BITS Pilani
Pilani Campus

OpenCL

K Hari Babu
Department of Computer Science & Information Systems

OpenCL

- OpenCL (Open Computing Language) is a framework for writing programs that execute across heterogeneous platforms consisting of
  - central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators.
- OpenCL 1.0 was originally developed and released by Apple in 2009 as part of their MacOS system
- Apple submitted the initial OpenCL specification proposal to the Khronos Group in 2008. The Khronos Group in collaboration with AMD, IBM, Qualcomm, Intel, and NVidia published the OpenCL 1.0 specification in 2009

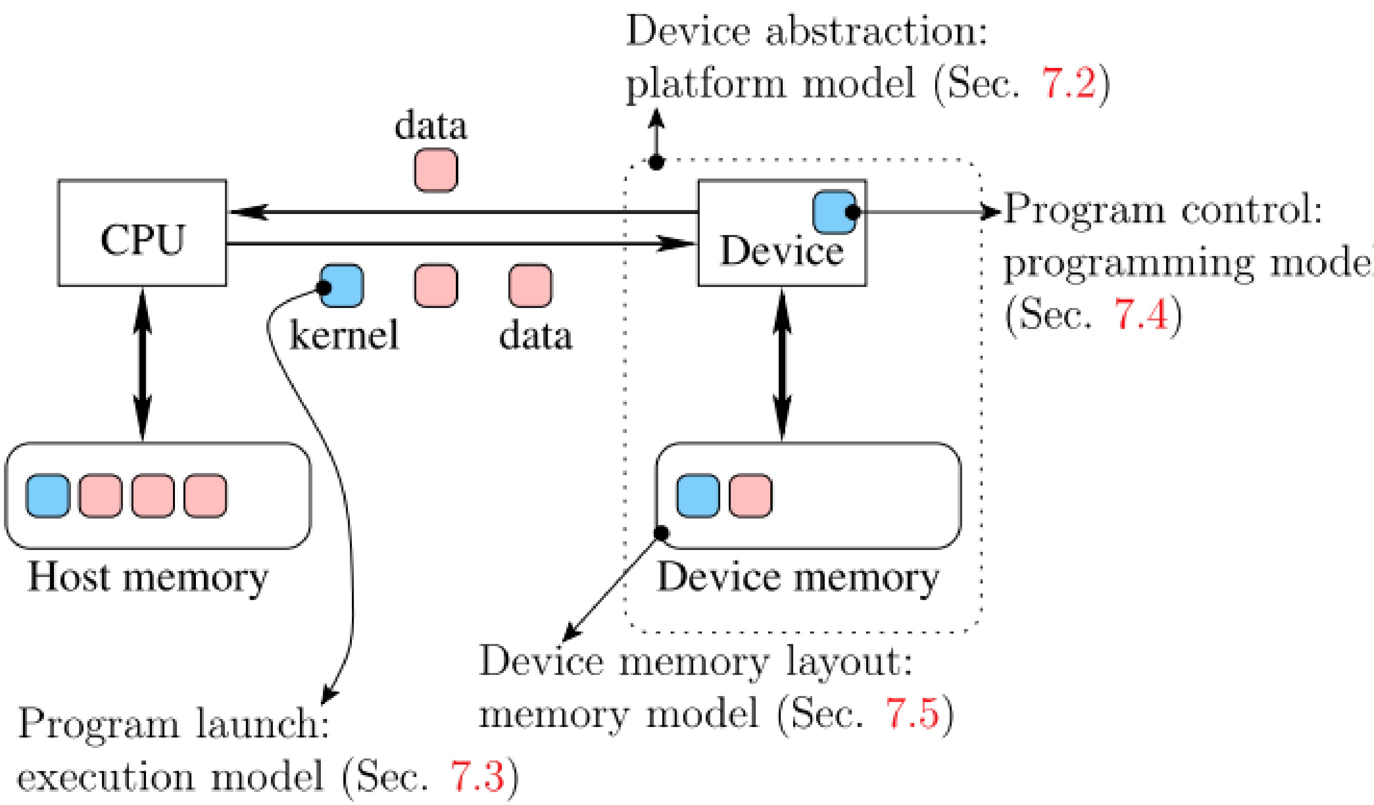
OpenCL

- Latest specification is 3.0 released in September 2020
- OpenCL 3.0 allows the use of C++17 features in kernels
- OpenCL is an open standard that allows a program to run on any supported device, as long as there is exists an Installable Client Driver (ICD)
  - An ICD is supposed to compile and execute an OpenCL program on-line (equivalent to using a PTX in CUDA). The intermediate code representation that is compiled by an ICD is called Standard Portable Intermediate Representation (SPIR).
- Off-line compilation is also supported.

OpenCL models

- OpenCL models describe different aspect of systems' operation, in order to facilitate the widest device support possible:
- The **Platform** Model : describes the composition of a system at the hardware level
- The **Execution** Model : describes how OpenCL programs are launched and executed
- The **Programming** Model : describes how an OpenCL program is mapped to hardware resources.
- The **Memory** Model : describes how memory is managed and data are transferred between memory spaces

