

Open Computing Language (OpenCL) Architecture and Programming

6 Hours

Topics covered:

1. Introduction to OpenCL
2. The OpenCL Specification
3. Kernels and OpenCL Execution Model
4. Steps in Executing an OpenCL program
5. Platform and Devices
6. The Execution Environment
7. Memory Model
8. Writing Kernels
9. Vector-vector Addition Example

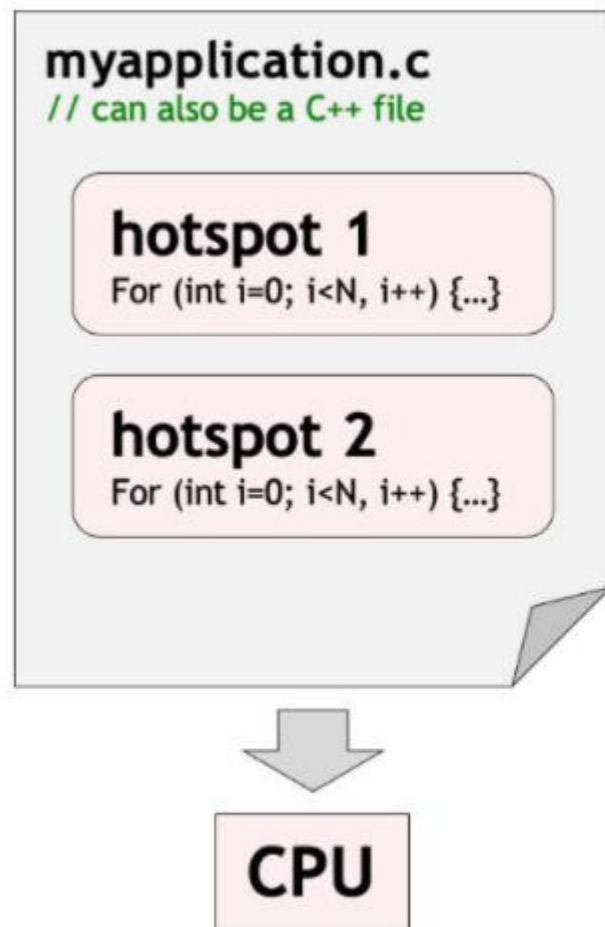
TextBook:

OpenCL - Benedict R. Gaster, Lee Howes, David R, Perhaad Mistry,
Dana Schaa, “Heterogeneous Computing with OpenCL”, Elsevier
Inc., 1st Edition, 2012.

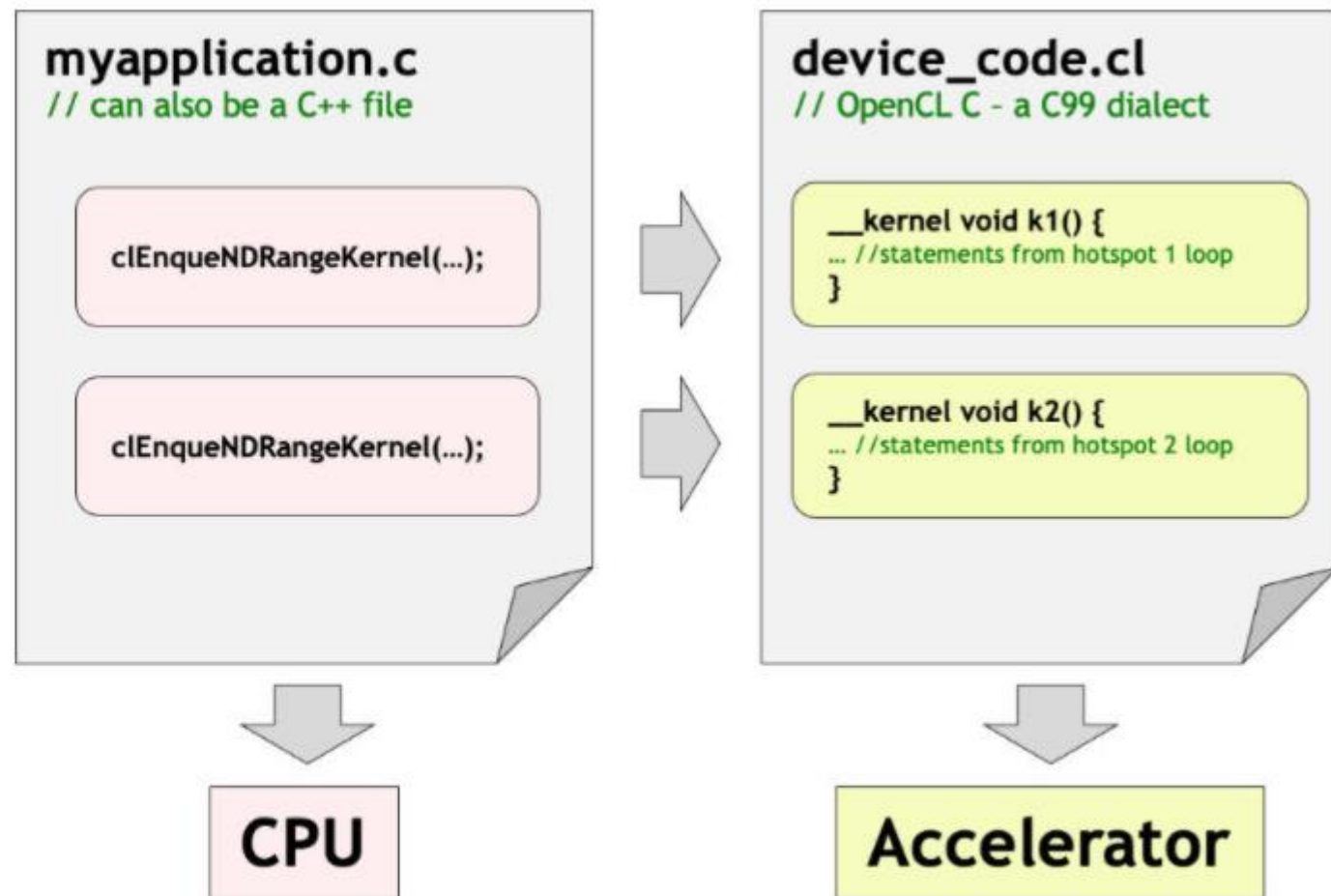
Introduction to OpenCL

- Open Computing Language is a **heterogeneous** programming framework that is managed by [Khronos group](#).
- OpenCL is a **framework** for developing applications that execute across a range of device type made by **different vendors**.
- OpenCL creates **portable, vendor- and device-independent** programs that are capable of being accelerated on many different hardware platforms.
- The OpenCL API is a **C** with a **C++ Wrapper** API that is defined in terms of the C API.
- The code that executes on an OpenCL device, which in general is **not the same** device as the host CPU, is written in the OpenCL C language.

C/C++ Programming



OpenCL Programming



Traditional vs OpenCL programming paradigm

The OpenCL Specification

The OpenCL specification is defined in four parts, called models:

1. **Platform model:** Specifies that there is one processor coordinating execution (the *host*) and one or more processors capable of executing OpenCL C code (the *devices*). **It defines an abstract hardware model** that is used by programmers when writing OpenCL C functions (called *kernels*) that execute on the devices.
2. **Execution model:** Defines how the OpenCL environment is configured on the host and how kernels are executed on the device. This includes :
 - ❑ setting up an OpenCL **context** on the host
 - ❑ providing mechanisms for **host–device interaction**
 - ❑ defining a **concurrency model** used for kernel execution on devices.
1. **Memory model:** Defines the **abstract memory hierarchy** that kernels use, regardless of the actual underlying memory architecture. The memory model closely resembles current **GPU memory hierarchies**.
2. **Programming model:** Defines how the concurrency model is **mapped** to physical hardware.

