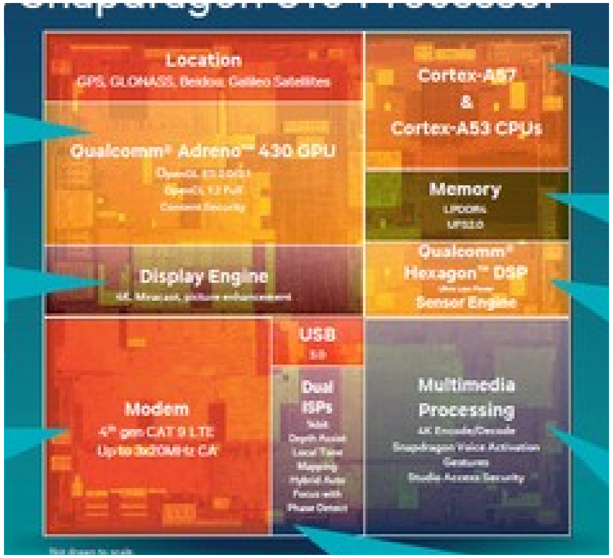
Latest trends … convergence



- Intel Skylake:
  Core i7 6xxx



- Nvidia Tegra X1
  Nvidia Shield TV
- Qualcom snapdragon



Industry Standards for Programming Heterogeneous Platforms

CPUs
Multiple cores driving performance increases

Emerging Intersection

GPUs
Increasingly general purpose data-parallel computing

OpenCL

Multi-processor programming – e.g. OpenMP

Heterogeneous Computing

Graphics APIs and Shading Languages

OpenCL – Open Computing Language

Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

The origins of OpenCL

AMD

ATI

Merged, needed commonality across products

NVIDIA

GPU vendor – wants to steal market share from CPU

Intel

CPU vendor – wants to steal market share from GPU

Apple

Was tired of recoding for many core, GPUs. Pushed vendors to standardize.

Wrote a rough draft straw man API

Khronos Compute group formed

ARM

Nokia

IBM

Sony

Qualcomm

Imagination

TI

+ many more

OpenCL

OpenCL Working Group within Khronos

- Diverse industry participation
  - Processor vendors, system OEMs, middleware vendors, application developers.
- OpenCL became an important standard upon release by virtue of the market coverage of the companies behind it.

OpenCL Timeline

- Launched Jun'08 … 6 months from "strawman" to OpenCL 1.0
- Rapid innovation to match pace of hardware innovation
  - 18 months from 1.0 to 1.1 and from 1.1 to 1.2
  - Goal: a new OpenCL every 18-24 months
  - Committed to backwards compatibility to protect software investments

Dec08
Khronos publicly releases **OpenCL 1.0** specification

During 2H09
Multiple conformant implementations ship across a diverse range of platforms.

Jun10
Khronos publicly releases **OpenCL 1.1** specification.
Conformant implementations available shortly thereafter

