OpenCL

Open Computing Language

-Memory architecture

-language sintax

André D. Mora & José M. Fonseca
TAPDI - 2024/2025 DEEC/FCT/UNL

Memory model

- Registers, private, local, global, constant, texture

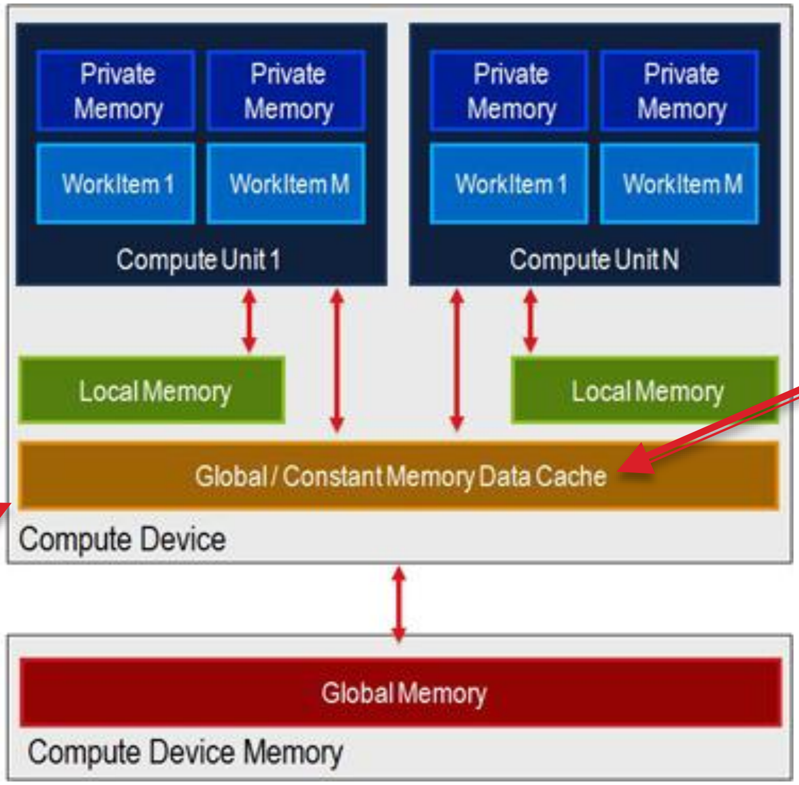
Memory organization

Not shared and includes registers (high efficiency)

Shared only over work-group (low concurrency)

Shared over all threads (high concurrency)

Cached



Appropriate use of these memory spaces can have significant performance implications for OpenCL applications.

The memory implementation is dependent on the devices used

Private memory

- Delivers the same performance as any other global memory region
- Normally, automatic variables declared in a kernel reside in registers, which provide very fast access.
- The compiler might choose to place automatic variables in private memory when:
  - There are too many register variables.
  - A structure would consume too much register space.
  - Registers are not addressable, so an array has to go into private memory.

Example:

Private

```
__kernel void multiply(__global float* arr)
{
   int i = get_global_id(0);
   __private int i = get_global_id(0); //OR
   __private float lookup[] = { 0, 1, 2 };
   arr[i] = arr[i] * arr[i];
}
```

`i` – register
`lookup` – memory

Local Memory

- Local to a work-group and can be used to allocate variables that are shared by all work-items in that work-group.
- It may be implemented as dedicated regions of memory on the OpenCL device.
- Or alternatively, the local memory region may be mapped onto sections of the global memory.

Example:

```
__kernel void test()
{
__local float x = 4.0; // DOES NOT WORK
__local float x; // WORKS BECAUSE IT IS NOT INITIALIZED
 x = 4.0;
}
```

Constant memory

- Constant memory is read-only from kernels
- It is hardware optimized for the case when all threads read the same location
- If threads read multiple locations, the accesses are serialized
- The constant cache is written only by the host and is persistent across kernel calls within the same application.