Kernels and the OpenCL execution model

- Kernels are the parts of an OpenCL program that actually execute on a device.
- An OpenCL kernel is syntactically similar to a standard C function; the key difference is a set of additional keywords and the execution model that OpenCL kernels implement.
- When developing concurrent programs for a CPU, the programmer considers the physical resources available (e.g., CPU cores) and the overhead of creating and switching between threads.
- With OpenCL, the goal is often to represent parallelism programmatically at the finest granularity possible.

Example:- Element-wise vector addition

1. Serial C implementation:

// Perform an element-wise addition of A and B and store in C
// There are N elements per array

```
void vecadd (int *C, int* A, int *B, int N)
{
    for(int i = 0; i < N; i++)
    {
        C[i] = A[i] + B[i];
    }
}
```


2. Thread C implementation:

A **coarse-grained, multithreaded** version code for a **multi-core device** require dividing the work (i.e., loop iterations) between the threads.

```
// Perform element-wise addition of A & B and store in C  
// There are N elements per array and  $N_p$  CPU cores
```

```
void vecadd (int *C, int* A, int *B, int N, int  $N_p$ , int tid)  
{  
    int ept = N/ $N_p$ ;    // elements per thread  
    for (int i = tid*ept; i < (tid+1)*ept; i++)  
    {  
        C[i] = A[i] + B[i];  
    }  
}
```

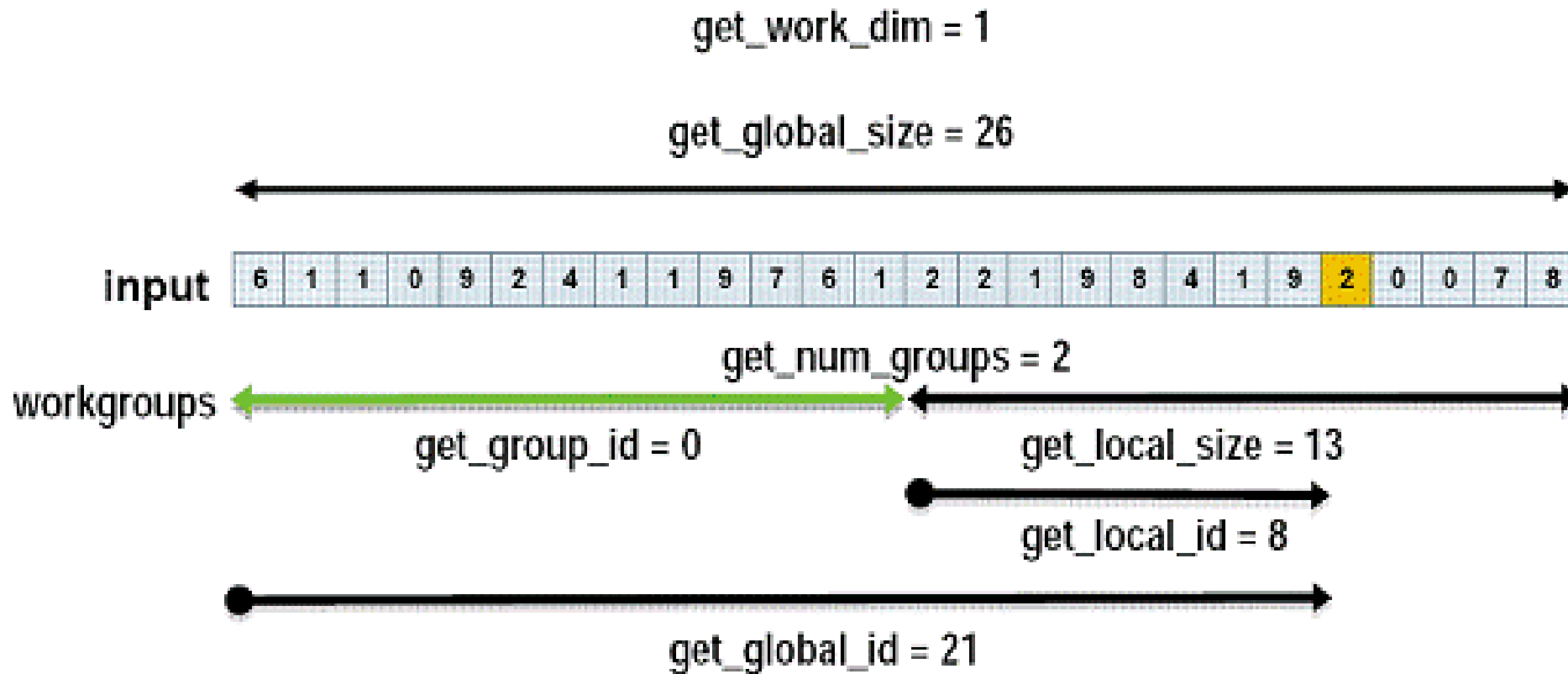
* Because there may be a large number of loop iterations and the work per iteration is small, we chunk the loop iterations into a larger granularity (a technique called *strip mining*)

3. OpenCL implementation (kernel):

- The unit of concurrent execution in OpenCL is a **work-item** (WI).
- Each WI executes the **kernel** function body.
- We map a single iteration of the loop to a WI.
- We tell the OpenCL runtime to **generate as many WIs as elements in the input and output arrays** and allow the OpenCL to map those WIs to the underlying hardware, (CPU / GPU cores).
- When an OpenCL device begins executing a kernel, it allow the programmer to identify the **position of the current WI** using a call to OpenCL API **get_global_id(0)**.

```
// N work-items will be created to execute this kernel
__kernel void vecadd (__global int *C, __global int* A, __global int *B)
{
    int tid = get_global_id(0); // OpenCL intrinsic function
    C[tid] = A[tid] + B[tid];
}
```

global-id Vs local-id



3. OpenCL implementation (kernel):

- When a kernel is executed, the programmer specifies the **number of work-items** that should be created as an ***n-dimensional range*** (NDRange).
- An NDRange is a one-, two-, or three-dimensional array of work-items that will often map to the dimensions of either the input or the output data.

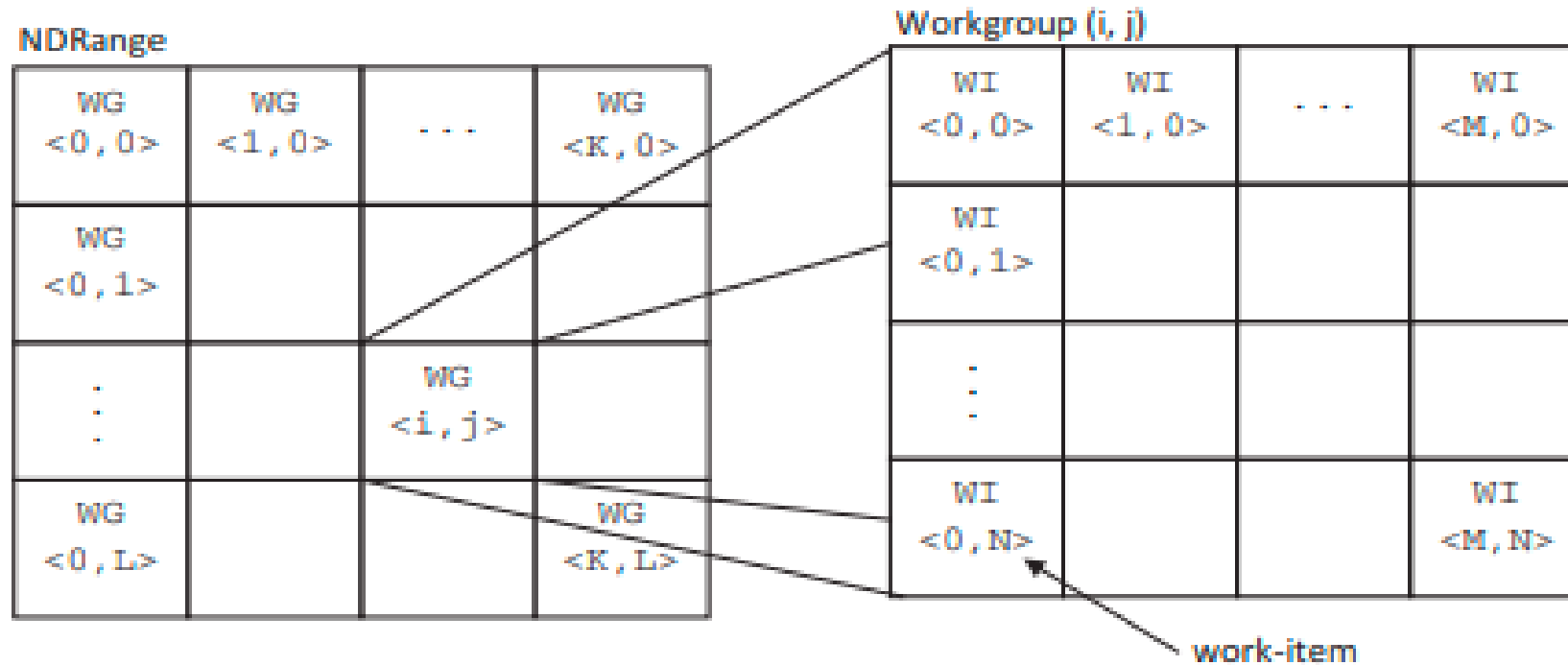
Example: `size_t WorkItemsSize[3] = {1024, 1, 1}`