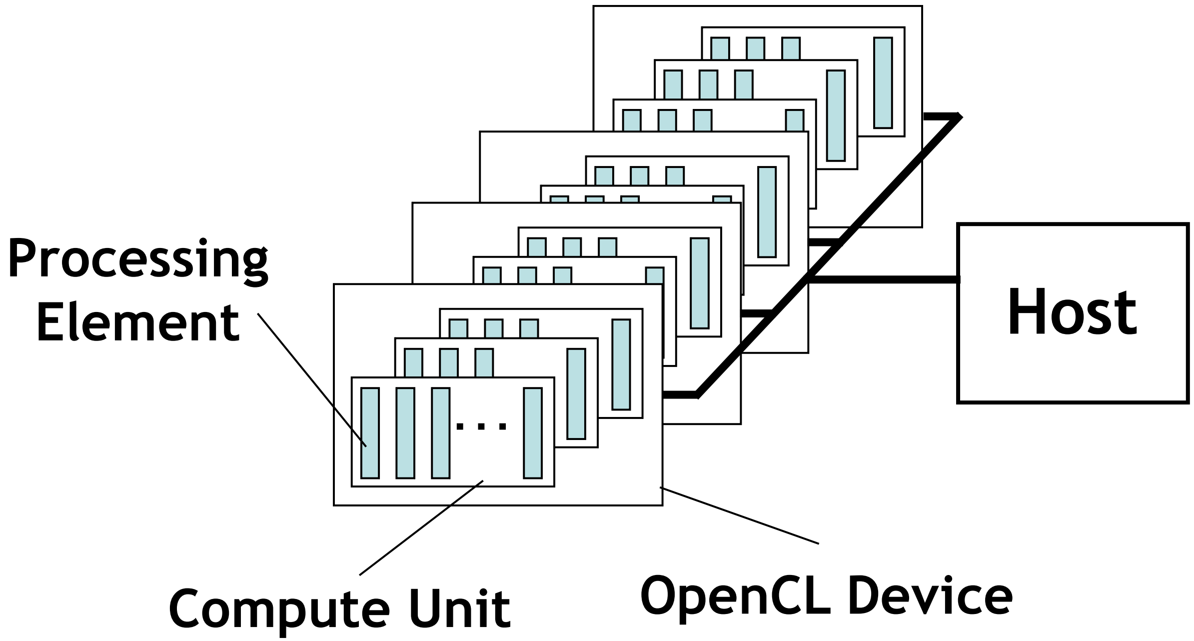
Nov11
Release of **OpenCL 1.2**

OpenCL: From cell phone to supercomputer

- OpenCL Embedded profile for mobile and embedded silicon
  - Relaxes some data type and precision requirements
  - Avoids the need for a separate “ES” specification
- Khronos APIs provide computing support for imaging & graphics
  - Enabling advanced applications in, e.g., Augmented Reality
- OpenCL will enable parallel computing in new markets
  - Mobile phones, cars, avionics

A camera phone with GPS processes images to recognize buildings and landmarks and provides relevant data from internet

OpenCL Platform Model



- One ***Host*** and one or more ***OpenCL Devices***
  - Each OpenCL Device is composed of one or more ***Compute Units***
  - Each Compute Unit is divided into one or more ***Processing Elements***
- Memory divided into ***host memory*** and ***device memory***

OpenCL Platform Example (One node, two CPU sockets, two GPUs)

CPUs:

- Treated as one OpenCL device
  - One CU per core
  - 1 PE per CU, or if PEs mapped to SIMD lanes, *n* PEs per CU, where *n* matches the SIMD width
- Remember:
  - the CPU will also have to be its own host!

GPUs:

- Each GPU is a separate OpenCL device
- Can use CPU and all GPU devices concurrently through OpenCL

CU = Compute Unit; PE = Processing Element

Inspect your system

- Use the *clinfo* program:

Number of platforms: 2

```
  Platform Profile:                    FULL_PROFILE
  Platform Version:                    OpenCL 1.2 LINUX
  Platform Name:                       Intel(R) OpenCL
  Platform Vendor:                     Intel(R) Corporation
  Platform Extensions:                 cl_khr_icd
cl_khr_global_int32_base_atomics
cl_khr_global_int32_extended_atomics cl_khr_local_int32_base_atomics
cl_khr_local_int32_extended_atomics cl_khr_byte_addressable_store
cl_khr_spir cl_intel_exec_by_local_thread cl_khr_fp64

  Platform Profile:                    FULL_PROFILE
  Platform Version:                    OpenCL 1.2 AMD-APP (1348.5)
  Platform Name:                       AMD Accelerated Parallel
Processing
  Platform Vendor:                     Advanced Micro Devices, Inc.
  Platform Extensions:                 cl_khr_icd cl_amd_event_callback
cl_amd_offline_devices
```

ESRF

Course materials

In addition to these slides, it is useful to have:

OpenCL 1.1 Reference Card

This card will help you keep track of the API as you do the exercises:

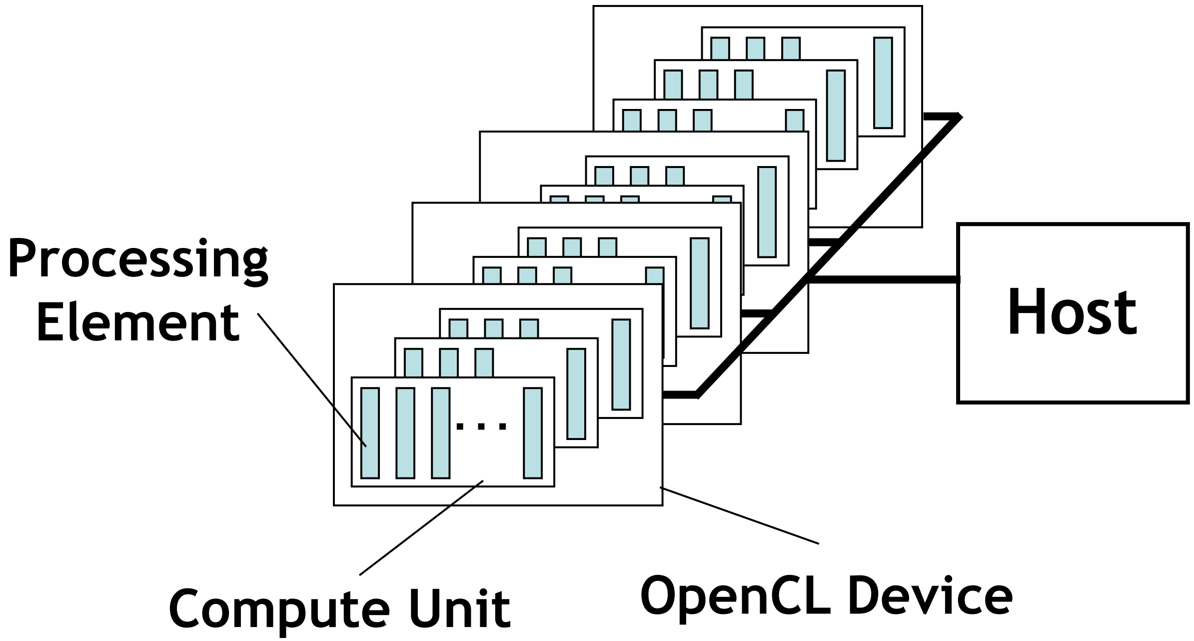
https://www.khronos.org/files/opencl-1-1-quick-reference-card.pdf

