Introduction to pyOpenCL

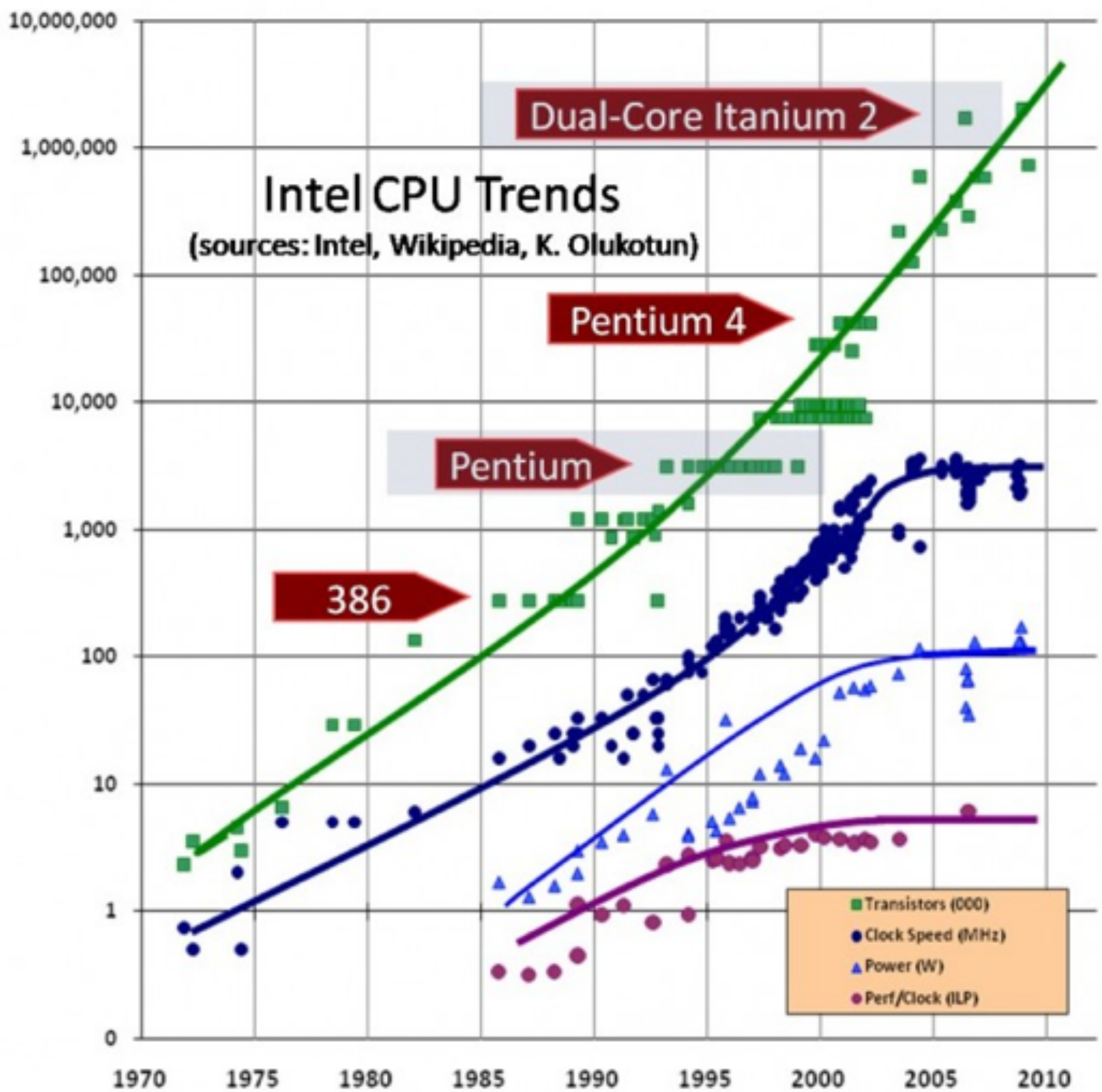
Massimo Di Pierro (DePaul University, Chicago)

Thanks to Andreas Klöckner (author of pyOpenCL)
NVIDIA (for making GPUs mainstream)
Khronos Compute Working Group (for OpenCL)

Contents

- Rationale for OpenCL and GPU programming
- pyOpenCL (python + numpy + opencl)
- Example: 2D Laplace solver
- mdpcl (Python to C99/OpenCL/JavaScript)
- Goal: make Python the only needed language for HPC

Moore's Law



Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)
Dual-Core Itanium 2
Pentium 4
Pentium
386
10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
0
1970
1975
1980
1985
1990
1995
2000
2005
2010
Transistors (000)
Clock Speed (MHz)
Power (W)
Perf/Clock (ILP)