- For achieving scalability, OpenCL divides the work-items of an NDRange into smaller, **equally sized** workgroups.

Example: `size_t workGroupSize[3] = {64, 1, 1}`

* This results in creating 16 work- groups ($1024 \text{ work-items} / (64 \text{ work-items per workgroup}) = 16 \text{ workgroups}$)

workItems-workGroup



Steps in Executing an OpenCL Program

- STEP 1:** Discover and initialize the platforms
- STEP 2:** Discover and initialize the devices
- STEP 3:** Create a context
- STEP 4:** Create a command queue
- STEP 5:** Create device buffers
- STEP 6:** Write host data to device buffers
- STEP 7:** Create and compile the program
- STEP 8:** Create the kernel
- STEP 9:** Set the kernel arguments
- STEP 10:** Configure the work-item structure
- STEP 11:** Enqueue the kernel for execution
- STEP 12:** Read the output buffer back to the host
- STEP 13:** Release OpenCL resources

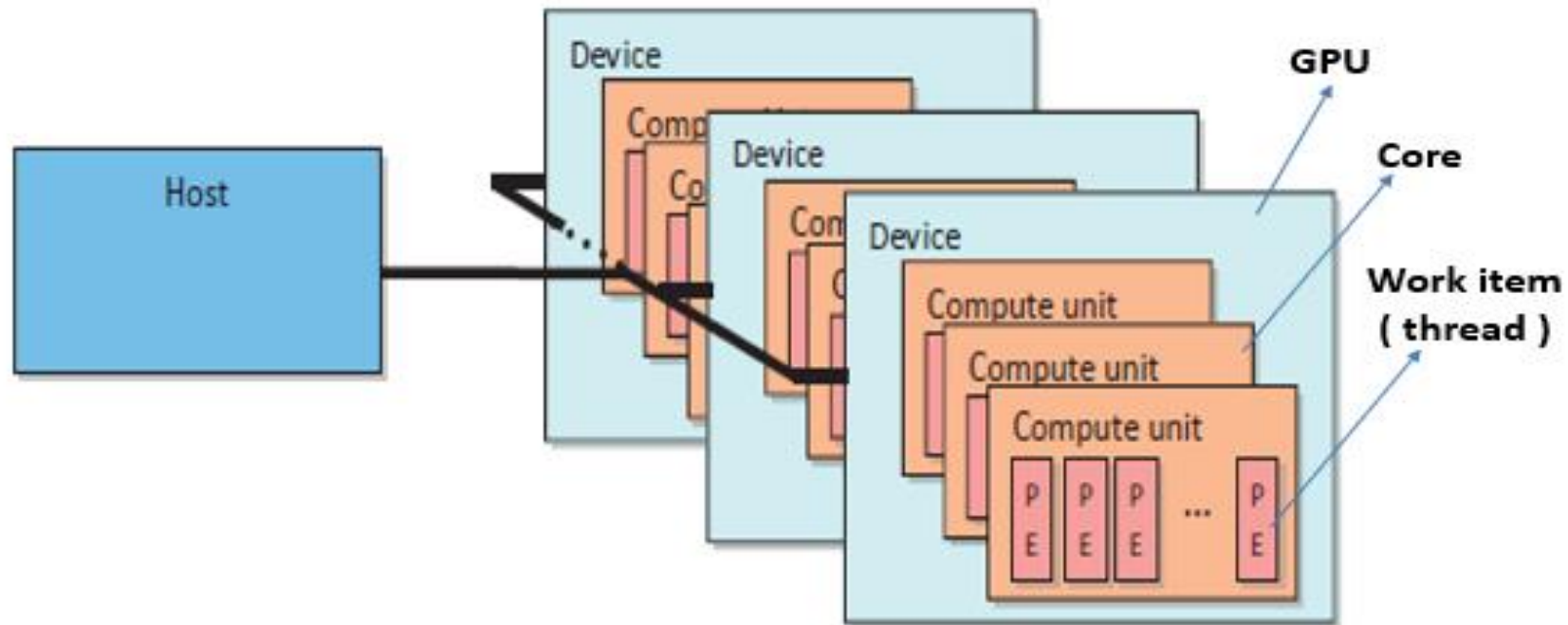
OpenCL primitive data types for host applications

Scalar data type	Bit width	Purpose
cl_char	8	Signed two's complement <u>integer</u>
cl_uchar	8	Unsigned two's complement integer
cl_short	16	Signed two's complement integer
cl_ushort	16	Unsigned two's complement integer
cl_int	32	Signed two's complement integer
cl_uint	32	Unsigned two's complement integer
cl_long	64	Signed two's complement integer
cl_ulong	64	Unsigned two's complement integer
cl_half	16	Half-precision floating-point value
cl_float	32	Single-precision floating-point value
cl_double	64	Double-precision floating-point value

Platform and Devices

- The OpenCL platform model **defines the roles of the host and devices** and provides an **abstract hardware model** for devices.
- In the platform model, there is a **single host** that coordinates execution on **one or more devices**.
- Platforms are **vendor-specific implementations** of the OpenCL API. The devices that a platform can target are thus limited to those with which a vendor knows how to interact.
- The platform model presents an **abstract device architecture** that programmers target when writing OpenCL C code. Vendors **map** this abstract architecture to the physical hardware.
- the platform model defines a device as an **array of compute units**, with each compute unit functionally independent from the rest. Compute units are further divided into **processing elements**

OpenCL abstract architecture for devices



STEP 1: Discover and initialize the platforms

- The API function **clGetPlatformIDs()** is used to discover the set of available platforms for a given system.

```
cl_int clGetPlatformIDs(  
    cl_uint num_entries,  
    cl_platform_id *platforms,  
    cl_uint *num_platforms )
```

- This API is often called twice by an application :
 - ❑ The first call passes an **unsigned int pointer** as the **num platforms** argument and **NULL** is passed as the **platforms** argument. The pointer is populated with the available number of platforms. The programmer can then allocate space to hold the platform information.
 - ❑ For the second call, a **cl_platform_id pointer** is passed with enough space allocated for **num_entries platforms**.
- After platforms have been discovered, the **clGetPlatformInfo()** call can be used to determine which implementation (vendor) the platform was defined by.