Example:

```
__constant float wtsA[] = { 0, 1, 2, . . . };

__kernel void kernel_func(__global float *f,
                          __constant int *abc) {

  __constant uint x = 5;
  __constant uint x; // ILLEGAL Not initalized
   ….
}
```

Global Memory

- Visible to all Work-items in all Work-groups
- The access can be defined in *CreateBuffer* as *Read* and/or *Write* (READ_ONLY, WRITE_ONLY or READ_WRITE)
- Can be accessed by the Host or the Device
- Multiple accesses are serialized

Example:

```
__kernel void kernel_func(__global float *f) {
  __global uint x = 5;
}
```

Image Memory (CUDA or AMD – texture memory)

- Image memory accelerates frequently performed operations such as mapping, or deforming a 2D "skin" onto a 3D polygonal model.
- CUDA supports a subset of the texturing hardware that the GPU uses for graphics to access texture memory
- Reading data from image memory instead of global memory can have several performance benefits

Example:

```
__kernel void negative_image2D(__read_only image2d_t image)
{
…
}
```

The extension to the C programming language

- Four types of extensions:
  - Function qualifiers
  - Variable type qualifiers
  - Built-in variables/functions

Functions in OpenCL

- A **kernel** function
  - starts with __kernel
  - Every kernel function must return void.
  - Some platforms won’t compile kernels without arguments.
  - Are callable by the host

- A **inline** function
  - Is called only by kernel functions and each call is replaced by its code.
  - Therefore, recursive algorithms are not available.
  - Can start with or without inline keyword
  - Can return native types

```
inline float helper_function(float4 input)
{
   return input.x + input.y + input.z + input.w;
}
```

Calling a inline function

Original code

```
inline float helper_function(float4 input)
{
   return input.x + input.y + input.z + input.w;
}

__kernel void kernel_function(__constant float4* arr, __global float* out)
{
  int id = get_global_id(0);
  out[id] = helper_function(arr[id]);
}
```

This code is (by default) compiled by inlining the function

```
__kernel void kernel_function( __constant float4* arr,
         __global float* out)
{
  int id = get_global_id(0);
  out[id] = arr[id].x + arr[id].y + arr[id].z + arr[id].w;
}
```

Variable type qualifiers

- **__global**
  - variable that resides on device memory (global memory by default).
  - Can be accessed by all work-items and by the host (**lifetime – application**).

- **__constant**
  - Resides on constant memory space.
  - Can be accessed by all work-items and by the host (**lifetime – application**).

- **__local**
  - Resides on local memory space.
  - Can only be accessed by the work-item of its work-group (**lifetime – work-group**).

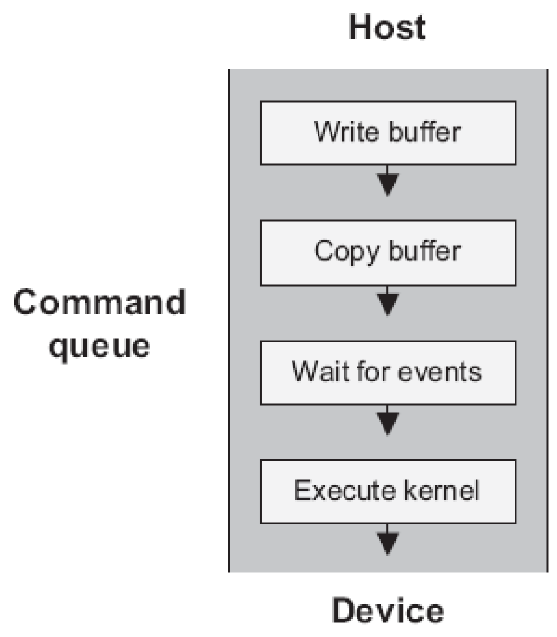
- **__private**
  - Resides on private memory and is the default type.
  - Can only be accessed by the work-item. (**lifetime – work-item**)

Scalar data types

- bool
- char, char{2|3|4|8|16},
- uchar, uchar{2|3|4|8|16},
- short, short{2|3|4|8|16},
- ushort, ushort{2|3|4|8|16},
- int, int{2|3|4|8|16},
- uint, uint{2|3|4|8|16},
- long, long{2|3|4|8|16},
- ulong, or ulong{2|3|4|8|16}
- float, or float{2|3|4|8|16}
- half, or half{2|3|4|8|16}
- double, or double{2|3|4|8|16}

Syncronization methods

- **Command Queue level**
  - Through Barrier command
  - Through markers that can wait or just notify the host (allows sync between queues)
  - Wait for event commands



- **Kernel level**
  - Barrier() command (kernels do not have events nor callback functions)
  - prevents all following instructions from executing until every preceding instruction has completed its execution.
  - **Only at work-group level**