OpenCL models

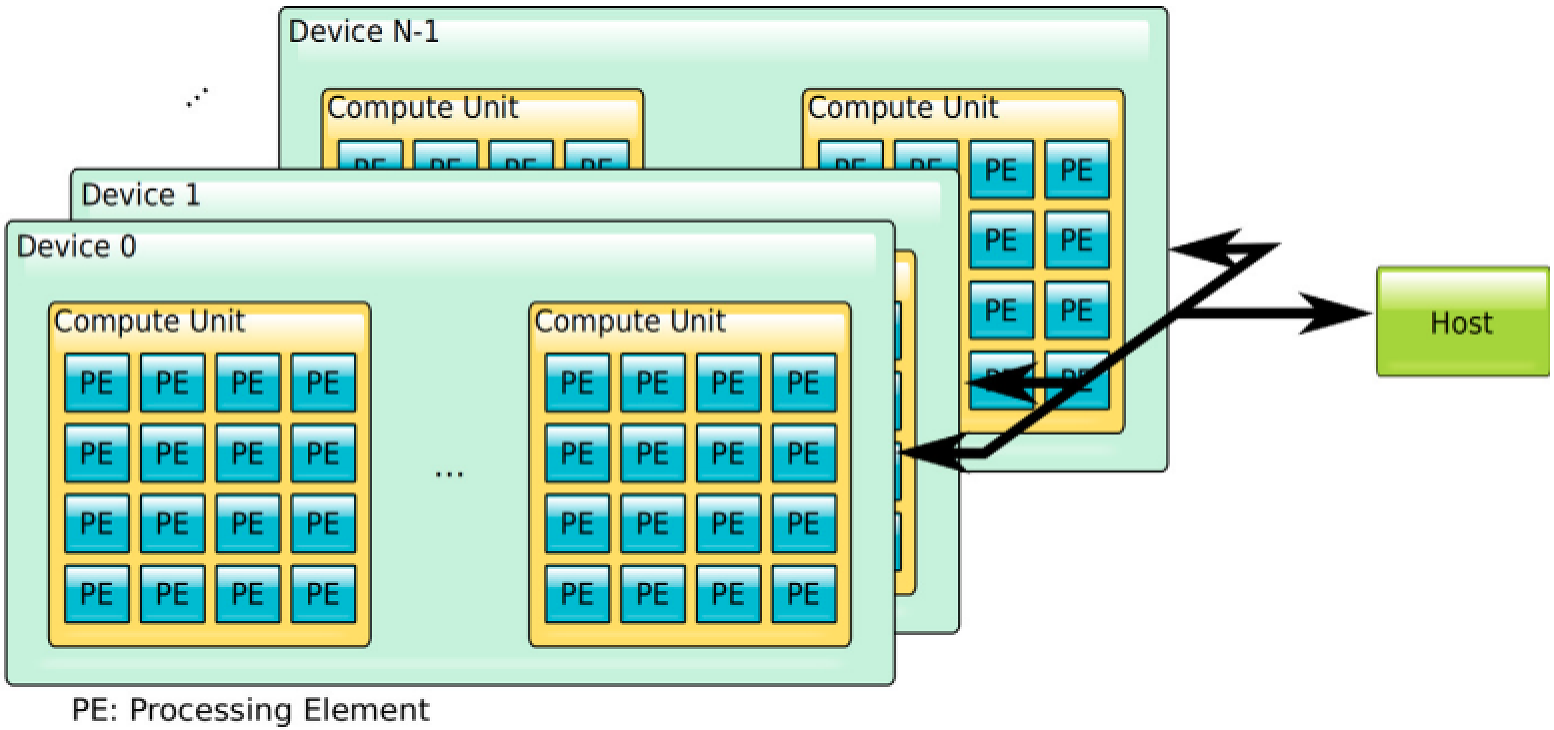


- Overview of the OpenCL models and how they relate to the operation of a heterogeneous system consisting of a CPU and an OpenCL device/accelerator with discrete memory

The platform model

- OpenCL assumes that the target execution platform is made of a host (typically a CPU) and a number of attached/hosted devices (typically one or more accelerators).
- Each device is composed of a number of compute units (CUs), which are in turn made of a number of processing elements (PEs).
- OpenCL's CUs correspond to CUDA's streaming multiprocessors (SMs)
- OpenCL's PEs correspond to CUDA's streaming processors (SPs)

The platform model



- An illustration of the OpenCL platform model.

Listing the devices

- Getting the available OpenCL devices can be accomplished by calling clGetPlatformIDs and clGetDeviceIDs

```
cl_int errNum;
cl_uint numPlatforms;
cl_platform_id firstPlatformId;
cl_device_id devID[MAXNUMDEV];
cl_uint numDev;

// Get a reference to an object representing a platform
errNum = clGetPlatformIDs(1, &firstPlatformId, &numPlatforms);
if (errNum != CL_SUCCESS || numPlatforms <= 0)
{
    cerr << "Failed to find any OpenCL platforms." << endl;
    return 1;
}
```

Listing the devices

```
// Get the device IDs matching the CL_DEVICE_TYPE parameter, up to the ↩
 MAXNUMDEV limit
errNum = clGetDeviceIDs( firstPlatformId,CL_DEVICE_TYPE_ALL, MAXNUMDEV, ↩
 devID, &numDev);
if (errNum != CL_SUCCESS || numDev <= 0)
{
    cerr << "Failed to find any OpenCL devices." << endl;
    return 2;
}

char devName[100];
size_t nameLen;
for(int i=0;i<numDev;i++)
{
  errNum = clGetDeviceInfo(devID[i], CL_DEVICE_NAME, 100, (void*)devName, ↩
 &nameLen);
  if(errNum == CL_SUCCESS)
      cout << "Device " << i << " is " << devName << endl;
}
```

- Compilation does not require anything special:
- *$ g++ deviceList.cpp -lOpenCL -o deviceList*

OpenCL data types

- OpenCL provides these in an attempt to ensure cross-platform data compatibility.

Data type	Bits	Description
cl_char	8	Signed 8-bit integer
cl_uchar	8	Unsigned 8-bit integer
cl_short	16	Signed 16-bit integer
cl_ushort	16	Unsigned 16-bit integer
cl_int	32	Signed four byte integer
cl_uint	32	Unsigned four byte integer
cl_long	64	Signed eight-byte integer
cl_ulong	64	Unsigned eight-byte integer
cl_half	16	Half-precision floating point number
cl_float	32	Single-precision floating point number
cl_double	64	Double-precision floating point number

The execution model

- OpenCL programs are typically broken down into two components:
  - A host program that is controlling execution of the device code and providing I/O facilities.
  - One or more device programs that are called **kernels**.
- A executes on a device in the form of work items (equivalent to CUDA threads). Each work item executes on a PE, and there can be multiple work items mapped to a PE, executing one at a time.

The execution model

- Work items are grouped into work groups (equivalent to CUDA blocks) and each work group is mapped to a CU.
- Kernel launches are placed on asynchronous command-queues.
- Since OpenCL 2.0, a kernel can enqueue requests in command queues of other devices. This creates a parent kernel to child kernel relationship between associated kernels

Command queue states

- Each enqueued OpenCL command goes in sequence through the following states:
  - Queued : Command is placed on a host-side queue
  - Submitted : Command has moved to the device.
  - Ready : Command is ready to execute. Commands may have dependencies/precedence, so this state indicates that all conditions for execution are met.
  - Running : One or more work groups associated with the command are running on the device.
  - Ended : All work groups have finished execution.
  - Complete : The command and all its child commands have finished execution.