The v1.1 spec is also very readable and recommended to have on-hand:

https://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

Important OpenCL concepts

OpenCL Platform Model



- One ***Host*** and one or more ***OpenCL Devices***
  - Each OpenCL Device is composed of one or more ***Compute Units***
  - Each Compute Unit is divided into one or more ***Processing Elements***
- Memory divided into ***host memory*** and ***device memory***

The BIG idea behind OpenCL

- Replace loops with functions (a kernel) executing at each point in a problem domain
  - E.g., process a 1024x1024 image with one kernel invocation per pixel or 1024x1024=1,048,576 kernel executions

Traditional loops

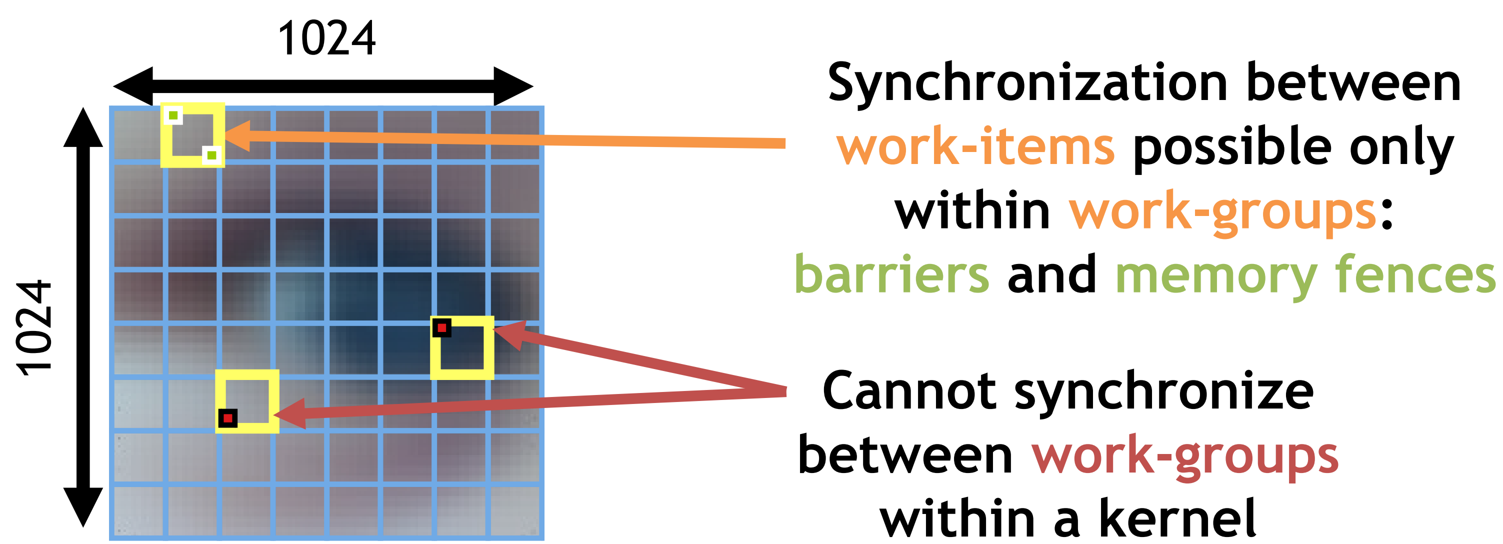
```
void
mul(const int n,
    const float *a,
    const float *b,
          float *c)
{
  int i;
  for (i = 0; i < n; i++)
    c[i] = a[i] * b[i];
}
```

Data Parallel OpenCL

```
__kernel void
mul(__global const float *a,
    __global const float *b,
    __global       float *c)
{
  int id = get_global_id(0);
  c[id] = a[id] * b[id];
}
// many instances of the kernel,
// called work-items, execute
// in parallel
```

An N-dimensional domain of work-items

- Global Dimensions:
  - 1024x1024 (whole problem space)
- Local Dimensions:
  - 128x128 (work-group, executes together)