Work-item Synchronization – kernel level

- Work-items can only be synchronized, if they are in the same work-group.
- **Barrier** function (kernel function) will stop execution until all work-items reach that position and all memory operations are complete.
- Only after a *barrier*(…), writes to local variables are guaranteed to be visible by other work-items

```
step1();
barrier(CLK_LOCAL_MEM_FENCE);
step2();
```

CLK_LOCAL_MEM_FENCE
CLK_GLOBAL_MEM_FENCE

- **Fence** functions are similar to barrier, except don't block and are for specific memory operations (waits for all previous memory operations to be complete):

```
void read_mem_fence (cl_mem_fence_flags flags)
void write_mem_fence (cl_mem_fence_flags flags)
void mem_fence (cl_mem_fence_flags flags)
```

Deprecated in OpenCL 2.0

Global memory concurrency

- What happens if two threads want to increase the value of a common variable?

  x = x+1;

- **Non-atomic** instructions
  - the number of serialized writes and the order in which they occur is undefined;
  - but at least one of the writes is guaranteed to succeed.

- ☑ **Atomic** instructions
  - ☑ reads and writes to the same location in global memory are all guaranteed and serialized
  - ☑ but with undefined order.

```
atomic_inc(local_result);
atomic_add(global_result, local_result[0]);
```

Check OpenCL extension: *cl_khr_global_int32_base_atomics*

Some Additional Notes

- How powerful is my GPU?

Nvidia Compute capability

- Only for NVIDIA
- Specifies Graphic Card performance version
- Defines board minimum specifications

Compute capability (version)	Cards
1.0	GeForce GT 420*, GeForce 8800 Ultra, GeForce 8800 GTX, GeForce GT 340*, GeForce GT 330*, GeForce GT 320*, …
1.1	GeForce G110M, GeForce 9300M GS, GeForce 9200M GS, GeForce 9100M G, GeForce 8400M GT, GeForce G105M, Quadro FX 4700 X2, Quadro FX 3700, …
1.2	GeForce GT 240, GeForce GT 220*, GeForce 210*, GeForce GTS 360M, GeForce GTS 350M, GeForce GT 335M, GeForce GT 330M, …
1.3	GeForce GTX 280, GeForce GTX 275, GeForce GTX 260, Quadro FX 5800, Quadro FX 4800, Quadro FX 4800 for Mac, Quadro FX 3800, Quadro CX, …
2.0	GeForce GTX 590, GeForce GTX 580, GeForce GTX 570, GeForce GTX 480, GeForce GTX 470, GeForce GTX 465, GeForce GTX 480M, Quadro 6000…
2.1	GeForce GTX 560 Ti, GeForce GTX 550 Ti, GeForce GTX 460, GeForce GTS 450, GeForce GTS 450*, GeForce GT 640 (GDDR3), GeForce GT 630, GeForce GT 620, …
3.0	GeForce GTX 770, GeForce GTX 760, GeForce GTX 690, GeForce GTX 680, GeForce GTX 670, GeForce GTX 660 Ti, GeForce GTX 660, GeForce GTX 650 Ti BOOST, …
3.5	GeForce GTX TITAN, GeForce GTX TITAN Black, GeForce GTX 780 Ti, GeForce GTX 780, GeForce GT 640 (GDDR5), GeForce GT 630 v2, Quadro K6000, Tesla K40, …
….	
7.5	NVIDIA TITAN RTX, Geforce RTX 2080 Ti, Geforce RTX 2080, Geforce RTX 2070

Nvidia Compute Capability 1.0

- The maximum number of threads per block is 512;
- The maximum sizes of the x-, y-, and z-dimension of a thread block are 512, 512, and 64, respectively;
- The maximum size of each dimension of a grid of thread blocks is 65535;
- The number of registers per multiprocessor is 8192;
- The amount of shared memory available per multiprocessor is 16 KB organized into 16 banks (see Section 5.1.2.5);
- The total amount of constant memory is 64 KB;
- The cache working set for constant memory is 8 KB per multiprocessor;
- The cache working set for texture memory varies between 6 and 8 KB per multiprocessor;
- The maximum number of active blocks per multiprocessor is 8;
- The maximum number of active warps per multiprocessor is 24;
- The maximum number of active threads per multiprocessor is 768;
- For a texture reference bound to a one-dimensional CUDA array, the maximum width is 213;
- For a texture reference bound to a two-dimensional CUDA array, the maximum width is 216 and the maximum height is 215;
- For a texture reference bound to a three-dimensional CUDA array, the maximum width is 211, the maximum height is 211, and the maximum depth is 211;
- For a texture reference bound to linear memory, the maximum width is 227;
- The limit on kernel size is 2 million PTX instructions;