Context

- A context is a special object that is used to manage the resources utilized by a resource during the execution of kernels
- No command can be issued to a device, before a context is created.

```
// Creates and returns an OpenCL context object
cl_context clCreateContext (
     const cl_context_properties *prop, // Null-terminated array of
                                        // desired property names
                                        // and their values(IN)
     cl_uint num_devices,               // Number of devices (IN)
     const cl_device_id *devices,       // Array of device IDs that
                                        // will be associated with
                                        // the context (IN)
     void (CL_CALLBACK*pfn_notify)      // Pointer to call-back
          (const char *errinfo,         // function, to be called
          const void *private_info,     // upon the detection of an
          size_t cb,                    // error during the creation
          void *user_data),             // of the context. Can be
                                        // set to NULL (IN)
     void *user_data,                   // Data to be passed to the
                                        // call-back function. Can
                                        // be set to NULL.(IN)
     cl_int *errcode);                  // Placeholder for error
                                        // code (IN/OUT)
```

Command queues

- Once a context is created, a command queue can be constructed with:

```
// Creates and returns an OpenCL command-queue for the specified device
cl_command_queue
   clCreateCommandQueueWithProperties (
          cl_context context,       // Context for created queue (IN)
          cl_device_id device,      // Device to be targeted by
                                    // queue (IN)
          const cl_command_queue_properties *prop, // Null-termina-
                                    // ted array of desired proper-
                                    // ties and their values (IN)
          cl_int *errcode);         // Returned error code (IN/OUT)
```

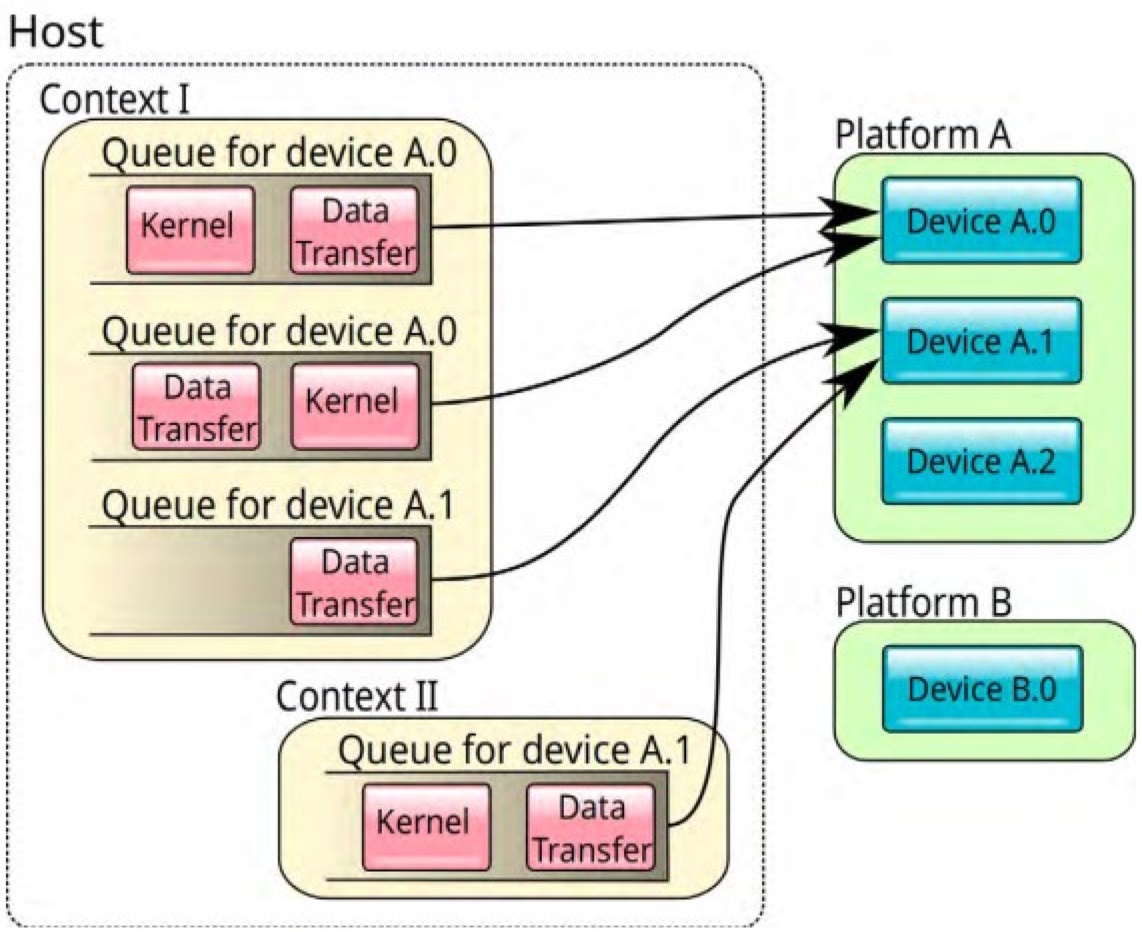
- The properties dictate how the queue will treat whatever is placed in it. For example, we can have queues that are out-of-order.
- The prop array should be NULL terminated.

Command queue example

```
cl_command_queue q;
cl_queue_properties qprop[]={
      CL_QUEUE_PROPERTIES,
      CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE,
      0};

q = clCreateCommandQueueWithProperties(
          cont,      // context
          devID[0], // and device ID are required
          qprop,     // this can be NULL if defaults are OK
          &errNum);
if (errNum != CL_SUCCESS)
  {
    cerr << "Failed to create a command queue" << endl;
    return 4;
  }
```

- If qprop was set to empty, i.e. {0}, the queue would default to a regular FIFO operation

Platforms, devices, contexts and queues



The programming model

- A kernel is defined as a normal C function, with the added decoration of the __kernel/kernel keyword. A kernel function cannot return anything directly to the host which invoked it, so it must be declared as returning void.
- Kernels are typically compiled on-line i.e., after the host program begins execution. For this reason, they have to be declared as strings. As such they can either be statically defined character arrays, or they can reside in separate text files that are loaded prior to their compilation.
- Once a kernel function is compiled, the device executable is generated, which in OpenCL jargon is called the program object (represented by the cl_program data type). The program object combined with the associated arguments, forms a kernel object that can be placed in the appropriate command-queue.