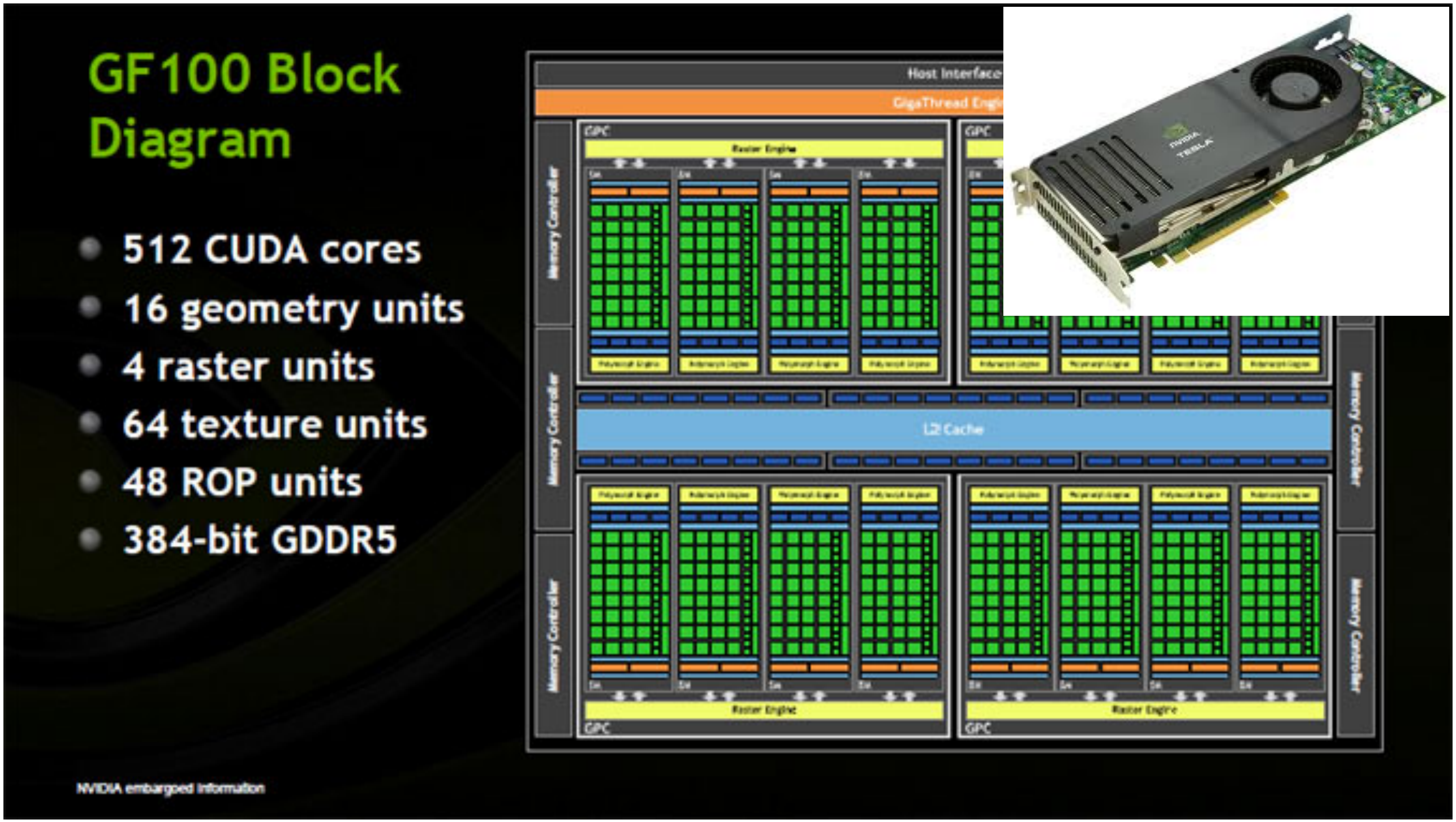
Cray XK7

20 petaFlops

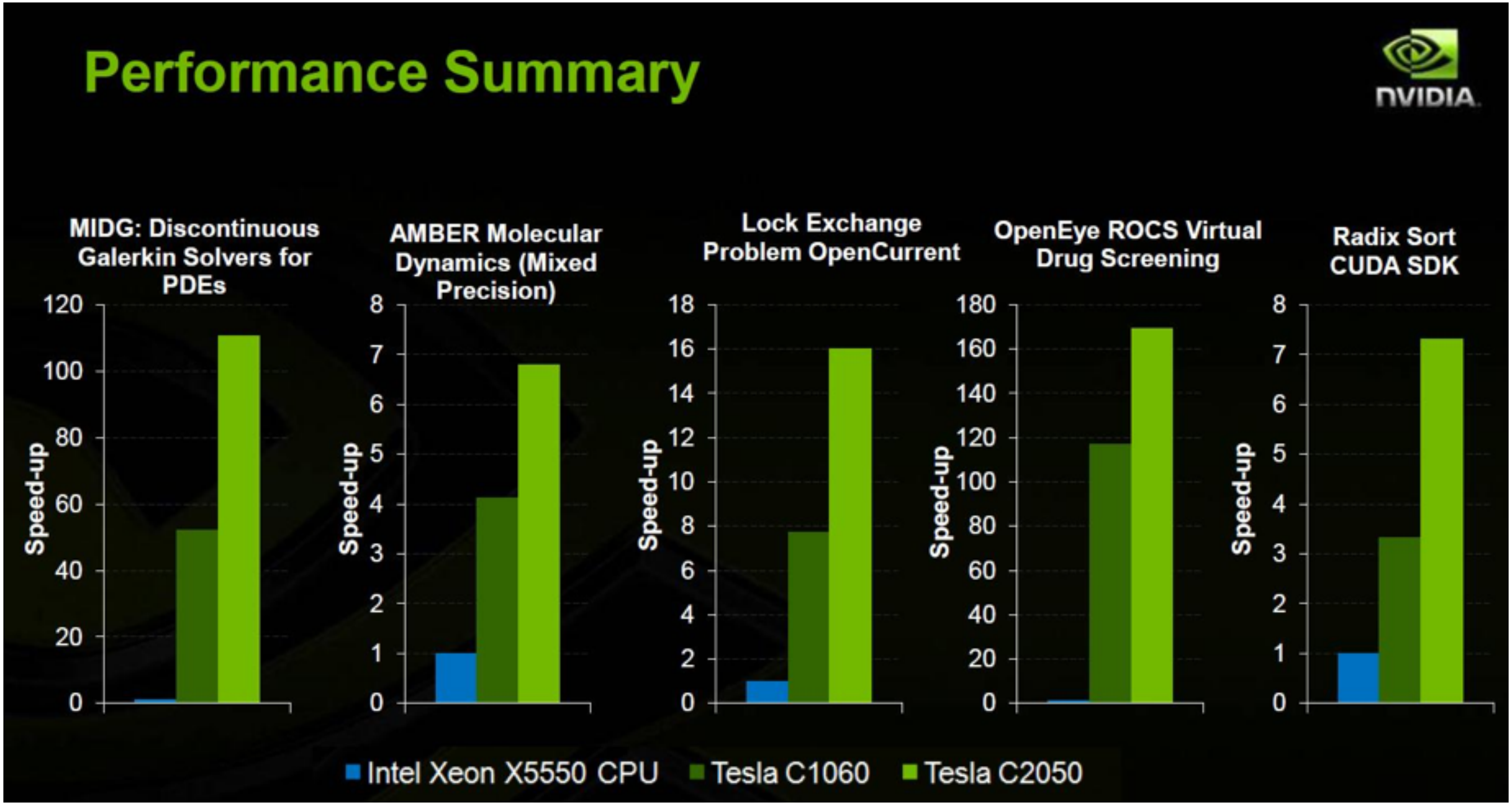
18,688 nodes x

AMD Opteron 6274 CPUs and
Nvidia Telsa K20 GPU

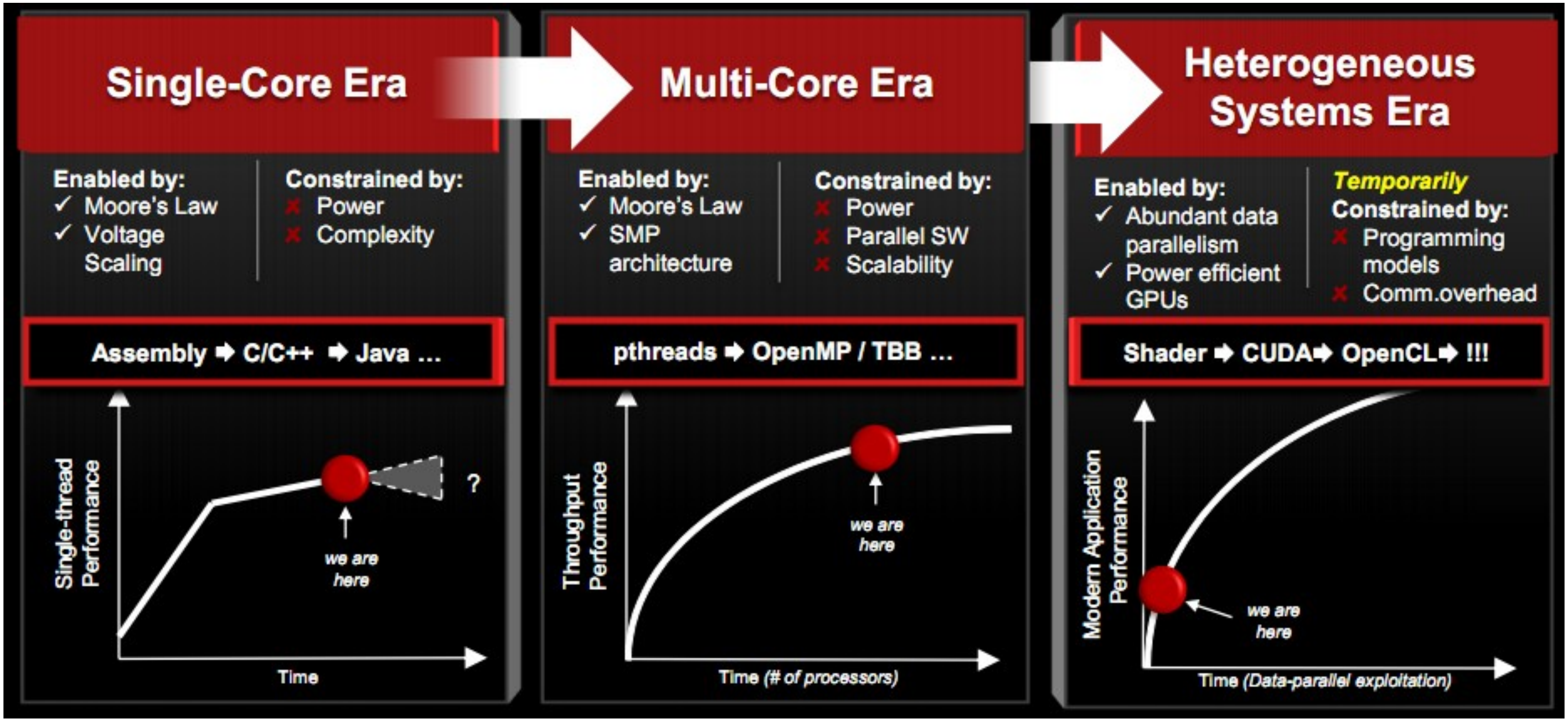
Nvidia Fermi (GF100)



Relative Benchmarks



Heterogenous Systems Era



Programming GPUs

- CUDA: Compute Unified Device Architecture) is a parallel computing architecture developed by NVIDIA
- OpenCL: open, royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers and handheld/embedded devices. It has been adopted by Intel, AMD, Nvidia, and ARM.