STEP 1: Discover and initialize the platforms

```
cl_uint num_platforms = 0;  
cl_platform_id *platforms = NULL;
```

```
// To get number of platforms
```

```
status= clGetPlatformIDs(0, NULL, &num_platforms);
```

```
// Allocate enough space for each platform
```

```
platforms=(cl_platform_id*) malloc (num_platforms*sizeof(cl_platform_id));
```

```
// To fill in Platforms info.
```

```
status=clGetPlatformIDs(num_platforms, platforms, NULL);
```

```
cl_int clGetPlatformIDs(  
    cl_uint num_entries,  
    cl_platform_id *platforms,  
    cl_uint *num_platforms )
```

- After platforms have been discovered, the **clGetPlatformInfo()** call can be used to determine which implementation (vendor) the platform was defined by.

STEP 2: Discover and Initialize the devices

- The API function **clGetDeviceIDs()** is used to discover the set of available devices for a given system.
- It works similar to `clGetPlatformIDs()` and takes the additional two arguments of a platform and a device type.

```
cl_int clGetDeviceIDs(  
    cl_platform_id platform,  
    cl_device_type device_type,  
    cl_uint num_entries,  
    cl_device_id *devices,  
    cl_uint *num_devices )
```

- The **device_type** argument can be used to limit the devices to

GPUs only	(CL_DEVICE_TYPE_GPU)
CPUs only	(CL_DEVICE_TYPE_CPU)
all devices	(CL_DEVICE_TYPE_ALL)

- As with platforms, **clGetDeviceInfo()** is called to retrieve information such as name, type, and vendor from each device.

STEP 2: Discover and initialize the devices

```
cl_uint num_devices = 0;  
cl_device_id *devices = NULL;
```

```
// To get number of devices
```

```
status=clGetDeviceIDs(platform[0], CL_DEVICE_TYPE_GPU, 0, NULL, &num_devices);
```

```
// Allocate enough space for each device
```

```
devices=(cl_device_id*) malloc (num_devices *sizeof(cl_device_id));
```

```
// To fill in Devices info.
```

```
status=clGetDeviceIDs(platform[0], CL_DEVICE_TYPE_GPU,num_devices,devices,NULL);
```

```
cl_int clGetDeviceIDs(  
    cl_platform_id platform,  
    cl_device_type device_type,  
    cl_uint num_entries,  
    cl_device_id *devices,  
    cl_uint *num_devices )
```

The **CLInfo** program in the AMD APP SDK uses the **clGetPlatformInfo()** and **clGetDeviceInfo()** commands to print detailed information about the OpenCL supported platforms and devices in a system.

```
$ ./CLInfo
Number of platforms: 1
Platform Profile: FULL_PROFILE
Platform Version: OpenCL 1.1 AMD-APP-SDK-v2.4
Platform Name: AMD Accelerated Parallel Processing
Platform Vendor: Advanced Micro Devices, Inc.
Number of devices: 2
Device Type: CL_DEVICE_TYPE_GPU
Name: Cypress
Max compute units: 20
Address bits: 32
Max memory allocation: 268435456
Global memory size: 1073741824
Constant buffer size: 65536
Local memory type: Scratchpad
Local memory size: 32768
Device endianness: Little

Device Type: CL_DEVICE_TYPE_CPU
Max compute units: 16
Name: AMD Phenom(tm) II X4 945 Processor
...
```

The Execution Environment

Before a host can request that a kernel be executed on a device, a **context** must be configured on the host that enables it to pass commands and data to the device.

Context:

- Context consists of one or more devices selected to work together
- Context is an abstract container that exists on the host and performs the following:
 - ❑ It coordinates the mechanisms for **host-device interactions**
 - ❑ It **Manages the memory objects** that are available to the devices
 - ❑ It **Keeps track of the programs and kernels** that are created for each device

STEP 3: Create a context

- The API function `cl_context clCreateContext` is used to discover the set of available devices for a given system.

```
cl_context clCreateContext (  
    const cl_context_properties *properties,  
    cl_uint num_devices,  
    const cl_device_id *devices,  
    void (CL_CALLBACK *pfn_notify)(  
        const char *errinfo,  
        const void *private_info,  
        size_t cb,  
        void *user_data),  
    void *user_data,  
    cl_int *errcode_ret)
```

- The **properties** argument is used to restrict the scope of the context to a specific platform, enable graphics interoperability, or enable other parameters in the future.
- The next two arguments specify the **number** and **IDs** of the devices that the programmer wants to associate with the context.
- OpenCL allows user **callbacks** to report additional error information that might be generated throughout its lifetime.

STEP 3: Create a context

```
Cl_int status
cl_context context = NULL;

// Create a context and associate it with the devices
context = clCreateContext (
    NULL,
    numDevices,
    devices,
    NULL,
    NULL,
    &status);
```

```
cl_context clCreateContext (
    const cl_context_properties *properties,
    cl_uint num_devices,
    const cl_device_id *devices,
    void (CL_CALLBACK *pfn_notify)(
        const char *errinfo,
        const void *private_info,
        size_t cb,
        void *user_data),
    void *user_data,
    cl_int *errcode_ret)
```

- After creating a context, the function **clGetContextInfo()** can be used to query information such as the **number of devices present** and the **device structures**.

The Execution Environment

Command Queues:

- Communication with a device occurs by submitting commands to a command queue.
- Once the host decides which devices to work with and a context is created, **one command queue needs to be created per device** (i.e., each command queue is associated with only one device).
- Whenever the host needs an action to be performed by a device, it will submit commands to the proper command queue.

STEP 4: Create a command queue

- The API function `clCreateCommandQueue()` is used to create a command queue and associate it with a device.

```
cl_command_queue clCreateCommandQueue (  
    cl_context context,  
    cl_device_id device,  
    cl_command_queue_properties properties,  
    cl_int *errcode_ret )
```

- The `properties` parameter is a *bit field* that is used to enable profiling of commands :
 - ☐ `CL_QUEUE_PROFILING_ENABLE` to enable/disable profiling of commands (for performance analysis)
 - ☐ `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` to allow out-of-order execution of commands
 - ☐ By default, it is in-order execution of commands
- Any API that specifies **host-device interaction** will always begin with `clEnqueue` and require a command queue as a parameter.

STEP 4: Create a command queue

```
cl_command_queue cmdQueue;
```

```
// Create a command queue and associate it with the device you want to execute on
```

```
cmdQueue = clCreateCommandQueue(  
    context,  
    devices[0],  
    0,  
    &status);
```

```
cl_command_queue clCreateCommandQueue (  
    cl_context context,  
    cl_device_id device,  
    cl_command_queue_properties properties,  
    cl_int *errcode_ret )
```

The Execution Environment

Memory Objects:

- In OpenCL the **data needs to be physically present on a device** before execution can begin.
- In order for data to be transferred to a device it must first be **encapsulated as a memory object**.
- Whenever a memory object is created, it is valid only within a single context.

2 types:

- ✓ buffers
- ✓ images

- **Buffers** are equivalent to arrays in C, created using malloc(), where data elements are stored contiguously in memory.
- **Images**, on the other hand, are designed as opaque objects, allowing for data padding and other optimizations that may improve performance on devices.

Opaque Objects:

MPI manages **system memory** that is used for buffering messages and for storing internal representations of various MPI objects such as groups, communicators, datatypes, etc. This memory is not directly accessible to the user, and objects stored there are **opaque**: their size and shape is not visible to the user. Opaque objects are accessed via **handles**, which exist in user space.