Program object creation

- This is a two step process:
  - (1) Create a program object using the kernel source(s)

```
// Creates a program object and returns an index/reference to it
cl_program clCreateProgramWithSource(
        cl_context context,    // Valid context  (IN)
        cl_uint count,         // Number of supplied strings in
                               // following argument (IN)
        const char **strings   // Points to array of source code
                               // strings. It is a pointer to an
                               // array of pointers (IN)
        const size_t *lengths,// Pointer to array of program string
                               // lengths. Can be NULL if the
                               // strings are NULL-terminated  (IN)
        cl_int *errcode_ret);  // Place holder for error code
                               // (IN/OUT)
```

Program object creation

- This is a two step process:
  - (2) Compile the program for one or more target OpenCL devices

```
// Compiles (and links) the provided sources for the target
// devices. Returns error code or CL_SUCCESS
cl_int clBuildProgram(
      cl_program program,   // Program object reference (IN)
      cl_uint num_devices,  // Number of targeted devices (IN)
      const cl_device_id *devlist, // Pointer to array of device
                            // IDs (IN)
      const char *options,  // String of build options for OpenCL
                            // compiler (IN)
      void (CL_CALLBACK *pfn_notify) // Pointer to call-back
           (cl_program program,      // function to be called
            void *user_data),        // when compilation is
                                     // complete. Can be NULL(IN)
      void *user_data);     // Data to be passed to call-back
                            // function. Can be NULL (IN)
```

Kernel creation

- A program may contain several kernel functions. The one to be executed needs to be identified y setting up a cl_kernel object after compilation:

```
// Creates and returns a kernel object from a compiled program.
cl_kernel clCreateKernel(
      cl_program program,        // Reference to compiled program (IN)
      const char *kernel_name,   // String identifying the kernel
                                 // function (IN)
      cl_int *errcode_ret);      // Placeholder for error code
                                 // (IN/OUT)
```

Kernel launch

- To execute a kernel, one must specify the 1D, 2D, or 3D index space over which work items will be generated. This index space is called an N-Dimensional Range or NDRange, with N being 1, 2, or 3. An NDRange is also used to specify a work group size, which in turn is used to partition the work items into work groups
- The work group size has to be less or equal to the size of the overall/global NDRange
- In contrast to CUDA where the grid and block dimension are different and independent, in OpenCL there is just one index space that the work groups partition
- The index space coordinates can start from any number. If unspecified they default to zero
- In OpenCL 2.0 the work group size does not have to evenly split the index space. So we can have work groups with a smaller number of work items

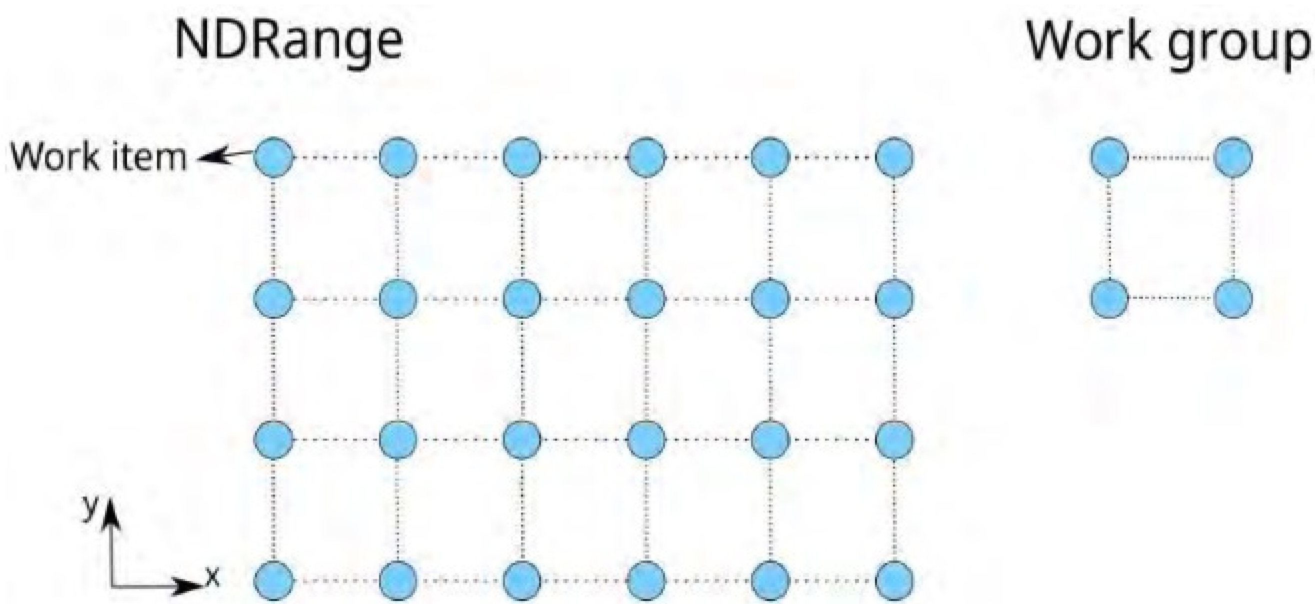
Kernel launch

- Asynchronous call for placing a kernel in a command queue:

```
cl_int clEnqueueNDRangeKernel (
      cl_command_queue queue,      // Queue for placement (IN)
      cl_kernel kernel,            // Kernel to execute (IN)
      cl_uint workDim,             // Size of the following three
                                   // arrays(IN)
      const size_t *globalOffset,  // Points to a 1D array for
                                   // global offsets, i.e.,
                                   // starting values for NDRange
                                   // IDs. Offsets default to zeros
                                   // if this is NULL.(IN)
      const size_t *globalSize,    // Vector for global work item
                                   // size (IN)
      const size_t *localSize,     // Vector for work group size(IN)
      cl_uint numEventsInWaitList, // Size of following vector (IN)
      const cl_event *eventWaitLst,// Vector of event objects that
                                   // need to be "triggered" before
                                   // the kernel can execute, i.e.,
                                   // a list of prerequisites. It
                                   // can be NULL (IN)
      cl_event *event);            // Address of event object that
```