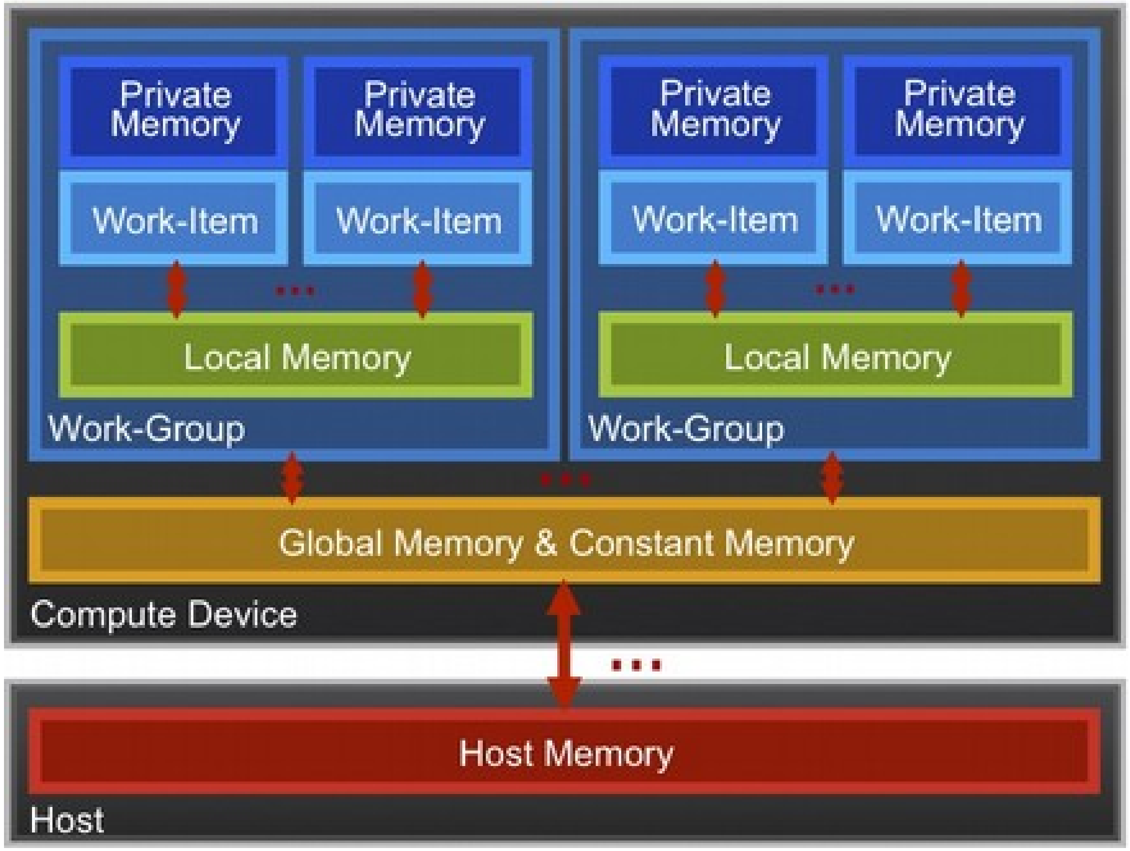
- Choose the dimensions that are “best” for your algorithm

OpenCL N Dimensional Range (NDRange)

- The problem we want to compute should have some **dimensionality**;
  - For example, compute a kernel on all points in a cube
- When we execute the kernel we specify **up to 3 dimensions**
- We also **specify the total problem size** in each dimension – this is called the **global** size
- We associate each point in the iteration space with a **work-item**

OpenCL Memory model

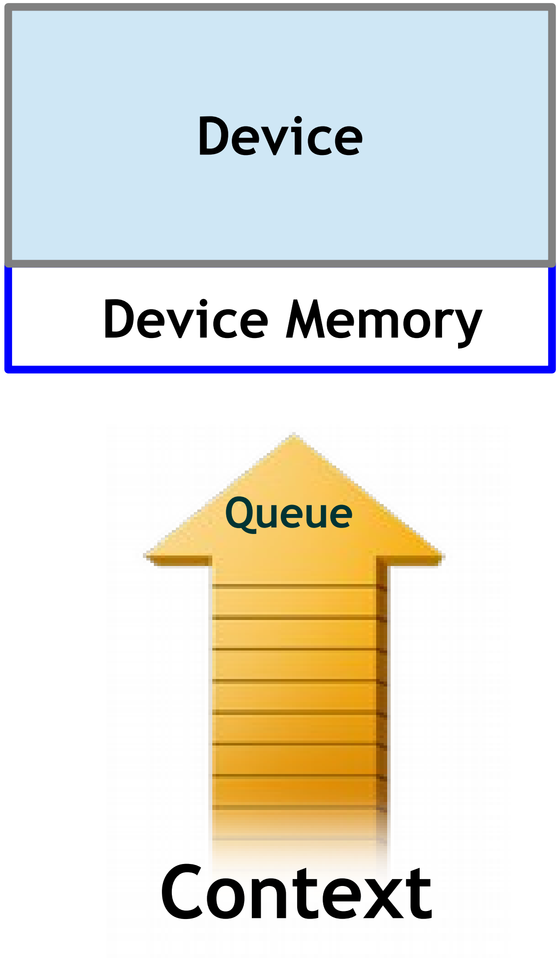
- *Private Memory*
  - Per work-item
- *Local Memory*
  - Shared within a work-group
- *Global Memory /Constant Memory*
  - Visible to all work-groups
- *Host memory*
  - On the CPU



