-
-
-
-
-

-
-
-
-

```
import pyopencl as cl

print('\n' + '=' * 60 + '\nOpenCL Platforms and Devices')
# Print each platform on this computer
for platform in cl.get_platforms():
    print('=' * 60)
    print('Platform - Name:  ' + platform.name)
    print('Platform - Vendor:  ' + platform.vendor)
    print('Platform - Version:  ' + platform.version)
    print('Platform - Profile:  ' + platform.profile)
    # Print each device per-platform
    for device in platform.get_devices():
        print('    ' + '-' * 56)
        print('    Device - Name:  ' + device.name)
        print('    Device - Type:  ' +
cl.device_type.to_string(device.type))
        print('    Device - Max Clock Speed:  {0}
Mhz'.format(device.max_clock_frequency))
        print('    Device - Compute Units:
{0}'.format(device.max_compute_units))
        print('    Device - Local Memory:  {0:.0f}
KB'.format(device.local_mem_size/1024.0))
        print('    Device - Constant Memory:  {0:.0f}
KB'.format(device.max_constant_buffer_size/1024.0))
        print('    Device - Global Memory: {0:.0f}
GB'.format(device.global_mem_size/1073741824.0))
        print('    Device - Max Buffer/Image Size: {0:.0f}
```

```
=============================================================
OpenCL Platforms and Devices
=============================================================
Platform - Name:  Intel(R) OpenCL HD Graphics
Platform - Vendor:  Intel(R) Corporation
Platform - Version:  OpenCL 2.1
Platform - Profile:  FULL_PROFILE
    -------------------------------------------------------
    Device - Name:  Intel(R) Gen9 HD Graphics NEO
    Device - Type:  GPU
    Device - Max Clock Speed:  1050 Mhz
    Device - Compute Units:  24
    Device - Local Memory:  64 KB
    Device - Constant Memory:  4194296 KB
    Device - Global Memory: 25 GB
    Device - Max Buffer/Image Size: 4096 MB
    Device - Max Work Group Size: 256
=========================================
```

-
  -
  -
    - 

```
import numpy as np
import pyopencl as cl

a_np = np.random.rand(50000).astype(np.float32)
b_np = np.random.rand(50000).astype(np.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

mf = cl.mem_flags
a_g = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR,
hostbuf=a_np)
b_g = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR,
hostbuf=b_np)

prg = cl.Program(ctx, """
__kernel void sum(
    __global const float *a_g, __global const float *b_g,
__global float *res_g)
{
  int gid = get_global_id(0);
  res_g[gid] = a_g[gid] + b_g[gid];
}
""").build()

res_g = cl.Buffer(ctx, mf.WRITE_ONLY, a_np.nbytes)
prg.sum(queue, a_np.shape, None, a_g, b_g, res_g)

res_np = np.empty_like(a_np)
cl.enqueue_copy(queue, res_np, res_g)

# Check on CPU with Numpy:
print(res_np - (a_np + b_np))
print(np.linalg.norm(res_np - (a_np + b_np)))
assert np.allclose(res_np, a_np + b_np)
```

```
buf = cl . Buffer ( context , flags , size =0, hostbuf=None)
```

-
-
-
  -
  -
-
-

```
buf = cl . Buffer ( context , flags , size =0, hostbuf=None)
```

-
-
  -
-
  -
  -
-
  -

```
prg = cl.Program(context, src)
```

-
-

```
prg.build(options="",devices=None)

kernel = prg.kernelname(queue,(Gx,Gy,Gz), (Lx,Ly,Lz), *args)
```

-
-
  -
  -
  -

```
kernel = prg.kernelname(queue,(Gx,Gy,Gz), (Lx,Ly,Lz), *args)
```

-
-
-

```
kernel = """
__kernel void sum(
    __global float* a,
    __global float* b,
    __global float* c,
    const unsigned int count)
{
   //  your code here
}

"""
vector_size = 1024
context = cl.create_some_context()
queue = cl.CommandQueue(context)

h_a = np.random.rand(vector_size).astype(np.float32)
h_b = np.random.rand(vector_size).astype(np.float32)
h_c = np.empty(vector_size).astype(np.float32)

d_a = cl.Buffer(context, cl.mem_flags.READ_ONLY |
cl.mem_flags.COPY_HOST_PTR, hostbuf=h_a)
d_b = cl.Buffer(context, cl.mem_flags.READ_ONLY |
cl.mem_flags.COPY_HOST_PTR, hostbuf=h_b)
d_c = cl.Buffer(context, cl.mem_flags.WRITE_ONLY, h_c.nbytes)

program = cl.Program(context, kernel).build()
# Utiliser obligatoirement une variable pour utiliser une seule
instance (cf. doc)
prg = program.sum
prg.set_scalar_arg_dtypes([None, None, None, np.uint32])
prg(queue, h_a.shape, None, d_a, d_b, d_c, vector_size)
// ...
```