STEP 5: Create device buffers

- The API function `clCreateBuffer()` allocates the buffer and returns a memory object:

```
cl_mem clCreateBuffer(  
    cl_context context,  
    cl_mem_flags flags,  
    size_t size,  
    void *host_ptr,  
    cl_int *errcode_ret)
```

- Creating a buffer requires supplying the `size` of the buffer and a `context` in which the buffer will be allocated; it is visible for all devices associated with the context.
- The `host_ptr` is used to initialize the buffer.
- The `flags` parameter can take the following options:
 - ☐ `CL_MEM_READ_ONLY`
 - ☐ `CL_MEM_WRITE_ONLY`
 - ☐ `CL_MEM_READ_WRITE`

STEP 5: Create device buffers

```
cl_mem clCreateBuffer(  
    cl_context context,  
    cl_mem_flags flags,  
    size_t size,  
    void *host_ptr,  
    cl_int *errcode_ret)
```

```
cl_mem bufferA; // Input array on the device
```

```
cl_mem bufferB; // Input array on the device
```

```
cl_mem bufferC; // Output array on the device
```

```
// Create a buffer object that will contain the data from the host array A
```

```
bufferA = clCreateBuffer( context, CL_MEM_READ_ONLY, datasize, NULL, &status);
```

```
// Create a buffer object that will contain the data from the host array B
```

```
bufferB = clCreateBuffer( context, CL_MEM_READ_ONLY, datasize, NULL, &status);
```

```
// Create a buffer object with enough space to hold the output data
```

```
bufferC = clCreateBuffer( context, CL_MEM_WRITE_ONLY, datasize, NULL, &status);
```

STEP 6: Write host data to device buffers

- Data contained in host memory is transferred to and from an OpenCL buffer using the commands `clEnqueueWriteBuffer()` and `clEnqueueReadBuffer()`, respectively.
- The API calls for reading and writing to buffers are very similar.

```
cl_int clEnqueueWriteBuffer (  
    cl_command_queue command_queue,  
    cl_mem buffer,  
    cl_bool blocking_write,  
    size_t offset,  
    size_t cb,  
    const void *ptr,  
    cl_uint num_events_in_wait_list,  
    const cl_event *event_wait_list,  
    cl_event *event)
```

Parameters:

- `command_queue` refers to the command-queue in which the write command will be queued
- `blocking_write` indicates if the write operation is blocking (**CL_TRUE**) or nonblocking (**CL_FALSE**)
- `offset` in bytes in the buffer object to write to
- `cb` is the size in bytes of data being written

STEP 6: Write host data to device buffers

Parameters:

- **ptr** refers to the pointer to buffer in host memory where data is to be written from
- **event_wait_list, num_events_in_wait_list** parameters specify events that need to complete before this particular command can be executed. If *event_wait_list* is *NULL*, then this particular command does not wait on any event to complete. If *event_wait_list* is *NULL*, *num_events_in_wait_list* must be **0**. If *event_wait_list* is *not NULL*, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be **greater than 0**
- **event** returns an event object

```
cl_int clEnqueueWriteBuffer (  
    cl_command_queue command_queue,  
    cl_mem buffer,  
    cl_bool blocking_write,  
    size_t offset,  
    size_t cb,  
    const void *ptr,  
    cl_uint num_events_in_wait_list,  
    const cl_event *event_wait_list,  
    cl_event *event)
```

STEP 6: Write host data to device buffers

```
cl_int clEnqueueWriteBuffer (  
    cl_command_queue command_queue,  
    cl_mem buffer,  
    cl_bool blocking_write,  
    size_t offset,  
    size_t cb,  
    const void *ptr,  
    cl_uint num_events_in_wait_list,  
    const cl_event *event_wait_list,  
    cl_event *event)
```

```
// Write input array A to the device buffer bufferA
```

```
status = clEnqueueWriteBuffer( cmdQueue, bufferA, CL_FALSE, 0, datasize, A, 0, NULL, NULL);
```

```
// Write input array B to the device buffer bufferB
```

```
status = clEnqueueWriteBuffer( cmdQueue, bufferB, CL_FALSE, 0, datasize, B, 0, NULL, NULL);
```

The Execution Environment

Creating an OpenCL program object:

- OpenCL C code (written to run on an OpenCL device) is called a **program**.
- A program is a collection of functions called **kernels**, where kernels are **units of execution** that can be scheduled to run on a device.
- OpenCL programs are **compiled at runtime** through a series of API calls.
- This runtime compilation gives the system an opportunity to optimize for a specific device.
- The process of creating a kernel is as follows:
 1. The OpenCL C source code is stored in a character string. If the source code is stored in a file on a disk, it must be read into memory and stored as a character array.
 2. The source code is turned into a program object, **cl_program**, by calling **clCreate ProgramWithSource()**.
 3. The program object is then compiled, for one or more OpenCL devices, with **clBuildProgram()**. If there are compile errors, they will be reported here.

The Execution Environment

Transferring kernel code to a character array

```
#define MAX_SOURCE_SIZE (0x100000)

FILE *fp;
char *source_str;
size_t source_size;

fp = fopen("vector_add_kernel.cl", "r");
if (!fp) {
    fprintf(stderr, "Failed to load kernel.\n");
    exit(1);
}

source_str = (char*)malloc(MAX_SOURCE_SIZE);
source_size = fread( source_str, 1, MAX_SOURCE_SIZE, fp);
fclose( fp );
```

The OpenCL C source code is stored in a character string

```
const char* programSource =
    "__kernel                                     \n"
    "void vecadd(__global int *A,                 \n"
    "             __global int *B,                 \n"
    "             __global int *C)                 \n"
    "{                                              \n"
    "                                              \n"
    "    // Get the work-item's unique ID          \n"
    "    int idx = get_global_id(0);                \n"
    "                                              \n"
    "    // Add the corresponding locations of      \n"
    "    // 'A' and 'B', and store the result in 'C'. \n"
    "    C[idx] = A[idx] + B[idx];                 \n"
    "}                                              \n"
    ;
```

STEP 7: Create and compile the program

- The API function `clCreateProgramWithSource()` is used to create a OpenCL program with source code.

```
cl_program clCreateProgramWithSource(  
    cl_context context,  
    cl_uint count,  
    const char **strings,  
    const size_t *lengths,  
    cl_int *errcode_ret);
```

Parameters:

- program** is the program object
- context** is a valid context
- strings** An array of **count** pointers to optionally null-terminated character strings that make up the source code
- lengths** An array with the number of chars in each string (the string length). If an element in lengths is **zero**, its accompanying string is null-terminated. If lengths is **NULL**, all strings in the strings argument are considered null-terminated

STEP 7: Create and compile the program

- The API function `clBuildProgram()` is used to build (compile) the program for the connected devices.

```
cl_int clBuildProgram(  
    cl_program program,  
    cl_uint num_devices,  
    cl_device_id * device_list,  
    const char *options,  
    void (pfn_notify *) (cl_program, void *user_data),  
    void *user_data)
```

Parameters:

- `program` is the program object
- `num_devices` is the number of devices listed in `device_list`
- `device_list` is a pointer to a list of devices that are in program
- `options` is a pointer to a string that describes the build options to be used for building the program executable
- `pfn_notify` is a function pointer to a notification routine. The notification routine is a callback function that an application can register and which will be called when the program executable has been built (successfully or unsuccessfully)
- `User_data` is passed as an argument when `pfn_notify` is called

STEP 7: Create and compile the program

```
cl_program clCreateProgramWithSource(  
    cl_context context,  
    cl_uint count,  
    const char **strings,  
    const size_t *lengths,  
    cl_int *errcode_ret);
```

```
cl_int clBuildProgram(  
    cl_program program,  
    cl_uint num_devices,  
    cl_device_id * device_list,  
    const char *options,  
    void (pfn_notify *) (cl_program, void *user_data),  
    void *user_data)
```

// Create a program

```
cl_program program = clCreateProgramWithSource( context, 1, (const char**)&programSource, NULL, &status);
```