Memory management is explicit:
You are responsible for moving data from host → global → local *and* back

Context and Command-Queues

- ***Context*:**
  – The environment within which kernels execute and in which synchronization and memory management is defined.
- The ***context*** includes:
  – One or more devices
  – Device memory
  – One or more command-queues
- All ***commands*** for a device (kernel execution, synchronization, and memory transfer operations) are submitted through a ***command-queue***.
- Each ***command-queue*** points to a single device within a context.



Execution model (kernels)

- OpenCL execution model … define a problem domain and execute an instance of a **kernel** for each point in the domain

```
__kernel void times_two(
     __global float* input,
     __global float* output)
{
    int i = get_global_id(0);
    output[i] = 2.0f * input[i];
}
```

get_global_id(0)

10

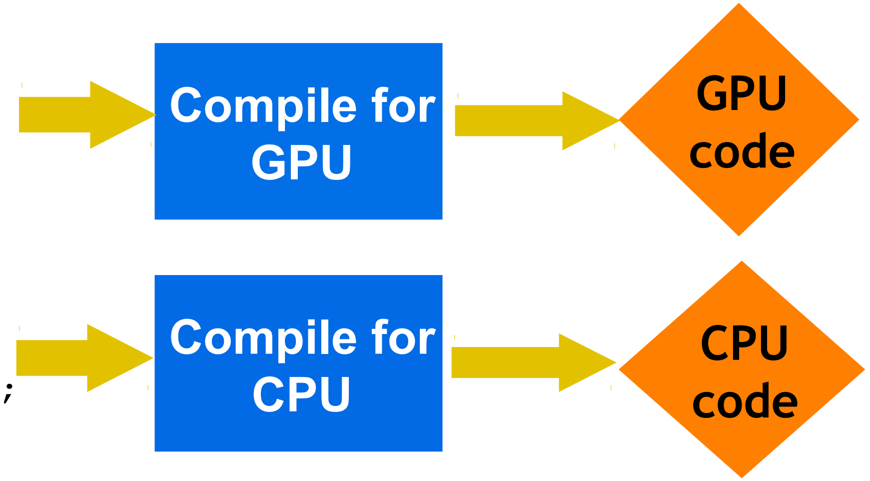
Input	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Output	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46	48	50

Building Program Objects

- The program object encapsulates:
  - A context
  - The program kernel source or binary
  - List of target devices and build options
- The build process to create a program object:
  - pyopencl.Program(context, sources). build()

OpenCL uses runtime compilation … because in general you don't know the details of the target device when you ship the program

```
__kernel void
horizontal_reflect(read_only image2d_t src,
                   write_only image2d_t dst)
{
  int x = get_global_id(0);  // x-coord
  int y = get_global_id(1);  // y-coord
  int width = get_image_width(src);
  float4 src_val = read_imagef(src, sampler,
                        (int2)(width-1-x, y));
  write_imagef(dst, (int2)(x, y), src_val);
}
```



Example: vector addition

- The “hello world” program of data parallel programming is a program to add two vectors

```
C[i] = A[i] + B[i] for i=0 to N-1
```

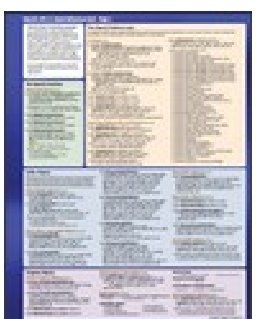
- For the OpenCL solution, there are two parts
  - Kernel code
  - Host code