Nvidia Compute Capability

- **Compute Capability 1.1**
  - Support for atomic functions operating on 32-bit words in global memory
- **Compute Capability 1.2**
  - Support for atomic functions operating in local memory and atomic functions operating on 64-bit words in global memory
  - The number of registers per multiprocessor is 16384;
  - The maximum number of active threads per multiprocessor is 1024.
- **Compute Capability 1.3**
  - Support for double-precision floating-point numbers.
- …
- **Compute Capability 12**

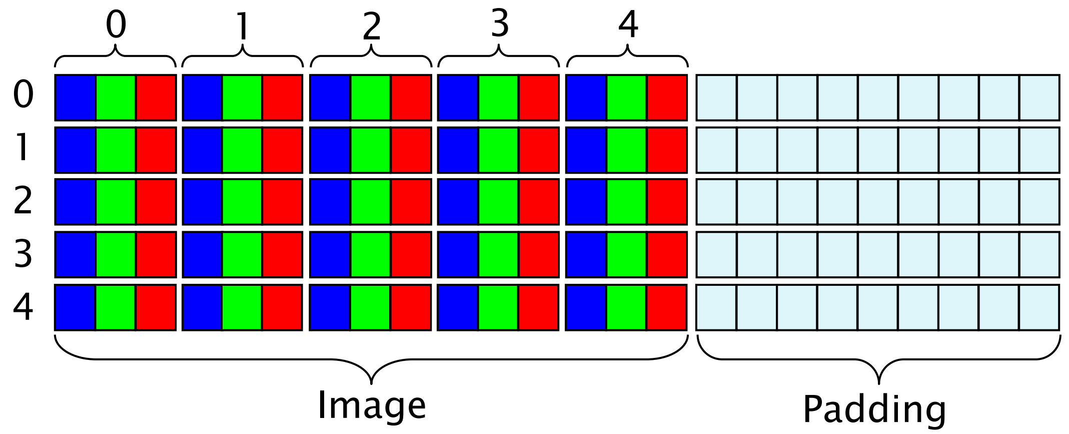
Non-breaking functions

Image Processing

- Parallelizing Techniques
  - Direct Memory Access
  - Image Texture Memory

What is pitch linear memory?

- Memory accesses have better performance if rows are aligned in memory
- There are two memory allocation methods: **Buffer** and **Image**
- It is important to distinguish between "linear memory" created with **Buffer** and "pitch linear" memory created with **Image**
- Both methods create linear memory, but **Image** pads the allocation to get best performance for the memory subsystem