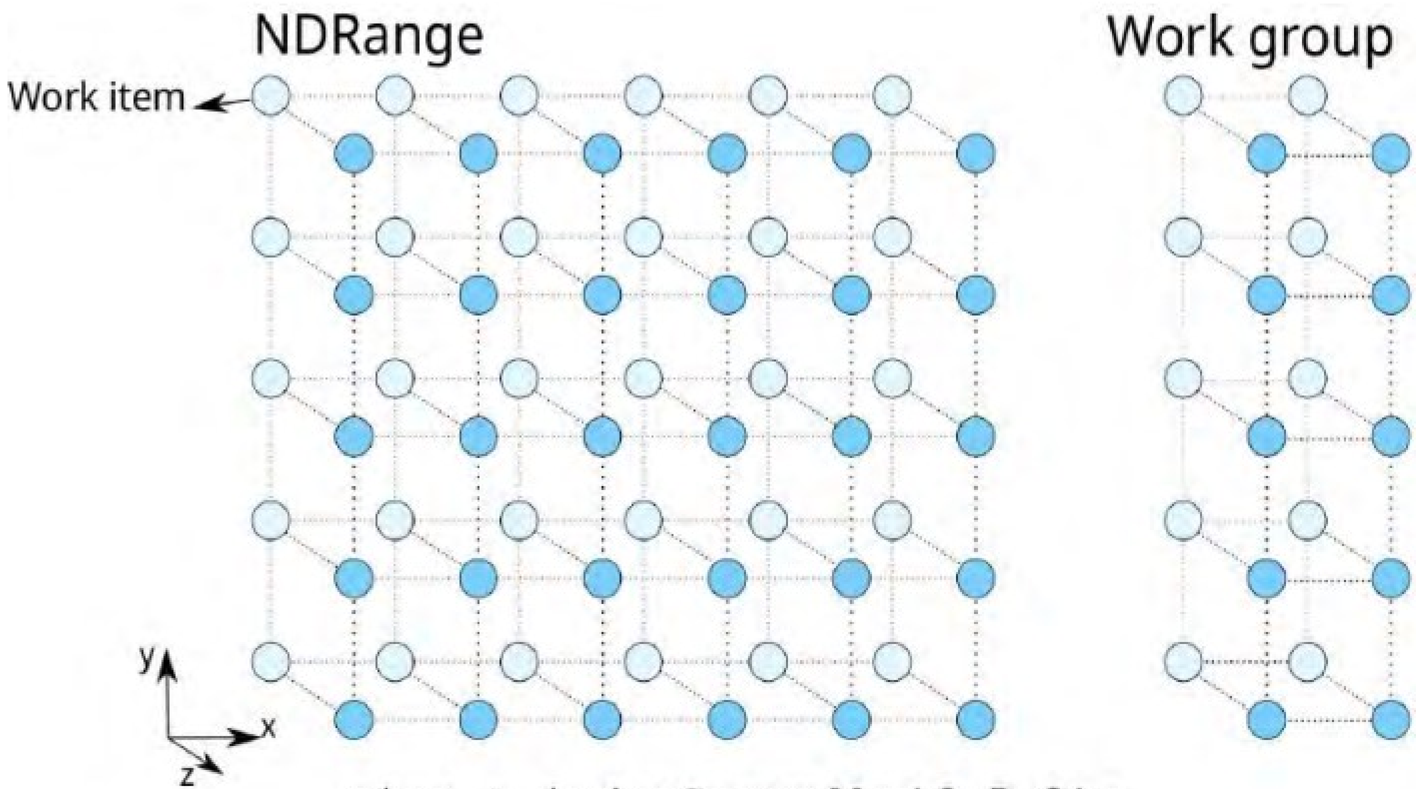
These can have up to 3 elements

NDRange example (1)



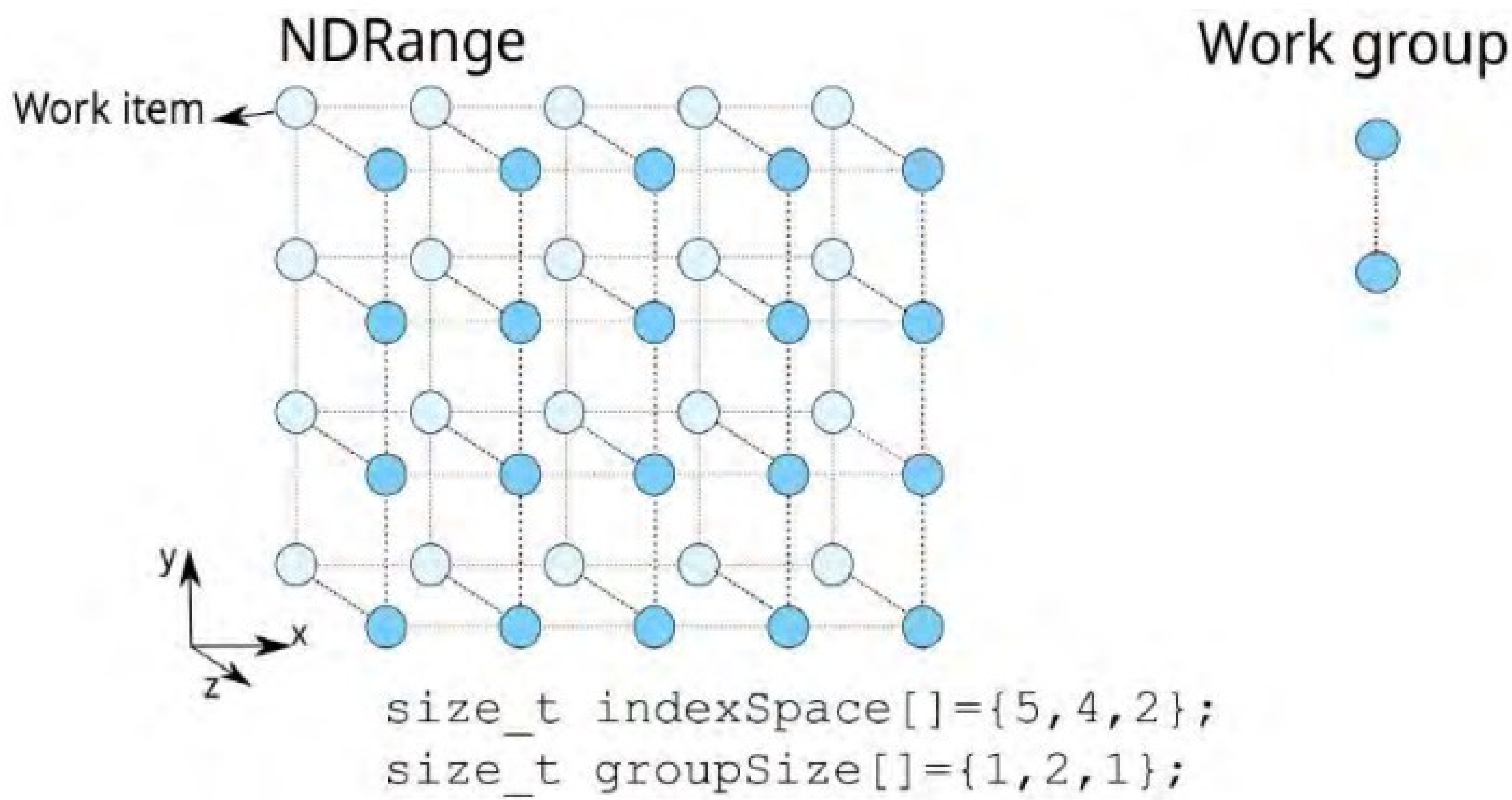

```
size_t indexSpace[]={6,4};
size_t groupSize[]={2,2};
```