Vector Addition - Kernel

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

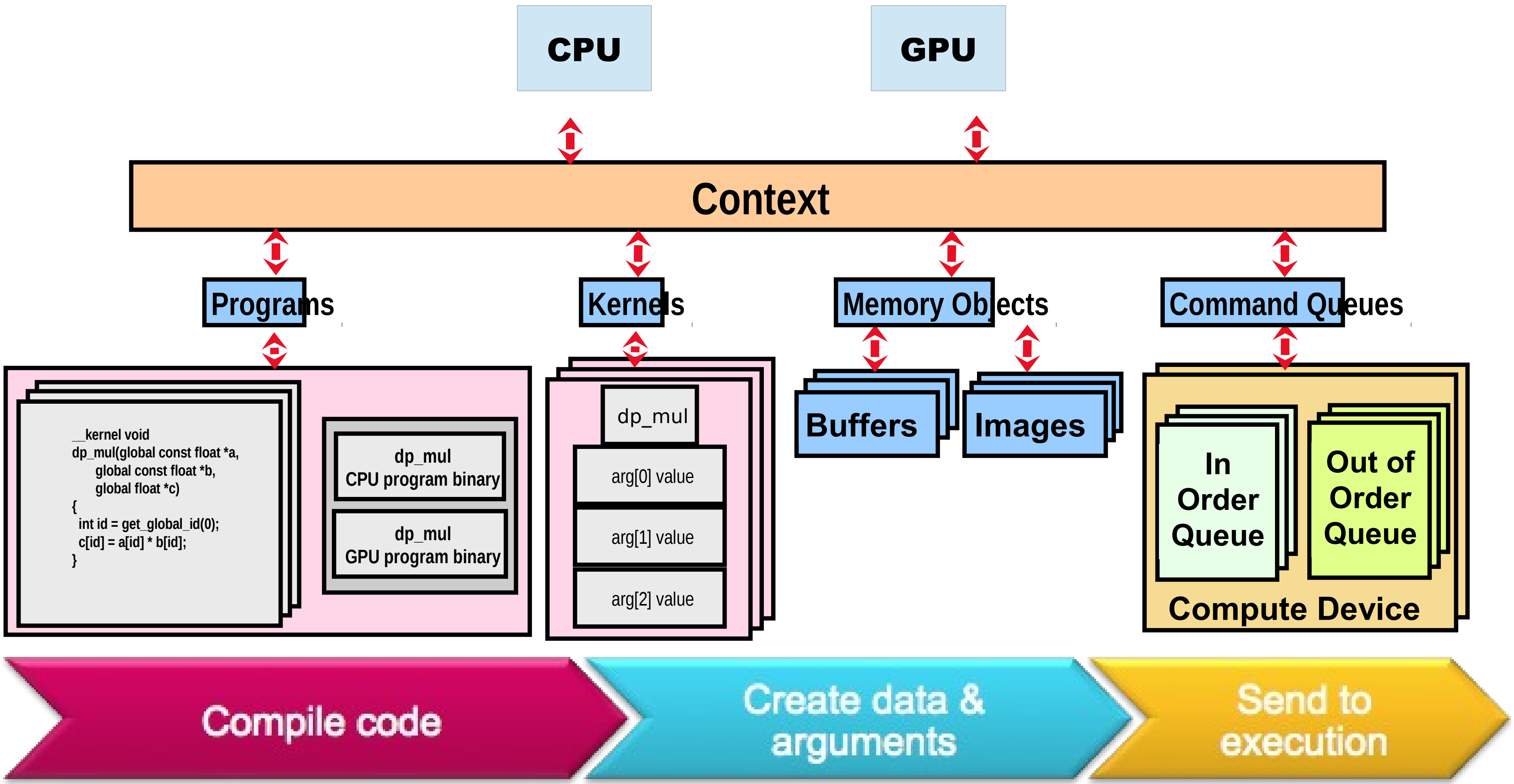
Vector Addition – Host

- The <u>host program</u> is the code that runs on the host to:
  - Setup the environment for the OpenCL program
  - Create and manage kernels
- 5 simple steps in a basic host program:
  1. Define the ***platform*** … platform = devices+context+queues
  2. Create and Build the ***program*** (dynamic library for kernels)
  3. Setup ***memory*** objects
  4. Launch the ***kernel*** (with sizes and arguments)
  5. ***Retrieve*** data: transfer memory objects back to host



As we go over the next set of slides, cross reference content on the slides to the reference card. This will help you get used to the reference card and how to pull information from the card and express it in code.

The basic platform and runtime APIs in OpenCL



1. Define the platform

Use PyOpenCL to manage devices from Python

```
import pyopencl
```

- Create a context:
  - In interactive mode one can immediately create it:

    ```
    ctx = pyopencl.create_some_context()
    ```

  - For programmatic selection of the device:

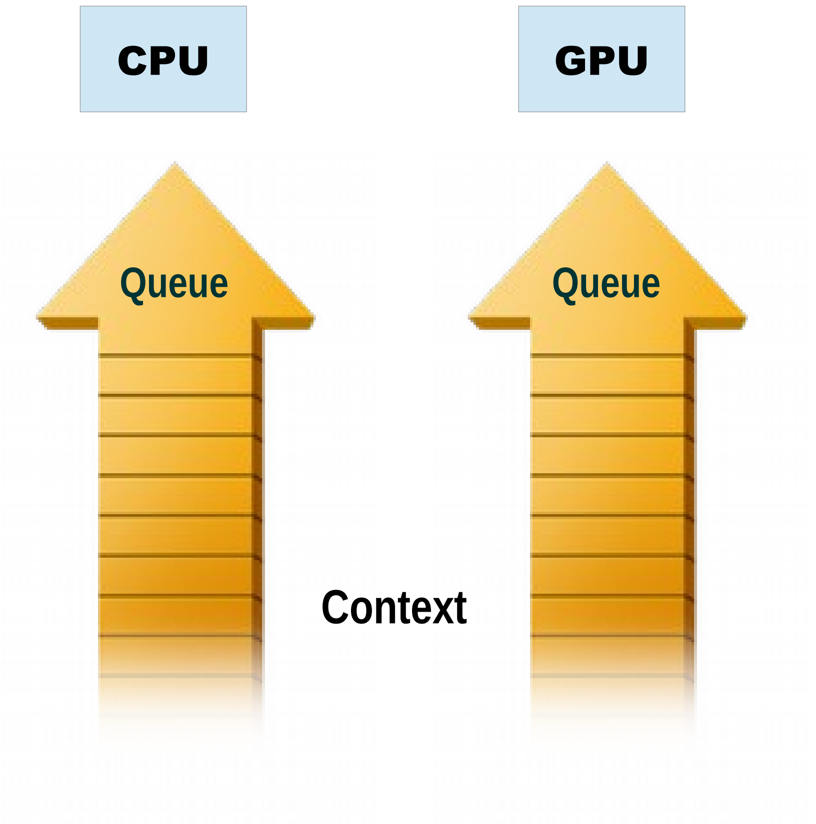
    ```
    platforms = pyopencl.get_platforms()
    devices = platforms[0].get_devices()
    ctx = pyopencl.Context((devices[0],))
    ```

- Create a simple command-queue to feed our device:

```
queue = pyopencl.CommandQueue(ctx)
```

Command-Queues

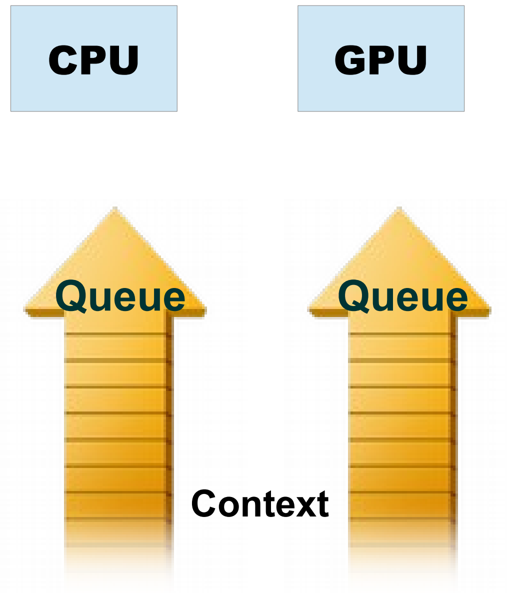
- Commands include:
  - Kernel executions
  - Memory object management
  - Synchronization
- The only way to submit commands to a device is through a command-queue.
- Each command-queue points to a single device within a context.
- Multiple command-queues can feed a single device.
  - Used to define independent streams of commands that don't require synchronization



Command-Queue execution details

Command queues can be configured in different ways to control how commands execute

- *In-order queues*:
  - Commands are enqueued and complete in the order they appear in the program (program-order)
- *Out-of-order queues*:
  - Commands are enqueued in program-order but can execute (and hence complete) in any order.
- Execution of commands in the command-queue are guaranteed to be completed at synchronization points
- Command queue have a profiling flag
  - Events will then have profiling information



2. Create and build the program

- Define source code for the kernel-program as a string literal (great for toy programs) or read from a file (for real applications).

```
src = '''__kernel void vadd(__global const float *a,
  __global const float *b,
  __global       float *c)
{
   int gid = get_global_id(0);
   c[gid]  = a[gid] + b[gid];
}'''
```

- Build the program object:

```
prg = pyopencl.Program(ctx, src)
```

- Compile the program to create a "dynamic library" from which specific kernels can be pulled:

```
prg = prg.build()
```

3. Setup Memory Objects

- For vector addition we need 3 memory objects, one each for input vectors A and B, and one for the output vector C.
- Create input vectors and assign values on the host:

```
import numpy

h_a = numpy.random.random(LENGTH).astype(numpy.float32)
h_b = numpy.random.random(LENGTH).astype(numpy.float32)
h_c = numpy.empty((LENGTH,),dtype=numpy.float32)
```

- Define pyOpenCL arrays memory objects:

```
import pyopencl.array

d_a = pyopencl.array.to_device(queue , h_a)
d_b = pyopencl.array.to_device(queue , h_b)
d_c = pyopencl.array.empty(queue,(LENGTH,),
dtype='float32')
```

Those are built on top of OpenCL Buffers like Numpy is built on top of C buffers.