// Build (compile) the program for the devices

```
status = clBuildProgram( program, numDevices, devices, NULL, NULL, NULL);
```

The Execution Environment

The OpenCL Kernel

- The final stage in OpenCL program execution is to obtain a `cl_kernel` object that can be used to **execute kernels on a device** by extracting the kernel from the `cl_program`.
- The **name of the kernel** is passed to `clCreateKernel()`, along with the **program object**, and the kernel object will be returned if the program object was valid, and the particular kernel is found.

```
cl_kernel clCreateKernel(  
    cl_program program,  
    const char *kernel_name ,  
    cl_int *errcode_ret);
```


STEP 8: Create the kernel

```
cl_kernel clCreateKernel(  
    cl_program program,  
    const char *kernel_name ,  
    cl_int *errcode_ret);
```

```
cl_kernel kernel = NULL;  
// Create a kernel from the vector addition function (named "vecadd")  
kernel = clCreateKernel(program, "vecadd", &status);
```

STEP 9: Set the kernel arguments

- Executing a kernel requires dispatching it through an enqueue function.
- We must specify each kernel argument individually using the function `clSetKernelArg()`.
- This function takes a **kernel object**, an index specifying the **argument number**, the **size of the argument**, and a **pointer to the argument**.

```
cl_int clSetKernelArg(  
    cl_kernel kernel,  
    cl_uint arg_index ,  
    size_t arg_size,  
    const void *arg_value);
```

- We must specify each kernel argument individually using the function `clSetKernelArg()`.
- When the kernel is executed, this information is used to transfer arguments to the device.

STEP 9: Set the kernel arguments

```
cl_int clSetKernelArg(  
    cl_kernel kernel,  
    cl_uint arg_index ,  
    size_t arg_size,  
    const void *arg_value);
```

// Associate the input and output buffers with the kernel

```
status = clSetKernelArg( kernel, 0, sizeof(cl_mem), &bufferA);  
status |= clSetKernelArg( kernel, 1, sizeof(cl_mem), &bufferB);  
status |= clSetKernelArg( kernel, 2, sizeof(cl_mem), &bufferC);
```

STEP 10: Configure the work-item structure

```
// Define an index space (global work size) of work items for execution.  
// A workgroup size (local work size) is not required, but can be used.  
size_t globalWorkSize[1];  
// There are 'elements(1024)' work-items  
globalWorkSize[0] = elements;
```

Work Dimension is **1** and `global_work_size` (GSZ)=**1024**

`local_work_size` (LSZ)=**128**

Since there are **1024** total work-items and **128** work-items / work-group, there are :

$$\mathbf{1024 / 128 = 8 \text{ work-groups (WG).}}$$

STEP 11: Enqueue the kernel for execution

- After any required memory objects are transferred to the device and the kernel arguments are set, the kernel is ready to be executed.
- We can request that a device begin executing a kernel with a call to `clEnqueueNDRangeKernel()`.

```
cl_int clEnqueueNDRangeKernel(  
    cl_command_queue command_queue,  
    cl_kernel kernel,  
    cl_uint work_dim,  
    const size_t *global_work_offset,  
    const size_t *global_work_size,  
    const size_t *local_work_size,  
    cl_uint num_events_in_wait_list,  
    const cl_event *event_wait_list,  
    cl_event *event)
```

Four fields are related to work-item creation:

- **work_dim**: Specifies the number of dimensions in which work-item will be created
- **global_work_size**: Specifies the number of work items in each dimension of the ND range
- **local_work_size**: Specifies the number of work items in each dimension of the workgroup
- **global_work_offset**: Used to provide global IDs to the work items that do not start from zero

STEP 11: Enqueue the kernel for execution

```
cl_int clEnqueueNDRangeKernel(  
    cl_command_queue command_queue,  
    cl_kernel kernel,  
    cl_uint work_dim,  
    const size_t *global_work_offset,  
    const size_t *global_work_size,  
    const size_t *local_work_size,  
    cl_uint num_events_in_wait_list,  
    const cl_event *event_wait_list,  
    cl_event *event)
```

//Execute the kernel

```
status = clEnqueueNDRangeKernel( cmdQueue,  
                                kernel,  
                                1,  
                                NULL,  
                                globalWorkSize,  
                                NULL,  
                                0,  
                                NULL,  
                                NULL);
```

The `clEnqueueNDRangeKernel()` call is **asynchronous**: it will return immediately after the command is enqueued in the command queue

STEP 12: Read the output buffer back to the host

- Once the result is computed by the device, it is placed in the output buffer from where the host read it by calling `clEnqueueReadBuffer()`.

```
cl_int clEnqueueReadBuffer (  
    cl_command_queue command_queue,  
    cl_mem buffer,  
    cl_bool blocking_write,  
    size_t offset,  
    size_t cb,  
    const void *ptr,  
    cl_uint num_events_in_wait_list,  
    const cl_event *event_wait_list,  
    cl_event *event)
```

Four fields are related to work-item creation:

- **work_dim**: Specifies the number of dimensions in which work-item will be created
- **global_work_size**: Specifies the number of work items in each dimension of the ND range
- **local_work_size**: Specifies the number of work items in each dimension of the workgroup
- **global_work_offset**: Used to provide global IDs to the work items that do not start from zero

STEP 12: Read the output buffer back to the host

```
cl_int clEnqueueReadBuffer (  
    cl_command_queue command_queue,  
    cl_mem buffer,  
    cl_bool blocking_write,  
    size_t offset,  
    size_t cb,  
    const void *ptr,  
    cl_uint num_events_in_wait_list,  
    const cl_event *event_wait_list,  
    cl_event *event)
```

```
// Read the OpenCL output buffer (bufferC) to the host output array (C)
```

```
clEnqueueReadBuffer( cmdQueue, bufferC, CL_TRUE, 0, datasize, C, 0, NULL, NULL);
```

```
// Verify the output
```

```
bool result = true;
```

```
for(int i = 0; i < elements; i++)
```

```
{  
    if(C[i] != i+i)  
    {  
        result = false;  
        break;  
    }  
}
```

```
if(result)  
    printf("Output is correct\n");  
else  
    printf("Output is incorrect\n");
```