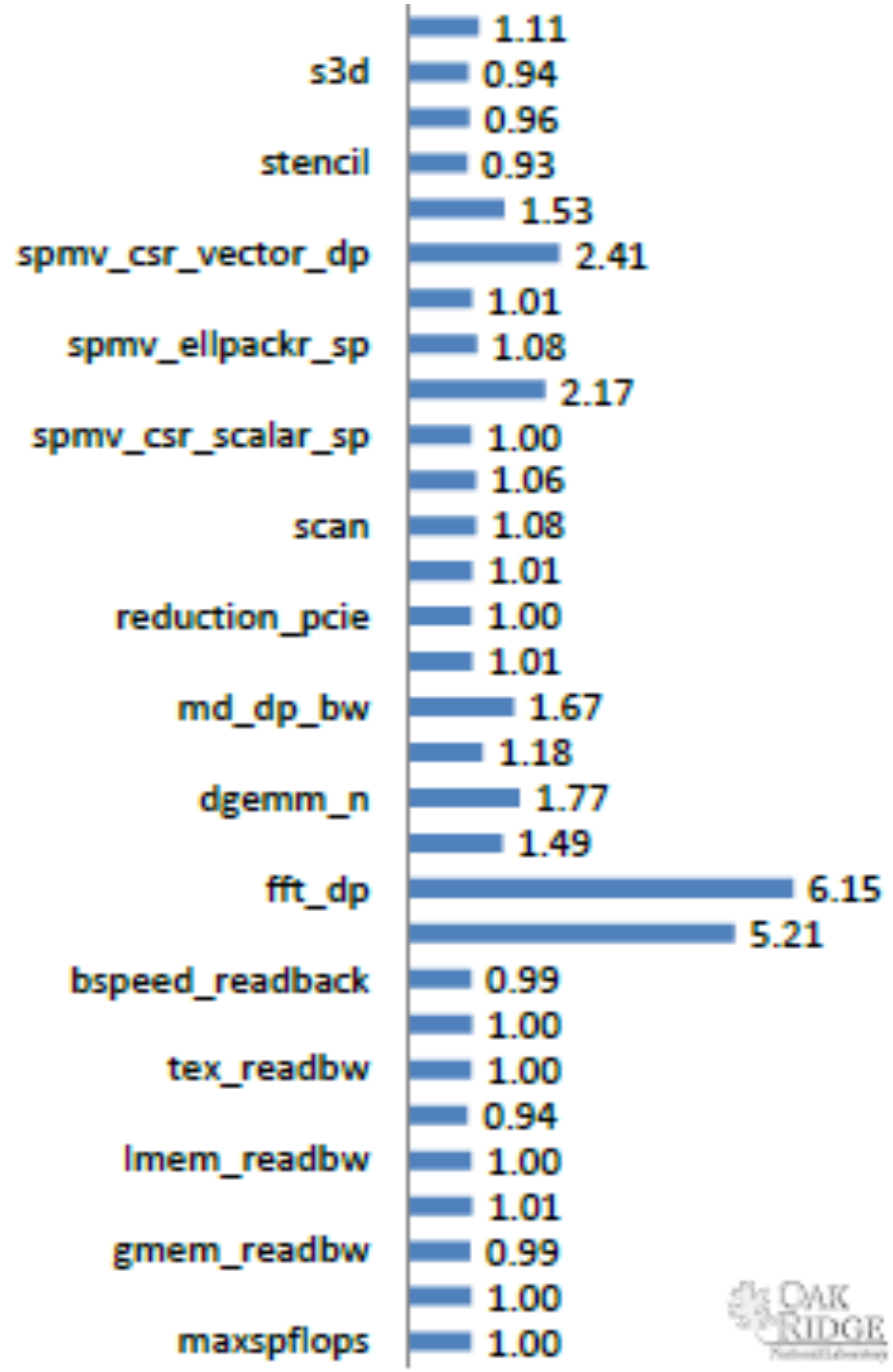
CUDA vs OpenCL (pro)

PROS: C-like language Very well documented Brilliant Debugger Open Compiler Many libraries: Thrust, cuLA, cuBlas, cuFFT, cuSPARSE, etc. CONS: Nvidia GPU only No CPU support	PROS: C-like language (C99) Open Standard Share resources with OpenGL Runs on GPU, CPUs, Android Phones, iPhones Comes with JIT compiler CONS: Debugger and Documentation not as good as CUDA Not always latest drivers

CUDA vs OpenCL

CUDA and OpenCL

- What does performance look like today?
- This chart shows the speedup of CUDA over OpenCL on a single Tesla M2070 on KIDS (CUDA 4.0)
- Note that performance is (in most cases, close to equivalent)
- Cases where it's not tend to be related to texture memory or transcendental intrinsics



The C programming model

```
define a[], b[]
load data into a
for(i=0; i<N; i++)
   b[i] = do_something(i,b[i])
output b[]
```

if so_something(i) can be executed in arbitrary order, then the calls can be executed in parallel, for example different threads.