Numpy ndarray – OpenCV Image support

C_CONTIGUOUS
WRITEABLE
ALIGNED

width in bytes

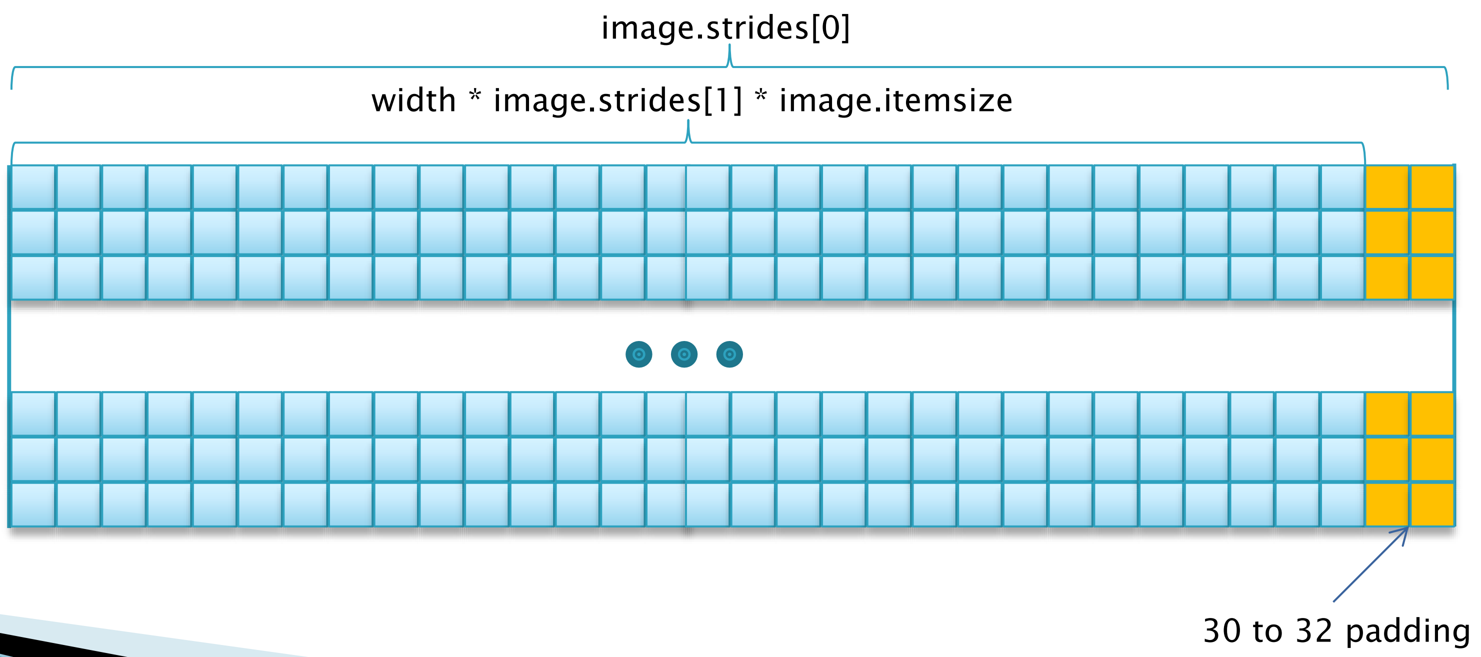
ndarray.flags	Information about the memory layout of the array.
ndarray.shape	Tuple of array dimensions.
ndarray.strides	Tuple of bytes to step in each dimension when traversing an array.
ndarray.ndim	Number of array dimensions.
ndarray.data	Python buffer object pointing to the start of the array's data.
ndarray.size	Number of elements in the array.
ndarray.itemsize	Length of one array element in bytes.
ndarray.nbytes	Total bytes consumed by the elements of the array.
ndarray.base	Base object if memory is from some other object.

The Image padding

- An important detail sometimes ignored!

padding = image.strides[0] – width * image.strides[1] * image.itemsize

- Why?



The Image padding – example

- image.shape = (33,25,3) # pixels (width, height, nChannels)
- image.strides = (108,3)

$$\text{padding} = \underset{108}{\text{image.strides[0]}} - \underset{33}{\text{width}} * \underset{3}{\text{image.strides[1]}} * \underset{1}{\text{image.itemsize}}$$

padding = 9

Image transfers (Host)

- **Buffer**– Create and copy from Host (H) to Device (D)

```
bufferMem = cl.Buffer(
        ctx,
        flags= cl.mem_flags.COPY_HOST_PTR |
                cl.mem_flags.READ_WRITE,
        size = arrayIn.nbytes,
        hostbuf=arrayIn)
```

Access type

- READ_WRITE
- WRITE_ONLY
- READ_ONLY
- COPY_HOST_PTR
- USE_HOST_PTR
- ALLOC_HOST_PTR
- HOST_NO_ACCESS
- HOST_READ_ONLY
- HOST_WRITE_ONLY
- KERNEL_READ_AND_WRITE

Constant Memory

Copy Data Host->Device

Device accesses Host memory

Image size

Pointer to Image

Image transfers (Host)

- **enqueue_copy** – Transfer H->D, D->H, D->D

- **_enqueue_write_buffer**– Copy from H to D
- **_enqueue_read_buffer**– Copy from D to H

(deprecated)

```
cl.enqueue_copy(
        commQ,
        dest= Buffer / Image / Host,
        src= Buffer / Image / Host ,
        origin ,
        region ,
        pitches,
        is_blocking = True
)
```

Only H->H is not available

Region to copy (Image)

Width in bytes (Image stride) (optional)

Wait until it finishes

A kernel with direct memory access

- The image buffer must be sent to device memory as any other memory transfer operation