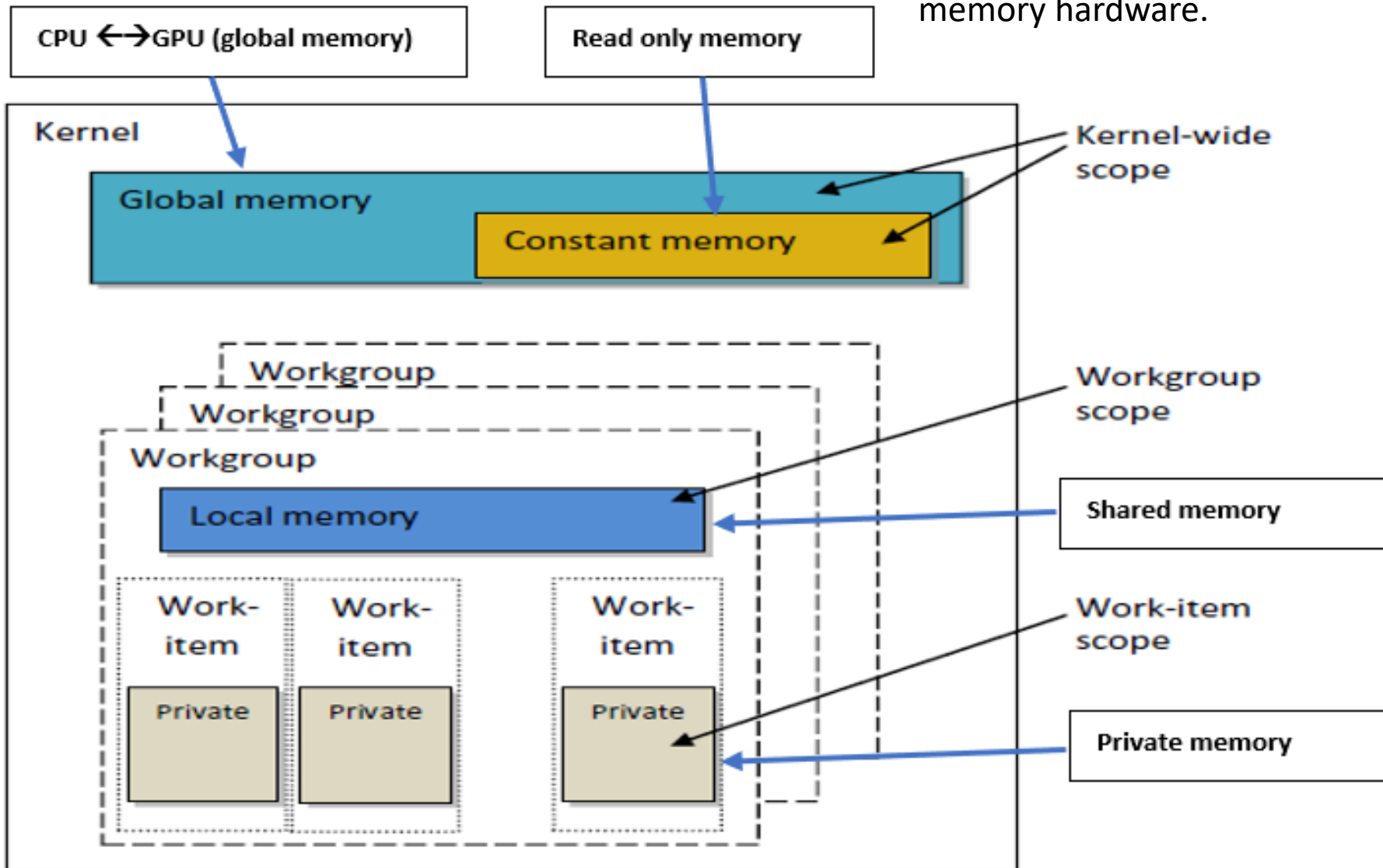

STEP 13: Release OpenCL resources

- The final step in OpenCL program execution is to release all the resources held by the program before the termination

```
// Free OpenCL resources
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cmdQueue);
clReleaseMemObject(bufferA);
clReleaseMemObject(bufferB);
clReleaseMemObject(bufferC);
clReleaseContext(context);
// Free host resources
free(A);
free(B);
free(C);
free(platforms);
free(devices);
```

Memory Model

- To support code portability, OpenCL's approach is to define an abstract memory model that programmers can target when writing code and vendors can map to their actual memory hardware.



Memory Model

Global memory:

- It is visible to all compute units on the device (similar to the main memory on a CPU-based host system).
- Whenever data is transferred from the host to the device OR from the device to the host, it must reside in global memory.
- The keyword **__global** is added to a pointer declaration to specify that data referenced by the pointer resides in global memory.

example: **__global float* A**

Constant memory:

- It is specifically designed for data where each element is accessed simultaneously by all work-items.
- Variables whose values never change (e.g., a data variable holding the value of π) also fall into this category.
- Constant memory is modelled as a part of global memory, so memory objects that are transferred to global memory can be specified as constant.
- Data is mapped to constant memory by using the **__constant** keyword.

Memory Model

Local memory:

- It is a scratchpad memory whose address space is unique to each compute device.
- Local memory is modelled as being shared by a workgroup.
- Inside a GPU, all work items of the same work group must be executed on a single "core".
- You can synchronize threads (work items) inside a work group, because they all are resident in the same core.
- Data is mapped to local memory by using the `__local` keyword.

Private memory:

- This memory is unique to an individual work-item.
- Local variables and nonpointer kernel arguments are private by default.

Writing Kernels

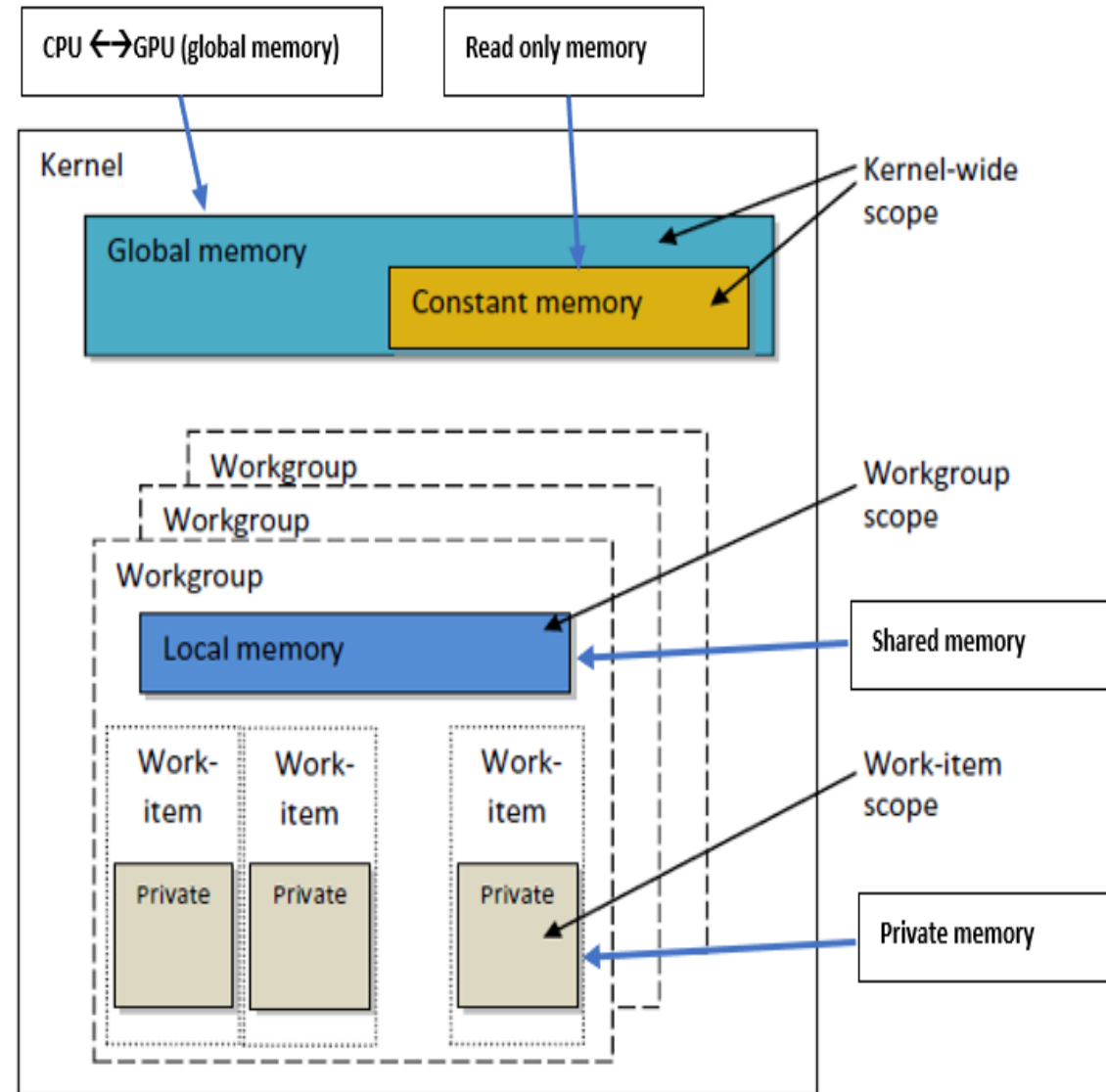
- OpenCL C kernels are similar to C functions.
- Kernel will be executed once for every work-item created.
- Kernels begin with the keyword **__kernel** and must have a return type of void.
- Buffers can be declared in global memory (**__global**) or constant memory (**__constant**).
- Kernel arguments can also use optional access qualifiers (**__read_only**, **__write_only**, and **__read_write**).
- The **__local** qualifier is used to declare memory that is shared between all work-items in a workgroup.