The OpenCL model

```
define a[], b[]
load data into a
initialize Device
copy a into Device memory
run “do_something” i,a,b on Device
copy b from Device memory
output b[]
```

One task for each i created and queued and executed on the available processing units (cores) “do_something” is called a Kernel and must be

The pyOpenCL model

Python + NumPy + OpenCL = pyOpenCL

Code in Python + NumPy + C99
Run computing intensive parts as OpenCL
Embed Kernels as strings
Use JIT to compile kernels to Device

example0.py (adding vectors)

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

print numpy.linalg.norm(c - (a + b))
```

example1.py (with OpenCL)

```
from device 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)


...
```

example1.py (...continued)

```
...

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

program.sum(device.queue,[n],None,a_buffer,b_buffer,c_buffer)
c = device.retrieve(c_buffer)
print numpy.linalg.norm(c - (a+b))
```

device.py (device interaction)

```python
import pyopencl
import numpy

class Device(object):
    flags = pyopencl.mem_flags
    def __init__(self):
        self.ctx = pyopencl.create_some_context()
        self.queue = pyopencl.CommandQueue(self.ctx)
    def buffer(self,source=None,size=0,mode= pyopencl.mem_flags.READ_WRITE):
        if source is not None: mode = mode | pyopencl.mem_flags.COPY_HOST_PTR
        buffer = pyopencl.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)
        pyopencl.enqueue_copy(self.queue, output, buffer)
        return output
    def compile(self,kernel):
        return pyopencl.Program(self.ctx,kernel).build()
```

device.py (device interaction)

```
import pyopencl
import numpy

class Device(object):
    flags = pyopencl.mem_flags
    def __init__(self):
        self.ctx = pyopencl.create_some_context()
        self.queue = pyopencl.          ue(self.ctx)
    def buffer                                          RITE):
        if sou                                          OST_PTR
        buffer                                          urce)
        return
    def retrie
        output
        pyopencl.enqueue_copy(self.queue, output, buffer)
        return output
    def compile(self,kernel):
        return pyopencl.Program(self.ctx,kernel).build()
```

`pyopencl.`**`create_some_context`**(*interactive=True*)

Create a `Context` 'somehow'.

If multiple choices for platform and/or device exist, *interactive* is True, and *sys.stdin.isatty()* is also True, then the user is queried about which device should be chosen. Otherwise, a device is chosen in an implementation-defined manner.

device.py (device interaction)

```
import pyopencl
import numpy

class Device(object):
    flags = pyopencl.mem_flags
    def __init__(self):
        self.ctx = pyopencl.create_some_context()
        self.queue = pyopencl.CommandQueue(self.ctx)
    def buffer(self,source=None,si         mode= pyopencl.mem_flags.READ_WRITE):
        if source is not N                   yopencl.mem_flags.COPY_HOST_PTR
        buffer = p                                                       e)
        return bu
    def retrieve(
        output = numpy.zeros(shape or buffer.size/4,dtype=dtype)
        pyopencl.enqueue_copy(self.queue, output, buffer)
        return output
    def compile(self,kernel):
        return pyopencl.Program(self.ctx,kernel).build()
```

class pyopencl.**CommandQueue**(*context*, *device=None*, *properties=None*)

Create a new command queue. *properties* is a bit field consisting of `command_queue_properties` values.

device.py (device interaction)

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

Create a `Buffer`. See `mem_flags` for values of *flags*. If *hostbuf* is specified, *size* defaults to the size of the specified buffer if it is passed as zero.

`Buffer` is a subclass of `MemoryObject`.

Note that actual memory allocation in OpenCL may be deferred. Buffers are attached to a `Context` and are only moved to a device once the buffer is used on that device. That is also the point when out-of-memory errors will occur. If you'd like to be sure that there's enough memory for your allocation, either use `enqueue_migrate_mem_objects()` (if available) or simply perform a small transfer to the buffer. See also `pyopencl.tools.ImmediateAllocator`.

```
import pyopencl
import numpy

class Device(ob
    flags = py
    def __init_
        self.ct
        self.qu
    def buffer(self,source=None,size=0,mode= pyopencl.mem_flags.READ_WRITE):
        if source is not None: mode = mode | pyopencl.mem_flags.COPY_HOST_PTR
        buffer = pyopencl.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)
        pyopencl.enqueue_copy(self.queue, output, buffer)
        return output
    def compile(self,kernel):
        return pyopencl.Program(self.ctx,kernel).build()
```

device.py (device interaction)

```
import pyopencl
import numpy

class Devi
    flags =
    def __i
        sel
        sel
    def buf                                                        D_WRITE):
        if                                                         Y_HOST_PTR
        buf                                                        =source)
        ret
    def retrieve(self,buffer,shape=None,dtype=numpy.float32):
        output = numpy.zeros(shape or buffer.size/4,dtype=dtype)
        pyopencl.enqueue_copy(self.queue, output, buffer)
        return output
    def compile(self,kernel):
        return pyopencl.Program(self.ctx,kernel).build()
```

`pyopencl.`**`enqueue_copy`**(*queue, dest, src, **kwargs*)

Copy from `Image`, `Buffer` or the host to `Image`, `Buffer` or the host. (Note: host-to-host copies are unsupported.)

The following keyword arguments are available:

Parameters:	• **wait_for** – (optional, default empty) • **is_blocking** – Wait for completion. Defaults to *True*. (Available on any copy involving host memory)
Returns:	A `NannyEvent` if the transfer involved a host-side buffer, otherwise an `Event`.

device.py (device interaction)