- **None**
-

```
printf("Hello from kernel #%d, got value: %d\n", global_id,
values[global_id]);
```

-
  -
  -
  -
  -
-
-
-
-

```
import pyopencl as cl
import numpy

A = numpy.random.rand(1000).astype(numpy.float32)
B = numpy.random.rand(1000).astype(numpy.float32)
C = numpy.empty_like(A)

ctx = cl.Context()
queue = cl.CommandQueue(ctx)

A_buf = cl.Buffer(ctx, cl.mem_flags.READ_ONLY |
cl.mem_flags.COPY_HOST_PTR, hostbuf=A)
B_buf = cl.Buffer(ctx, cl.mem_flags.READ_ONLY |
cl.mem_flags.COPY_HOST_PTR, hostbuf=B)
C_buf = cl.Buffer(ctx, cl.mem_flags.WRITE_ONLY, A.nbytes)
prg = cl.Program(ctx, """
  __kernel
  void sum(__global const float* a, __global const float* b,
__global float* c){
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
  }
  """).build()
prg = cl.Program(ctx,kernel).build()
prg.sum(queue, A.shape, A_buf, B_buf, C_buf)
cl.enqueue_copy(queue, C, C_buff)
```

-
  -

```
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];
}
```

```
__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];
}
```

-
-
-
-

-
-
-
  -
-
  -
-
  -

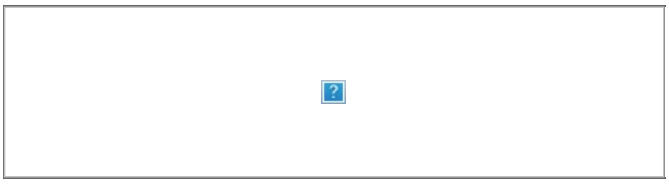
-
-
  -
  -
    -
    -
    -
-
-

-
-
-
-

-
-
-
-

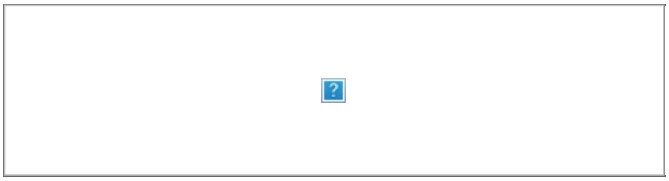
-
  -
  -
  -
-
  -
  -
  -

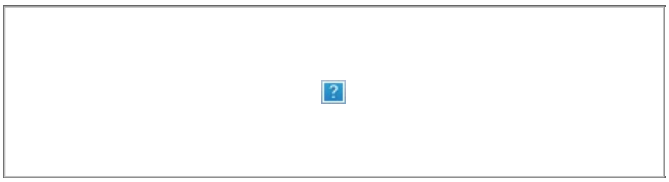
-
-
-

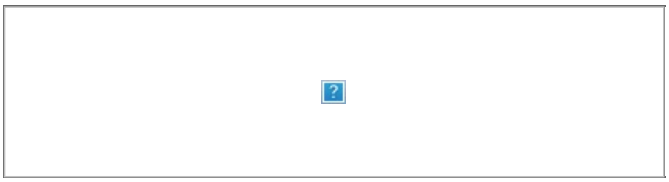


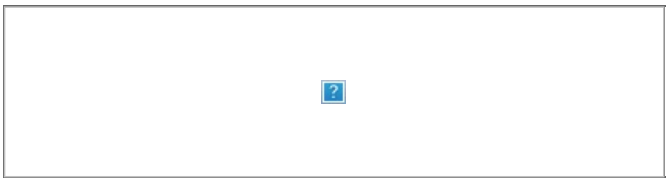
-
  -

-









-
  -

```
barrier(CLK_LOCAL_MEM_FENCE)
```

```
barrier(CLK_GLOBAL_MEM_FENCE)
```

-
-

```
void mat_mul(int N, float *A, float *B, float *C) {
    int i, j, k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            C[i*N+j] = 0.0f;
            for (k = 0; k < N; k++) {
            C[i*N+j] += A[i*N+k] * B[k*N+j];
            }
        }
    }
}
```

```
1   __kernel void mat_mul(const int N, __global float *A,
    __global float *B, __global float *C) {
2       int i, j, k;
3       // Les boucles sont transformées en calculs parallèles
4       i = get_global_id(0);
5       j = get_global_id(1);
6       // il ne reste que le produit scalaire
7      for (k = 0; k < N; k++) {
8          C[i*N+j] += A[i*N+k] * B[k*N+j];
9      }
10  }
```

```
1  __kernel void mmul(const int N,__global float *A,__global
   float *B,
2  __global float *C) {
3      int k;
4      int i = get_global_id(0);
5      int j = get_global_id(1);
6      float tmp = 0.0f;
7      for (k = 0; k < N; k++)
8          tmp += A[i*N+k]*B[k*N+j];
9     C[i*N+j] += tmp;
10 }
```