Conventions for naming buffers

- It can get confusing about whether a host variable is just a numpy array or an PyOpenCL array (which got transferred)

- A useful convention is to prefix the names of your regular **h**ost arrays with "**h_**" and your OpenCL arrays which will live on the **d**evice with "**d_**"

4. Launch the kernel

- The kernel object is a dynamical attribute from the program: `kernel = prg.vadd`
- The three first arguments correspond to:
  - Queue where the job will be enqueued
  - Global dimension size as a tuple of int: (1024, 1024)
  - The workgroup size as a tuple of int: (8, 1)

    The global size has to be a multiple of the workgroup size

    Only some workgroup size are allowed by certain devices (i.e. 1 for apple)
-

4. Launch the kernel (cont.)

Then all other arguments of the kernel function “vadd”:

- PyOpenCL arrays have their buffer in .data
- For simple types, they need to be enforces using numpy:
  - For an `int`: `numpy.int32(4)`

This returns an event `evt`:

- For synchronization: `evt.wait()`
- For profiling at the nano-second level:

```
timing_s = 1e-9*(evt.profile.end - evt.profile.start)
```

5. Retrieve data

- PyOpenCL array object have `.set()` and `.get()` methods
  - Enqueues the memory transfer between numpy and the device
  - Setter is only enqueued
  - Getter is enqueued & immediately synchronized

Exercise 2: Running the Vadd kernel

- Goal:
  - To inspect and verify that you can run an OpenCL kernel
- Procedure:
  - Take the provided Vadd program. It will run a simple kernel to add two vectors together.
  - Look at the host code and identify the API calls in the host code. Compare them against the API descriptions on the OpenCL reference card.
  - There are some helper files which time the execution, output device information neatly and check errors.
- Expected output:
  - A message verifying that the vector addition completed successfully

Hello world in PyOpenCL

```
import numpy, pyopencl

N = 1024

# create context, queue and program
context = pyopencl.create_some_context()
queue = pyopencl.CommandQueue(context)
src = open('vadd.cl').read()
prg = pyopencl.Program(context, src).build()

# create host arrays
h_a = numpy.random.rand(N).astype('float32')
h_b = numpy.random.rand(N).astype('float32')
h_c = numpy.empty(N).astype('float32')

# create device buffers
mf = pyopencl.mem_flags
d_a = pyopencl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=h_a)
d_b = pyopencl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=h_b)
d_c = pyopencl.Buffer(context, mf.WRITE_ONLY, h_c.nbytes)

# run kernel
evt = prg.vadd(queue, h_a.shape, (8,), d_a, d_b, d_c, numpy.uint32(N))

# return results
pyopencl.enqueue_copy(queue, h_c, d_c)

assert numpy.allclose(h_a+h_b, h_c)
```

Explicit buffer declaration

Explicit memory transfer

Port inside_polygon to OpenCL

- Copy your `.c` code into a `.cl` file
- Make the function a `void` function
- Write the result into the right place
  - Use an output array
- Add `__kernel` to declare the function a kernel
- Prefer floats to doubles:
  - many devices don't support it

Nota: in this example we do not care about image dimensions …
… we should !

Python side of OpenCL

- Create the context, the queue
- Allocate input & output buffers
- Read OpenCL source and compile
- Launch the kernel on the grid
- Retrieve the result & display it

Summary of speed-ups

	Execution time (s)	Speed-up
Python with tuples	36	1
Python with numpy	450	0.08
Cython with tuples	9	4
Cython with numpy+Opt	3.6	10
Cython + C	0.27	111
Cython-class + Opt	0.118	305
Cython with OpenMP	0.020	1800
PyOpenCL / GPU	0.003	12000

Measured on a dual-quadcore @2.27 GHz Xeon 5520 + Geforce Titan

PyOpenCL provides

- Clean & Pythonic programming interface to OpenCL
  - Provides garbage collection & buffer re-use
- Complete: exposes all of the C-API
- Removes most of the boiler-plate code
- Implements nice data-structures:
  - pyopencl.array similar to numpy.array
- Implements powerful algorithms:
  - Map,
  - Reduction,
  - Scan, ...

Don't re-invent the polygonal-wheel !!!

http://documen.tician.de/pyopencl/

Parallel programming patterns

- Map: pixel wise transformation
- Scatter / Gather: like convolutions
- Reduce: i.e. max over an array
- Scan: i.e. numpy's cumsum
- Compact: remove unused elements
- Allocate: predict the new occupation
- Sort: multiple // implementation
- Hashing: multiple // implementation

More advanced training on GPU programming:
https://developer.nvidia.com/udacity-cs344-intro-parallel-programming

ESRF | The European Synchrotron