```
import pyopencl
import numpy

class Devic
    flags =
    def __i
        sel
        sel
    def buf                                          D_WRITE):
        if                                           Y_HOST_PTR
        buf                                          =source)
        ret
    def ret
        out
        pyo
        return output
    def compile(self,kernel):
        return pyopencl.Program(self.ctx,kernel).build()
```

class pyopencl.**Program**(*context, src*)

class pyopencl.**Program**(*context, devices, binaries*)

binaries must contain one binary for each entry in *devices*.

info

Lower case versions of the program_info constants may be used as attributes on instances of this class to directly query info attributes.

get_info(*param*)

See program_info for values of *param*.

get_build_info(*device, param*)

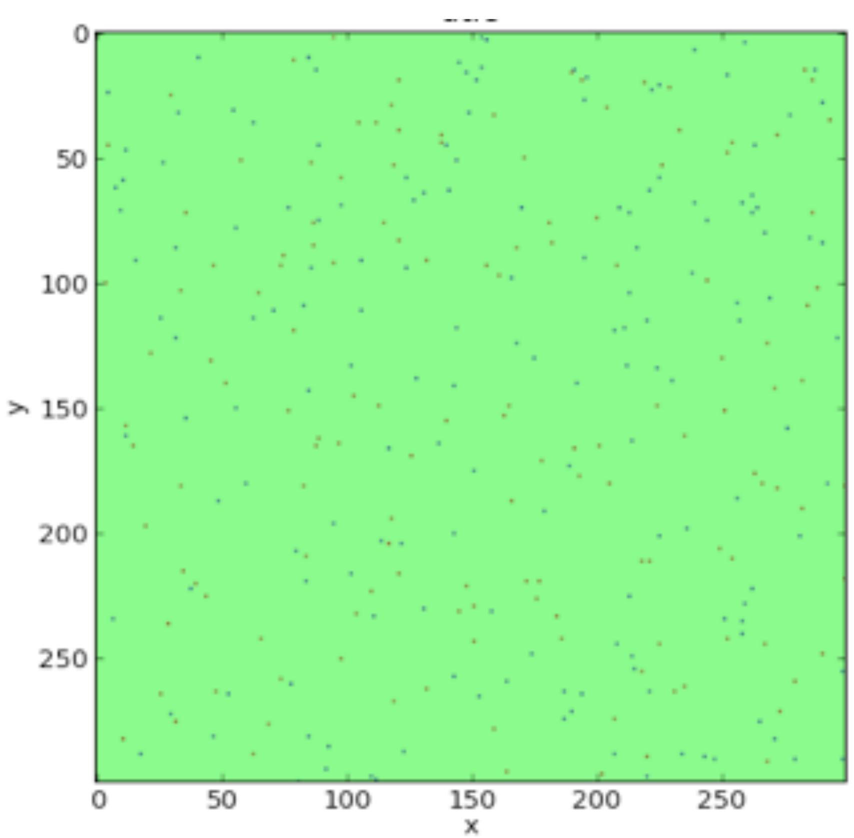
See program_build_info for values of *param*.

build(*options=[], devices=None*)

options is a string of compiler flags. Returns *self*.

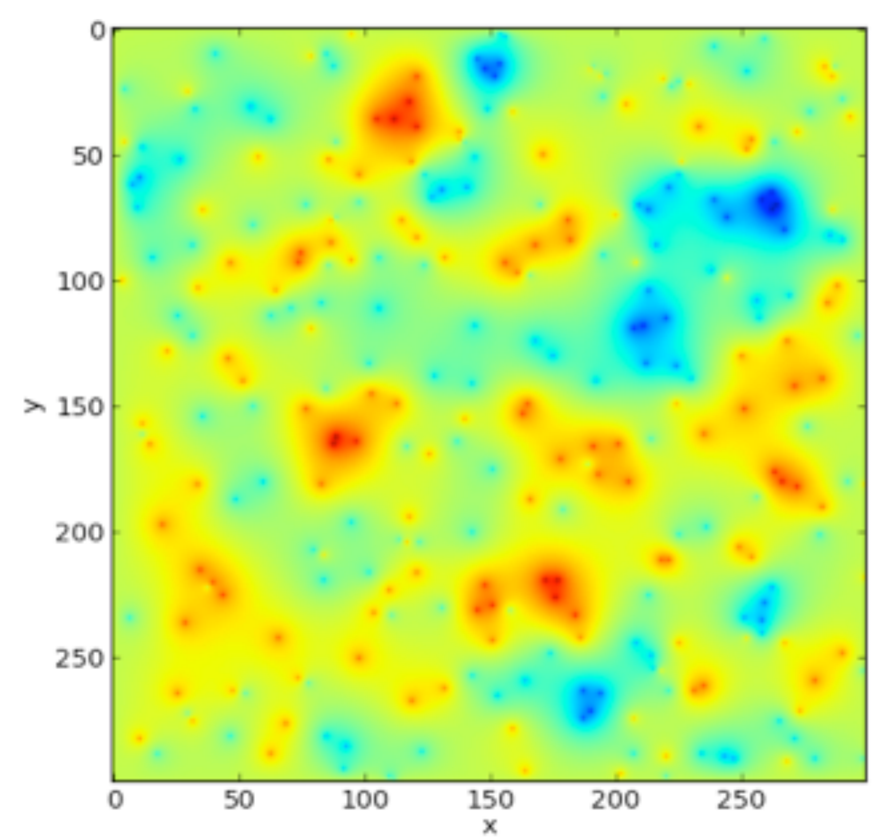
Solving a 2D Laplace Eq.

Input: distribution of charge (q)



$$\Delta u = q$$

Output: electrostatic potential (u)



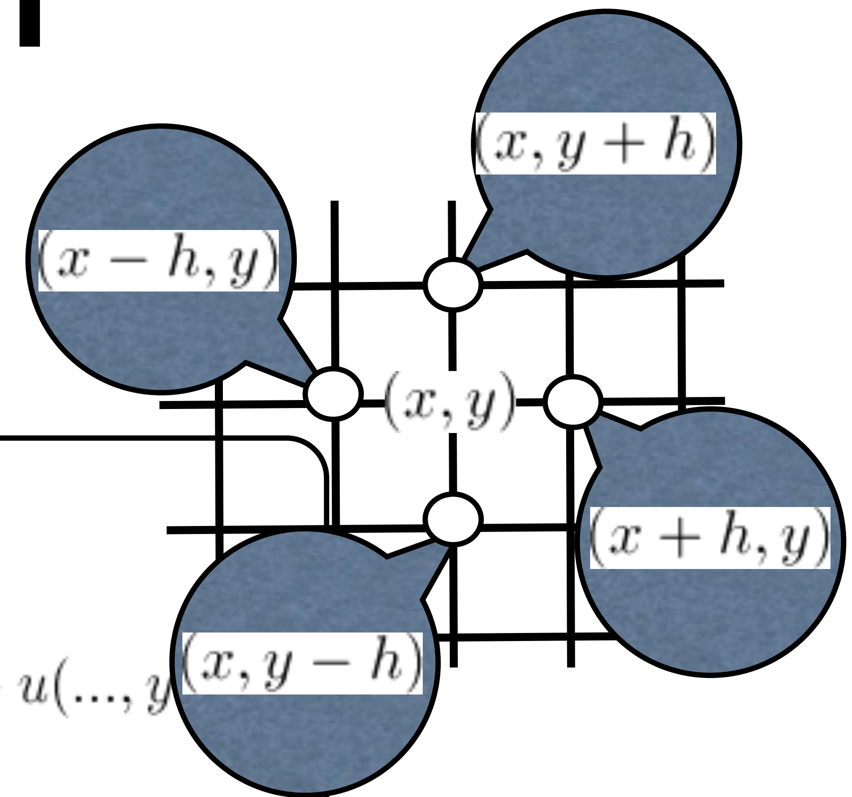
Solving a 2D Laplace Eq.

$$\Delta u = q$$

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

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

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



$$u(x+h,y) + u(x,y+h) - 4u(x,y) + u(x,y-h) + u(x-h,y) = q(x,y)$$

$$u(x,y) = 1/4\{u(x+h,y) + u(x,y+h) + u(x,y-h) + u(x-h,y) - q(x,y)\}$$