-
-

-

```
__kernel void mmul(const int N, __global float *A, __global
float *B,
__global float *C) {
    int j, k;
    int i = get_global_id(0);
    float tmp;
    for (j = 0; j < N; j++) {
        tmp = 0.0f;
        for (k = 0; k < N; k++)
            tmp += A[i*N+k]*B[k*N+j];
        C[i*N+j] = tmp;
        }
}
```

-
-

```
1  __kernel void mmul(const int N, __global float *A, __global
   float *B,
2                     __global float *C) {
3    int k;
4    int j;
5    int i = get_global_id(0);
6    float tmp;
7    float Awrk[1024]; // Mémoire privée
8
9    // copie de la ligne de A en memoire privée
10   for (k = 0; k < N; k++) {
11     Awrk[k] = A[i * N + k];
12   }
13
14   // Multiplication d'une ligne entière
15   for (j = 0; j < N; j++) {
16     tmp = 0.0f;
17     for (k = 0; k < N; k++) {
18       tmp += Awrk[k] * B[k * N + j];
19     }
20     C[i * N + j] = tmp;
21   }
22 }
```

```
1   __kernel void mmul(const int N, __global float *A, __global
    float *B,
2                      __global float *C, __local float *Bwrk)
    {
3     int k, j;
4     int i = get_global_id(0);
5     int iloc = get_local_id(0);
6     int nloc = get_local_size(0);
7
8     // Copy the row of A into private memory.
9     float Awrk[256];
10    float tmp;
11
12    for (k = 0; k < N; k++)
13          Awrk[k] = A[i * N + k];
14
15    for (j = 0; j < N; j++) {
16      // Initialisation de la colonne
17      for (k = iloc; k < N; k += nloc)
18          Bwrk[k] = B[k * N + j];
19
20     barrier(CLK_LOCAL_MEM_FENCE);
21
22      tmp = 0.0f;
23      for (k = 0; k < N; k++) {
24          tmp += Awrk[k] * Bwrk[k];
25      }
26      C[i * N + j] = tmp;
27
28      barrier(CLK_LOCAL_MEM_FENCE);
29    }
30  }
```

-

```
atomic_add, atomic_sub, atomic_xchg, atomic_cmpxchg,
atomic_inc, atomic_and,
atomic_or, atomic_xor, atomic_dec, atomic_max, atomic_min
```

```
__local int n;
if (get_local_id (0) == 0) {
    n = 0;
}
barrier (CLK_LOCAL_MEM_FENCE);
//...
atomic_inc (&n);
//...
}
```

```
import pyopencl as cl
import numpy as np

context = cl.create_some_context()
queue = cl.CommandQueue(context)

# 2 pyopencl arrays initialisées aléatoirement
a = cl.array.to_device(queue,
np.random.rand(50000).astype(np.float32))
b = cl.array.to_device(queue,
np.random.rand(50000).astype(np.float32))
# et le résultat
c = cl.array.empty_like(a)

program = cl.Program(context, """
__kernel void sum(__global const float *a, __global const float
*b, __global float *c)
{
  int i = get_global_id(0);
  c[i] = a[i] + b[i];
}""").build()

# Place le programme dans la file (queue) et récupére le
résultat dans c
program.sum(queue, a.shape, None, a.data, b.data, c.data)
```

$$y_i=f_i(x_i) \text{, où :}$$

- f_i
- $f_i=...=f_N=f$
- $i \in \{1,...,N\}$

```
from pyopencl.elementwise import ElementwiseKernel

n = 10
a_np = np.random.randn(n).astype(np.float32)
b_np = np.random.randn(n).astype(np.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

a_g = cl.array.to_device(queue, a_np)
b_g = cl.array.to_device(queue, b_np)

lin_comb = ElementwiseKernel(ctx,
    "float k1, float *a_g, float k2, float *b_g, float
*res_g",
    "res_g[i] = k1 * a_g[i] + k2 * b_g[i]",
    "lin_comb"
)

res_g = cl.array.empty_like(a_g)
lin_comb(2, a_g, 3, b_g, res_g)
```

$$y=f(\cdots f(f(x_1,x_2),x_3),...,x_N)$$

ReductionKernel

-
-
-
-
-

```
a = pyopencl.array.arange(queue, 400, dtype=numpy.float32)
b = pyopencl.array.arange(queue, 400, dtype=numpy.float32)

krnl = ReductionKernel(ctx, numpy.float32, neutral="0",
        reduce_expr="a+b", map_expr="x[i]*y[i]",
        arguments="__global float *x, __global float *y")

prod = krnl(a, b).get() # recupère le numpy array
```