NDRange example




```
size_t indexSpace[]={6,5,2};
size_t groupSize[]={2,5,2};
```

NDRange example



NDRange position information

- OpenCL provides a large collection of functions for this purpose:

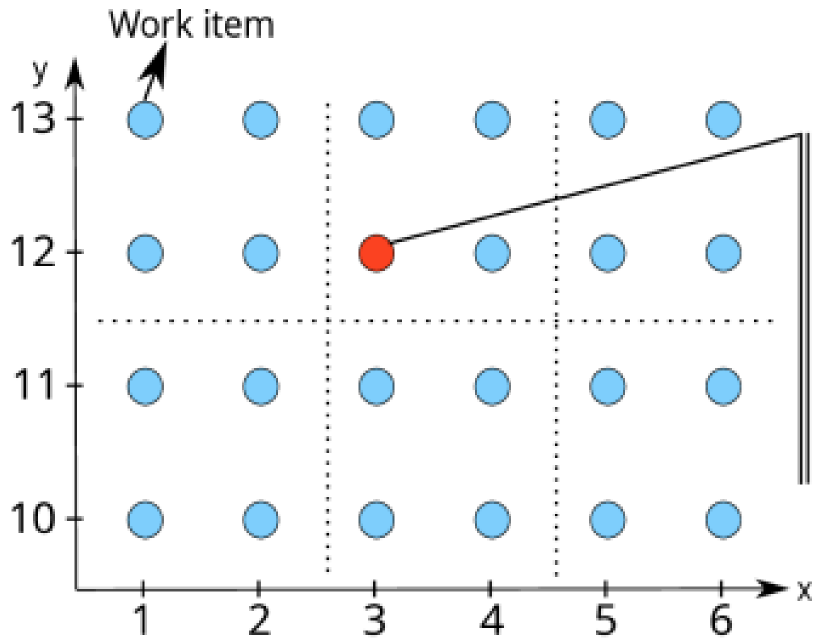
```
uint get_work_dim();                 // Returns the number of
                                     // dimensions in use
size_t get_global_size(uint dim);    // Returns the number of global
                                     // work-items in the given
                                     // dimension
size_t get_global_id(uint dim);      // Returns the global work item
                                     // position in the given
                                     // dimension
size_t get_local_size(uint dim);     // Returns the size of work group
                                     // in the given dimension, if
                                     // work-group size is uniform
size_t get_local_id(uint dim);       // Returns the local work item ID
                                     // in the given dimension
size_t get_num_groups(uint dim);     // Returns the number of work
                                     // groups in the given dimension
size_t get_group_id(uint dim);       // Returns the work group ID in
                                     // the given dimension
size_t get_global_offset(uint dim);// Returns the global offset for
                                     // the specified dimension
```

NDRange position information

- OpenCL 2.0 adds some convenience functions:

```
size_t get_global_linear_id();      // Returns the work item's global
                                    // ID, mapped to 1D
size_t get_local_linear_id();       // Returns the work item's local
                                    // ID, mapped to 1D
size_t get_enqueued_local_size(uint dim); // Returns the number
                                    // of local work items, even if
                                    // work group size is non-uniform
```

NDRange functions example




```
get_global_size(0)   -> 6
get_global_id(1)     -> 12
get_local_size(1)    -> 2
get_local_id(1)      -> 0
get_num_groups(0)    -> 3
get_group_id(0)      -> 1
get_global_offset(1)-> 10
```

```
size_t indexSpace  []={6,4};
size_t groupSize   []={2,2};
size_t globalOffset[]={1,10};
```

Compilation errors

- In order to capture any errors that happen during the online compilation, the clGetProgramBuildInfo function can be used
- A typical scenario is to be called two times, once for getting the error message length and once for getting the actual message:

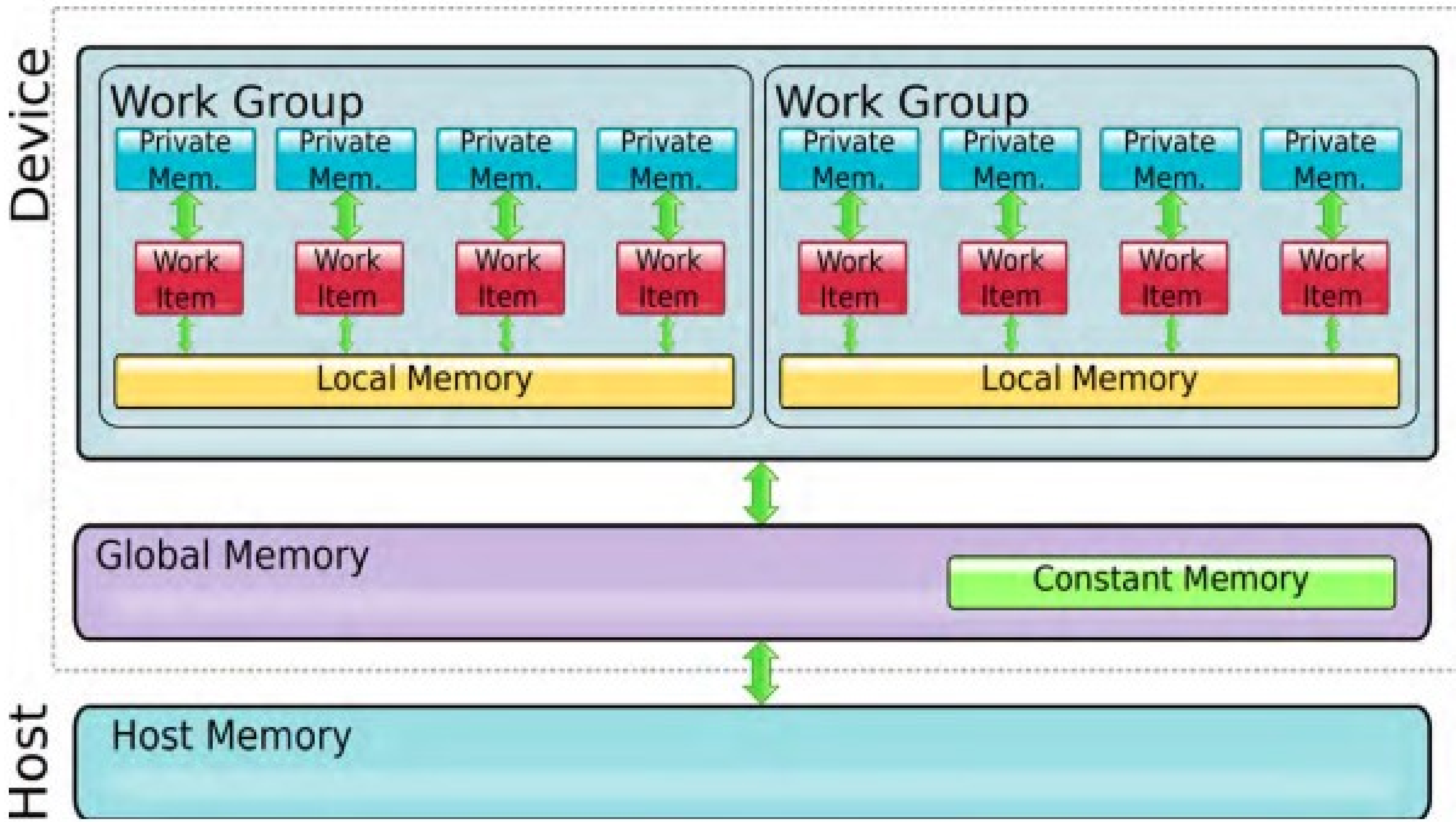
```
size_t logSize;
// Retrieve message size in "logSize"
clGetProgramBuildInfo(program, deviceID, CL_PROGRAM_BUILD_LOG, 0, NULL, &↩
     logSize);

// Allocate the necessary buffer space
char *logMsg = (char *)malloc(logSize);

// Retrieve the actual error message
clGetProgramBuildInfo(program, deviceID, CL_PROGRAM_BUILD_LOG, logSize, logMsg↩
     , NULL);
printf("Error : %s\n", logMsg);
```