Input buffer

Dimensions

Output buffer

```
__kernel void negative(__global uchar* image, int w, int h, int padding, __global uchar* imageOut)
{
    int x = get_global_id(0);
    int y = get_global_id(1);

    int idx = y * (w*3 + padding) + x*3 ;

    if ((x < w) && (y < h)) {    // check if x and y are valid image coordinates
      imageOut[idx] = 255 - image[idx];
      imageOut[idx+1] = 255 - image[idx+1];
      imageOut[idx+2] = 255 - image[idx+2];
    }
}
```

kernel using Image2D

- On GPUs, image data is stored in special global memory called *texture memory.*
- Image Create and Transfer functions are similar to buffer operations:
  - **Image** – Allocate memory in the device for an 2D or 3D image including pitch
  - **_enqueue_write_image** – Copy from Host to Device
  - **_enqueue_read_image** – Copy from Device to Host
  - **enqueue_copy** – similar as buffer
- *ImageFormat* and *Sampler* Objects are the basics for image manipulation

Create Image (*Host*)

```
imgFormat = cl.ImageFormat(
        cl.channel_order.BGRA,
        cl.channel_type.UNSIGNED_INT8)
```

Define image format

cl.get_supported_image_formats(ctx,
    cl.mem_flags.READ_WRITE,
    cl.mem_object_type.IMAGE2D)

```
bufferFilter = cl.Image(
        ctx,
        flags=cl.mem_flags.COPY_HOST_PTR | cl.mem_flags.READ_ONLY,
        format= imgFormat,
        shape=(width,height),
        pitches=(imageBGRA.strides[0],imageBGRA.strides[1]),
        hostbuf=imageBGRA.data)
```