-
-

```
int4 v_iA = (int4)(7, -3, -2, 5);
// ou int4 v_iA = {7, -3, -2, 5};

int4 v_iB = (int4)(1, 2, 3, 4);
int4 v_iC = v_iA + v_iB;
```

```
float4 pos = (float4)(4.0f, 3.0f, 2.0f, 1.0f);
float4 reverse = pos.wzyx; //reverse = (1.0f, 2.0f, 3.0f, 4.0f)
float4 duplicate = pos.xxyy; //duplicate = (4.0f, 4.0f, 3.0f,
3.0f)
```

[]

```
int4 a;
a.s0 = 2; // ou a[0] = 2
```

```
int2 a=(int2)(1,2);
int2 b=(int2)(3,4);

bool sup = all(a<b); // true
```

```
int4  i;
*uint4 u = (uint4)i;  // compile error

// OK si le type de base est le même
float4 f = 1.0f;
float4 va = (float4)f;  // va is a float4 vector
                        // with elements ( f, f, f, f )

// ou avec un scalaire
float deux = 2.0f;
int2 vc = (int2)deux;   // vc is an int2 vector with elements
                        // ( (int)deux, (int)deux )
```

```
uchar4 u;
int4 c = convert_int4(u);

float4 f = (float4)(-5.0f, 254.5f, 254.6f, 1.2e9f);
uchar4 c = convert_uchar4_sat_rte(f); // c contient :
// ((uchar)0, (uchar)254,(uchar)255,(uchar)255)
```

- sat
- rte

---

pyopencl

```
ImageFormat([channel_order, channel_type])
```

```
 Image(context, flags, format, shape=None, pitches=None,
hostbuf=None, is_array=False, buffer=None)
```

-
-
  - ~~image2d_t img[2]~~ image2d_array_t
  - struct

- image1d_t image2d_t image3d_t
- __read_only
__write_only
__read_write
-

-
  - read_imageui, read_imagei, read_imagef
    write_imageui, write_imagei, write_imagef
  - float4, int4, uint4
  - cl_image_format
    -
    -

CL_SNORM_INT8, CL_UNORM_INT8, CL_SNORM_INT16, CL_UNORM_INT16, CL_UNSIGNED_ INT8, CL_UNSIGNED_INT16, CL_UNSIGNED_INT32, CL_SIGNED_INT8, CL_SIGNED_ INT16, CL_SIGNED_INT32, CL_HALF_FLOAT, CL_FLOAT

```
numpy.asarray(...,order='F')
ary.T.copy()
```

```
import pyopencl as cl
from PIL import Image

im = Image.open("image.png")
buffer = im.tobytes()

clImageFormat =
cl.ImageFormat(cl.channel_order.RGBA,cl.channel_type.UNORM_INT8
)

clImage = cl.Image(context,
                   cl.mem_flags.READ_ONLY |
cl.mem_flags.COPY_HOST_PTR,
                   clImageFormat,
                   im.size,
                   None,
                   buffer
                  )
```

```
from imageio import imread, imsave

#Read in image
im = imread('image.png').astype(np.float32) # type au choix...
```

```
__kernel void copie(__read_only image2d_t src,
    __write_only image2d_t dest,
    const sampler_t sampler,
    const float sat)
    {
        int row_id = get_global_id(0);
        int col_id = get_global_id(1);

        int2 coords;
        coords.y = row_id;
        coords.x = col_id;

        float4 pixel = read_imagef(src,sampler,coords);

        // simple copie des trois premiers plans
        write_imagef(dest,coords,pixel);
    }
```

read_image sampler_t

- CLK_NORMALIZED_COORDS_TRUE
  CLK_NORMALIZED_COORDS_FALSE

- read_image

  - CLK_ADDRESS_MIRRORED_REPEAT CLK_ADDRESS_REPEAT
  - CLK_ADDRESS_NONE

- filter Mode
  - CLK_FILTER_NEAREST CLK_FILTER_LINEAR
- __read_write
_____

```
float4 read_imagef(image2d_t image,sampler_t sampler, float2
coord)

float4 read_imagef(image2d_t image,sampler_t sampler,int2
coord)

uint4 read_imageui(image2d_t image, sampler_t sampler,int2
coord)

int4 read_imagei(image2d_t image, sampler_t sampler,int2 coord)

...
```

float,int,uint

float2

```
void write_imagef(image2d_t image, int2 coord, float4 color)

void write_imagei(image2d_t image, int2 coord, int4 color)

...
```

- _____
- _____
_____