```
w[site] = 1.0/4.0*(u[up]+u[down]+u[left]+u[right] - q[site]);
u[site] = w[site]
```

example2.py (Laplace)

```
from device import Device
from canvas import Canvas
import random
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[random.randint(1, n-1),random.randint(1, n-1)] = random.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)

...
```

example2.py (continued)

```
from mdpcl import Device
...

program = 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 site = y*WIDTH + x, up, down, left, right;
          if(y!=0 && y!=WIDTH-1 && x!=0 && x!=WIDTH-1) {
             up=site+WIDTH; down=site-WIDTH; left=site-1; right=site+1;
             w[site] = 1.0/4.0*(u[up]+u[down]+u[left]+u[right] - q[site]);
          }
       }
    """.replace('WIDTH',str(n)))

...
```

example2.py (continued)

```
...
      __kernel void solve(__global float *w,
                          __global const float *u,
                          __global const float *q) {
....
```

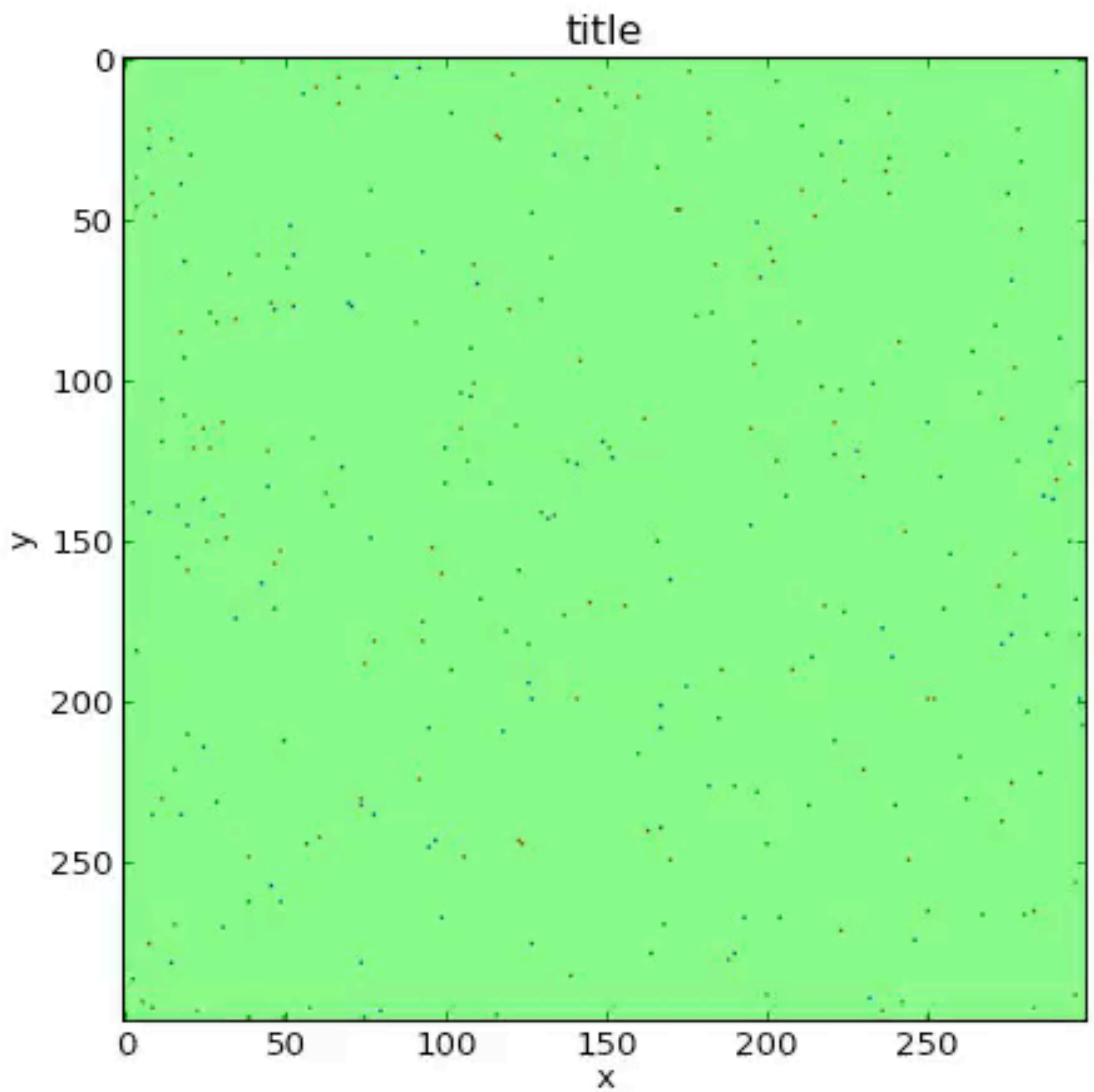
```
...

for k in range(1000):
    program.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()
```

Results



~~device.py~~ > mdpcl.py

- Goal: write only Python code and generate OpenCL code from Python code at runtime.
- Use a decorator to tag code for running with OpenCL
- `easy_install mdpcl`
- mdpcl maps a strongly typed language (Python) into statically types language (C99/OpenCL/JavaScript)

example2.py (previous)

```
from mdpcl import Device
...

program = 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 site = y*WIDTH + x, up, down, left, right;
          if(y!=0 && y!=WIDTH-1 && x!=0 && x!=WIDTH-1) {
             up=site+WIDTH; down=site-WIDTH; left=site-1; right=site+1;
             w[site] = 1.0/4.0*(u[up]+u[down]+u[left]+u[right] - q[site]);
          }
       }
    """.replace('WIDTH',str(n)))

...
```

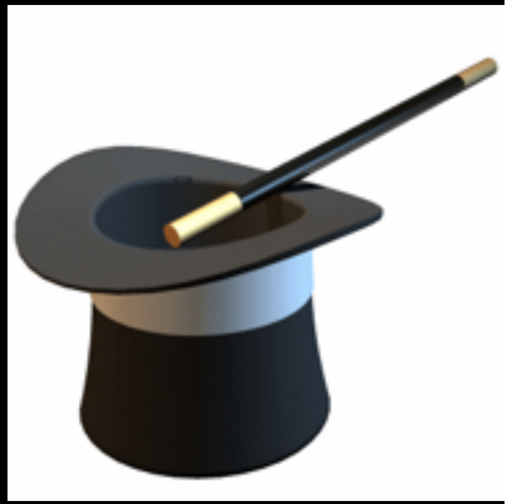
example3.py (new)

```
from mdpcl import Device
...

@device.compiler('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))
    site = new_int(x*n+y)
    if y!=0 and y!=n-1 and x!=0 and x!=n-1:
        up = new_int(site-n)
        down = new_int(site+n)
        left = new_int(site-1)
        right = new_int(site+1)
        w[site] = 1.0/4*(u[up]+u[down]+u[left]+u[right] - q[site])

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

...
```



example3.py (new)

```
@device.compiler('kernel',w='global:ptr_float',
                          u='global:const:ptr_float',
                          q='global:const:ptr_float')
def solve(w,u,q):...
```

- decorator specifies type of function ("kernel") and type of arguments,
- "global:ptr_float" means "__global float*".
- variables must be declared

```
x = new_int(get_global_id(0))
```

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

Resources

- http://www.khronos.org/opencl/
- http://documen.tician.de/pyopencl/
- http://wiki.tiker.net/PyOpenCL (installation info)
- http://mathema.tician.de/software/pycuda
- https://github.com/mdipierro/mdpcl