- An error would be flagged if something other than CL_SUCCESS was returned by clBuildProgram

The memory model

- Device memory is divided into the following memory regions:
  - Global memory : memory which is accessible from all work items. It is typically used for holding input data to kernels and output data from the kernel execution. It is the only device memory that can be accessed by the host using appropriate OpenCL calls
  - Constant memory : a part of global memory (hence universal access by all work items) that is meant to be read-only. For example, it can be used to hold tables of immutable values. The benefit of distinguishing constant memory as a separate entity from the global memory is that devices may have special support (e.g., dedicated caches) for this type of data. NVidia GPUs do have such dedicated caches
  - Local memory : memory which is local to a work group. Local memory is used to share data by the work items of a group, and as such it would be mapped to on-chip memory, making a low-latency, high-bandwidth option. On an NVidia GPU this would be equivalent to SM shared memory, as work groups map to SMs
  - Private memory : memory that is exclusive to a work item. It is used to hold the automatic variables of a kernel and it is typically implemented by registers.

The memory model illustrated



Memory region access and lifetime

Region	Host access	Kernel access	Definition	Lifetime
Global memory	R/W	R/W	`global` or `__global`	Program
Constant memory	R/W	R	`constant` or `__constant`	Program
Local memory	-	R/W	`local` or `__local`	Work group execution
Private memory	-	R/W exclusive to work item	Local variables and non-pointer kernel arguments	Work item execution

Device memory management

- OpenCL manages device data in the form of three types of objects, which can reside in global or constant memory:
  - Buffers : a buffer is the equivalent of a 1D array in C
  - Images : designated for storing image data. The exact storage arrangement depends on the device specifics. Image objects are supposed to provide accelerated access to image data by utilizing the specialized hardware provided by a device
  - Pipes : placeholders for arbitrary data chunks called packets. Pipes behave as FIFO queues and they are designed for supporting producer-consumer software designs.

OpenCL buffers

- An OpenCL buffer can be created with:

```
cl_mem clCreateBuffer (
       cl_context context,// Context associated with memory obj.(IN)
       cl_mem_flags flags,// Flags used for the type of
                          // allocation(IN)
       size_t size,       // Size of required memory in bytes (IN)
       void *hostPtr,     // Pointer to host memory to be copied
                          // over to the device for initialization.
                          // Its size has to be at least equal to
                          // 'size' if it is utilized. Could be set
                          // to NULL. (IN)
       cl_int *errcode);  // Address for storing the returned error
                          // code (IN/OUT)
```

- flags control the access rights and the initialization procedure

- For example, one of the available settings for flags is CL_MEM_USE_HOST_PTR which allows a simultaneous device memory initialization:

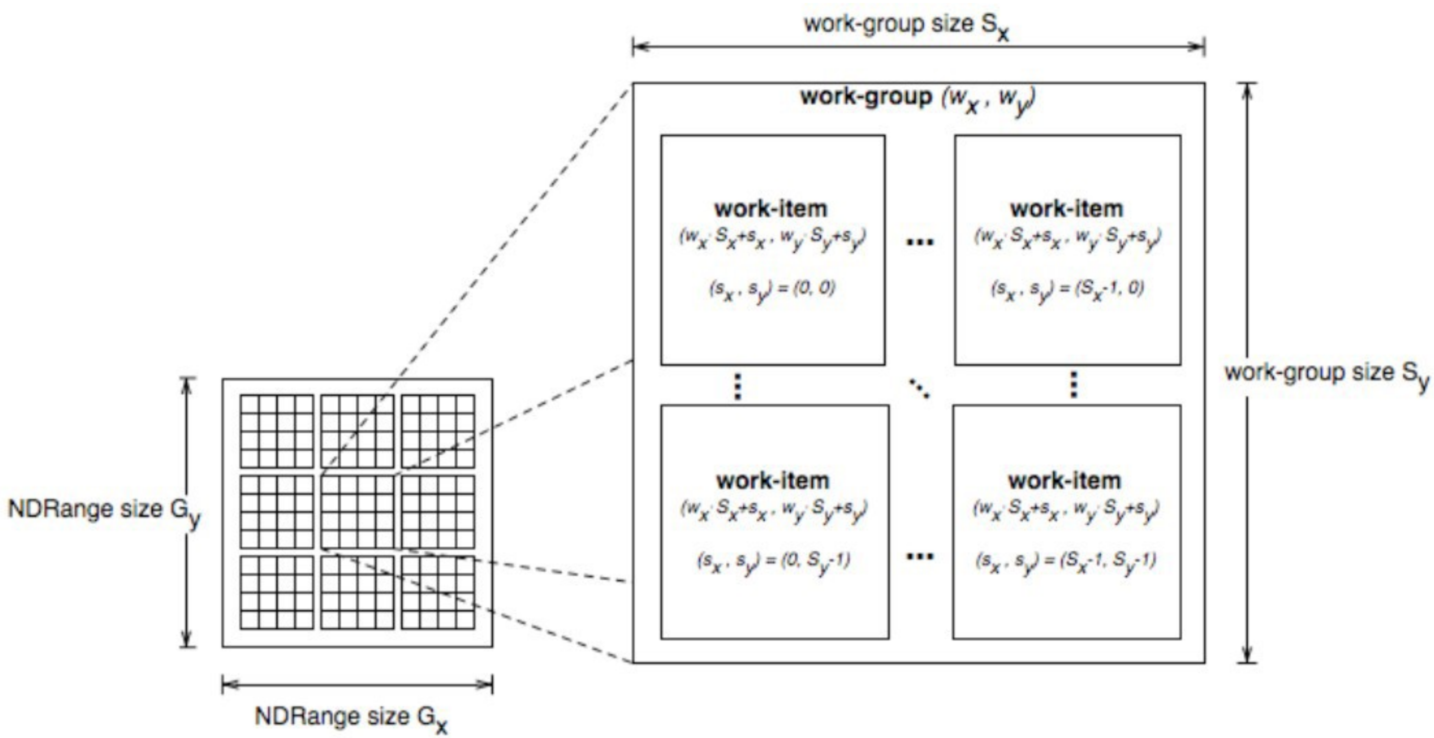
```
float h_data[N]; // data residing in host memory

cl_mem d_data;  // reference to an OpenCL buffer
size_t dataSize = N*sizeof(float);  // size of memory to be allocated in bytes

// allocation and initialization
d_data = clCreateBuffer (context, CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR, ↩
     dataSize, h_data, &errNum);
```

- If this flag is missing, an explicit memory transfer would be required:

```
d_data = clCreateBuffer (context, CL_MEM_READ_WRITE, dataSize, NULL, &errNum);

// data transfer enqueued
errNum = clEnqueueWriteBuffer(queue, d_data, CL_TRUE, 0, dataSize, h_data, 0, ↩
     NULL, NULL);
```

OpenCL to CUDA Data Parallelism Model Mapping

OpenCL Parallelism Concept	CUDA Equivalent
kernel	kernel
host program	host program
NDRange (index space)	grid
work item	thread
work group	block

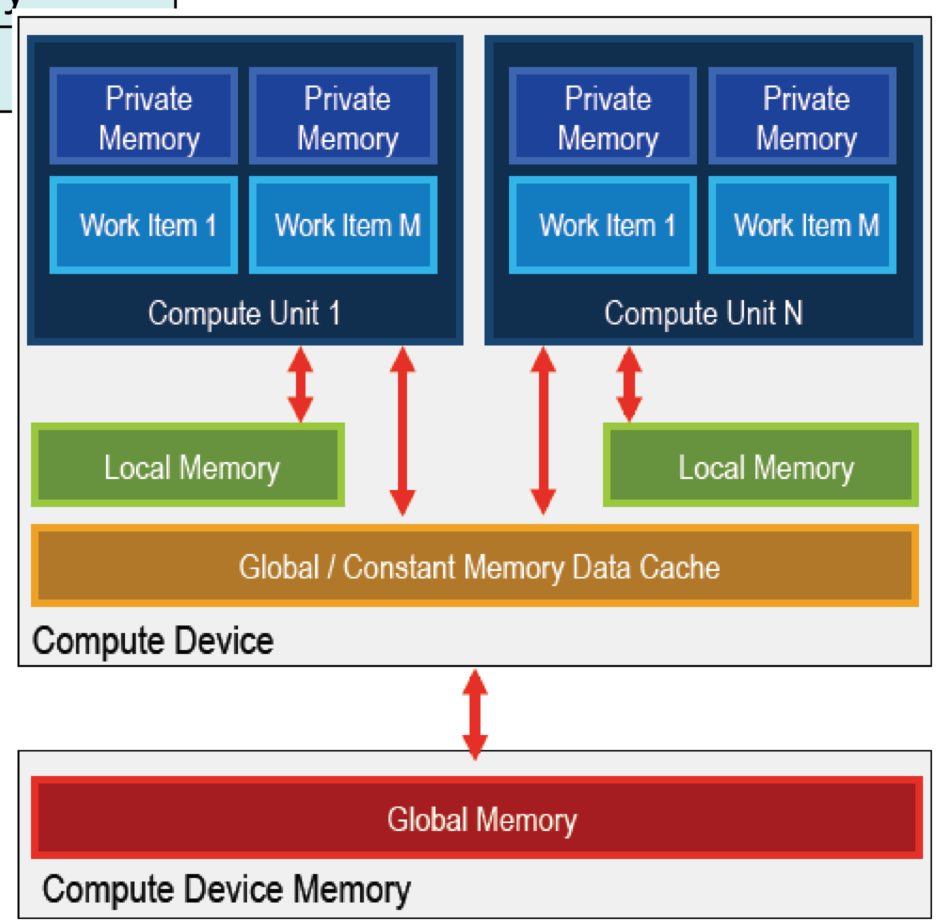


Mapping of OpenCL Dimensions and Indices to CUDA

OpenCL API Call	Explanation	CUDA Equivalent
get_global_id(0);	global index of the work item in the x dimension	blockIdx.x×blockDim.x+threadIdx.x
get_local_id(0)	local index of the work item within the work group in the x dimension	blockIdx.x
get_global_size(0);	size of NDRange in the x dimension	gridDim.x ×blockDim.x
get_local_size(0);	Size of each work group in the x dimension	blockDim.x

Mapping OpenCL Memory Types to CUDA

OpenCL Memory Types	CUDA Equivalent
global memory	global memory
constant memory	constant memory
local memory	shared memory
private memory	Local memory



References

- Chapter 7, GPU and accelerator programming: OpenCL, Multicore and GPU Programming An Integrated Approach, 2ns edition Gerassimos Barlas, Morgan Kaufmann

Thank You