Context

Access type *(ReadOnly or WriteOnly)*

Image format

Image size

Line pitch

Pointer to Image

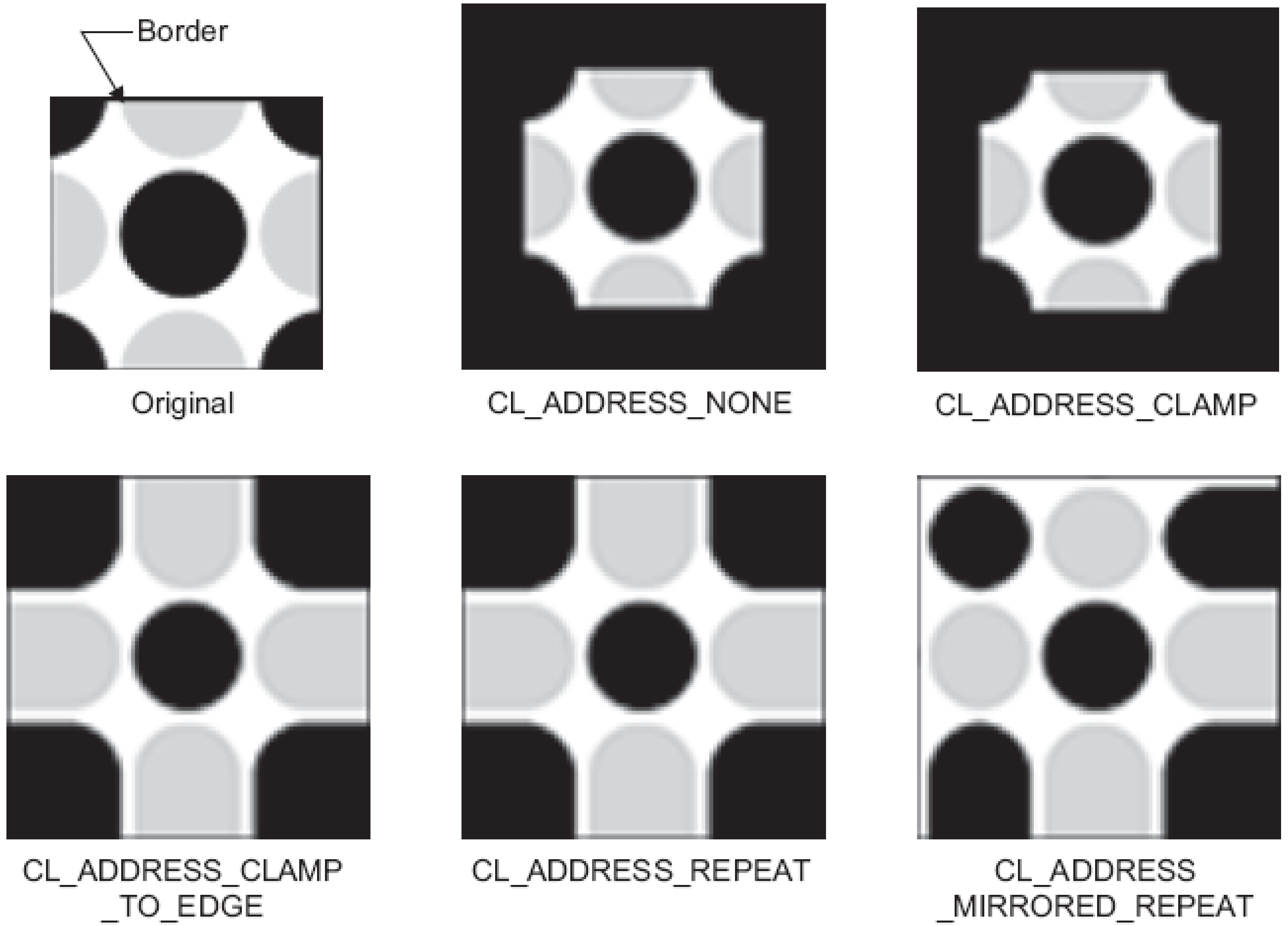
Sampler

- The sampler defines how pixels are accessed in image Objects.
- Can be defined from the Host or in the Device
  - *normalized_coords* – Identifies whether coordinates are normalized (given from 0.0 to 1.0) (TRUE / FALSE)
  - *addressing_mode* – Identifies how the kernel should process coordinates beyond the maximum value
  - *filter_mode* – Identifies how the kernel interpolates color values between pixels (LINEAR / Nearest Neighbor)

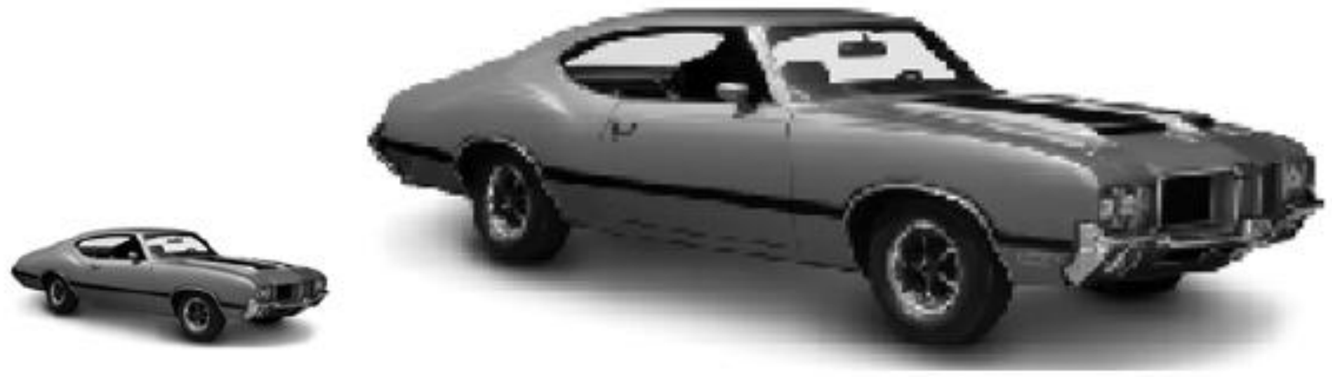
Definition in the device opencl file

```
__constant sampler_t sampler =
        CLK_NORMALIZED_COORDS_FALSE | //Natural coordinates
        CLK_ADDRESS_CLAMP_TO_EDGE |   //Clamp to zeros
        CLK_FILTER_NEAREST;           //Nearest Neighbor
```

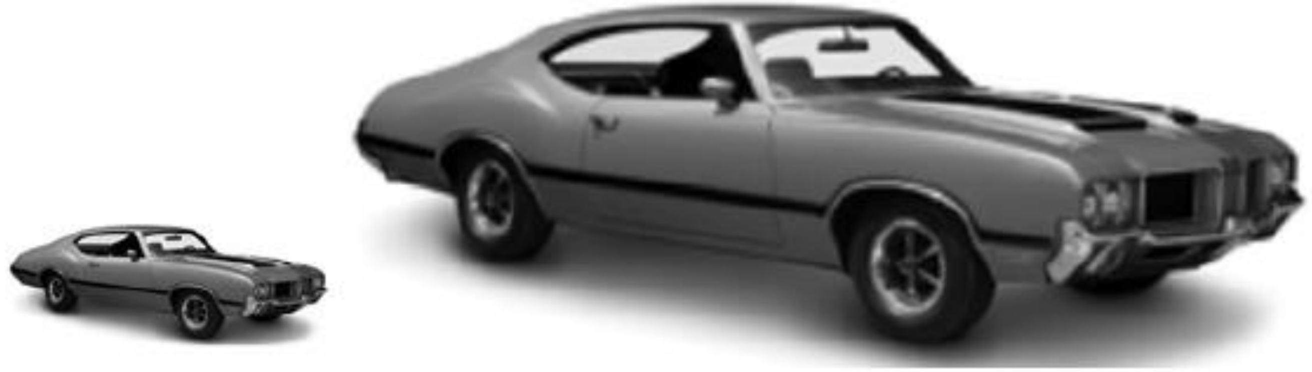
Sampler – address mode



Sampler – Filter mode



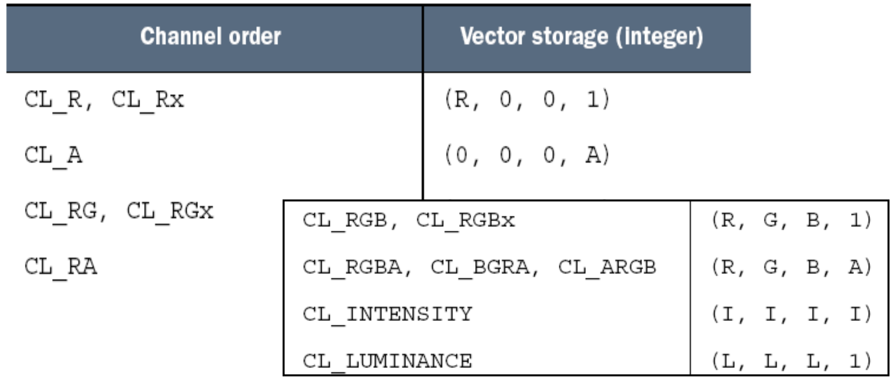
CLK_FILTER_NEAREST



CLK_FILTER_LINEAR

Pixel usage in Kernels

- For performance reasons pixels are accessed in 4xdata type format (int4…).
- Image formats with less than 4x data type per pixel the order of the returned values is defined by OpenCL



Channel order	Vector storage (Integer)
CL_R, CL_Rx	(R, 0, 0, 1)
CL_A	(0, 0, 0, A)
CL_RG, CL_RGx	
CL_RA	

CL_RGB, CL_RGBx	(R, G, B, 1)
CL_RGBA, CL_BGRA, CL_ARGB	(R, G, B, A)
CL_INTENSITY	(I, I, I, I)
CL_LUMINANCE	(L, L, L, 1)

uint4 **read_imageui** (image2d_t image,
sampler_t sampler,
int2/float2 coordinates)

void **write_imageui** (image2d_t img, int2/float2 coord, uint4 color)

```
Ex:  uint4 color = read_imageui(image, sampler, (int2)(3, 4));
```

- Additional versions for int4 and float4 are available (read_imagei, read_imagef, write_imagei, write_imagef)

```
Ex: write_imageui(image,(int2)(3, 4), (uint4)(255, 127, 86, 0));
```

Example – negative kernel

```
__constant sampler_t sampler =
        CLK_NORMALIZED_COORDS_FALSE | //Natural coordinates
        CLK_ADDRESS_CLAMP_TO_EDGE | //Clamp to zeros
        CLK_FILTER_NEAREST;


__kernel void negative_image2D(    __read_only image2d_t image,
                                   __write_only image2d_t imageOut,
                                   int w, int h)
{
    int iX = get_global_id(0);
    int iY = get_global_id(1);
    uint4 neg = (uint4)(255, 255, 255 , 0); //alpha must remain the same  (255)

    if ((iX >= 0)&&(iX < w) && (iY >= 0)&&(iY < h)) {
         uint4 pixelV = read_imageui( image, sampler, (int2)(iX,iY));
         write_imageui( imageOut, (int2)(iX,iY) , neg - pixelV );
    }
}
```