Writing Kernels

- When programming for OpenCL devices, particularly GPUs, performance may increase by using local memory to cache data that will be used multiple times by a work-item or by multiple work-items in the same workgroup.

```
__kernel void cache( __global float* data,  
__local float* sharedData)  
{  
    int globalId = get_global_id(0);  
    int localId = get_local_id(0);  
    // Cache data to local memory  
    sharedData[localId] = data[globalId];  
    ...  
}
```

- Once a work-item completes its execution, none of its state information or local memory storage is persistent. Any results that need to be kept must be transferred to global memory.



Vector-vector addition example

Header files

```
#include <stdio.h>
#include <stdlib.h>
#include <CL/cl.h>
```

OpenCL kernel

```
const char* programSource =
    "__kernel                                \n"
    "void vecadd(__global int *A,            \n"
    "             __global int *B,            \n"
    "             __global int *C)           \n"
    "{                                         \n"
    "                                         \n"
    "    // Get the work-item's unique ID     \n"
    "    int idx = get_global_id(0);          \n"
    "                                         \n"
    "    // Add the corresponding locations of \n"
    "    // 'A' and 'B', and store the result in 'C'. \n"
    "    C[idx] = A[idx] + B[idx];            \n"
    "}                                         \n"
    ;
```

Host code

```
int main() {
```

```
    // Host data buffers
```

```
    int *A = NULL;    // Input array
```

```
    int *B = NULL;    // Input array
```

```
    int *C = NULL;    // Output array
```

```
    // Elements in each array
```

```
    const int elements = 2048; // size of host data buffers
```

```
    // Compute the size of the data in bytes
```

```
    size_t datasize = sizeof(int)*elements;
```

```
    // Dynamically allocate space for input/output host data buffers
```

```
    A = (int*)malloc(datasize);
```

```
    B = (int*)malloc(datasize);
```

```
    C = (int*)malloc(datasize);
```

```
    // Initialize the input data
```

```
    for(int i = 0; i < elements; i++)
```

```
    {
```

```
        A[i] = i;
```

```
        B[i] = i;
```

```
    }
```

```
    // Use this to check the output of each API call
```

```
    cl_int status;
```


STEP 1: Discover and initialize the platforms

```
cl_uint num_platforms = 0;  
cl_platform_id *platforms = NULL;
```

```
// To get number of platforms
```

```
status= clGetPlatformIDs(0, NULL, &num_platforms);
```

```
// Allocate enough space for each platform
```

```
platforms=(cl_platform_id*) malloc (num_platforms*sizeof(cl_platform_id));
```

```
// To fill in Platforms info.
```

```
status=clGetPlatformIDs(num_platforms, platforms, NULL);
```

```
cl_int clGetPlatformIDs(  
    cl_uint num_entries,  
    cl_platform_id *platforms,  
    cl_uint *num_platforms )
```

STEP 2: Discover and initialize the devices

```
cl_uint num_devices = 0;  
cl_device_id *devices = NULL;
```

```
// To get number of devices
```

```
status=clGetDeviceIDs(platform[0], CL_DEVICE_TYPE_GPU, 0, NULL, &num_devices);
```

```
// Allocate enough space for each device
```

```
devices=(cl_device_id*) malloc (num_devices *sizeof(cl_device_id));
```

```
// To fill in Devices info.
```

```
status=clGetDeviceIDs(platform[0], CL_DEVICE_TYPE_GPU,num_devices,devices,NULL);
```

```
cl_int clGetDeviceIDs(  
    cl_platform_id platform,  
    cl_device_type device_type,  
    cl_uint num_entries,  
    cl_device_id *devices,  
    cl_uint *num_devices )
```

STEP 3: Create a context

```
Cl_int status
cl_context context = NULL;

// Create a context and associate it with the devices
context = clCreateContext (
    NULL,
    numDevices,
    devices,
    NULL,
    NULL,
    &status);
```

```
cl_context clCreateContext (
    const cl_context_properties *properties,
    cl_uint num_devices,
    const cl_device_id *devices,
    void (CL_CALLBACK *pfn_notify)(
        const char *errinfo,
        const void *private_info,
        size_t cb,
        void *user_data),
    void *user_data,
    cl_int *errcode_ret)
```

STEP 4: Create a command queue

```
cl_command_queue cmdQueue;
```

```
// Create a command queue and associate it with the device you want to execute on
```

```
cmdQueue = clCreateCommandQueue(  
    context,  
    devices[0],  
    0,  
    &status);
```

```
cl_command_queue clCreateCommandQueue (  
    cl_context context,  
    cl_device_id device,  
    cl_command_queue_properties properties,  
    cl_int *errcode_ret )
```

STEP 5: Create device buffers

```
cl_mem bufferA; // Input array on the device
cl_mem bufferB; // Input array on the device
cl_mem bufferC; // Output array on the device
```

```
// Create a buffer object that will contain the data from the host array A
```

```
bufferA = clCreateBuffer( context, CL_MEM_READ_ONLY, datasize, NULL, &status);
```

```
// Create a buffer object that will contain the data from the host array B
```

```
bufferB = clCreateBuffer( context, CL_MEM_READ_ONLY, datasize, NULL, &status);
```

```
// Create a buffer object with enough space to hold the output data
```

```
bufferC = clCreateBuffer( context, CL_MEM_WRITE_ONLY, datasize, NULL, &status);
```

```
cl_mem clCreateBuffer(
    cl_context context,
    cl_mem_flags flags,
    size_t size,
    void *host_ptr,
    cl_int *errcode_ret)
```

STEP 6: Write host data to device buffers

```
cl_int clEnqueueWriteBuffer (  
    cl_command_queue command_queue,  
    cl_mem buffer,  
    cl_bool blocking_write,  
    size_t offset,  
    size_t cb,  
    const void *ptr,  
    cl_uint num_events_in_wait_list,  
    const cl_event *event_wait_list,  
    cl_event *event)
```

```
// Write input array A to the device buffer bufferA
```

```
status = clEnqueueWriteBuffer( cmdQueue, bufferA, CL_FALSE, 0, datasize, A, 0, NULL, NULL);
```

```
// Write input array B to the device buffer bufferB
```

```
status = clEnqueueWriteBuffer( cmdQueue, bufferB, CL_FALSE, 0, datasize, B, 0, NULL, NULL);
```

STEP 7: Create and compile the program

```
cl_program clCreateProgramWithSource(  
    cl_context context,  
    cl_uint count,  
    const char **strings,  
    const size_t *lengths,  
    cl_int *errcode_ret);
```

```
cl_int clBuildProgram(  
    cl_program program,  
    cl_uint num_devices,  
    cl_device_id * device_list,  
    const char *options,  
    void (pfn_notify *) (cl_program, void *user_data),  
    void *user_data)
```

// Create a program

```
cl_program program = clCreateProgramWithSource( context, 1, (const char**)&programSource, NULL, &status);
```

// Build (compile) the program for the devices

```
status = clBuildProgram( program, numDevices, devices, NULL, NULL, NULL);
```

STEP 8: Create the kernel

```
cl_kernel clCreateKernel(  
    cl_program program,  
    const char *kernel_name ,  
    cl_int *errcode_ret);
```

```
cl_kernel kernel = NULL;  
// Create a kernel from the vector addition function (named "vecadd")  
kernel = clCreateKernel(program, "vecadd", &status);
```

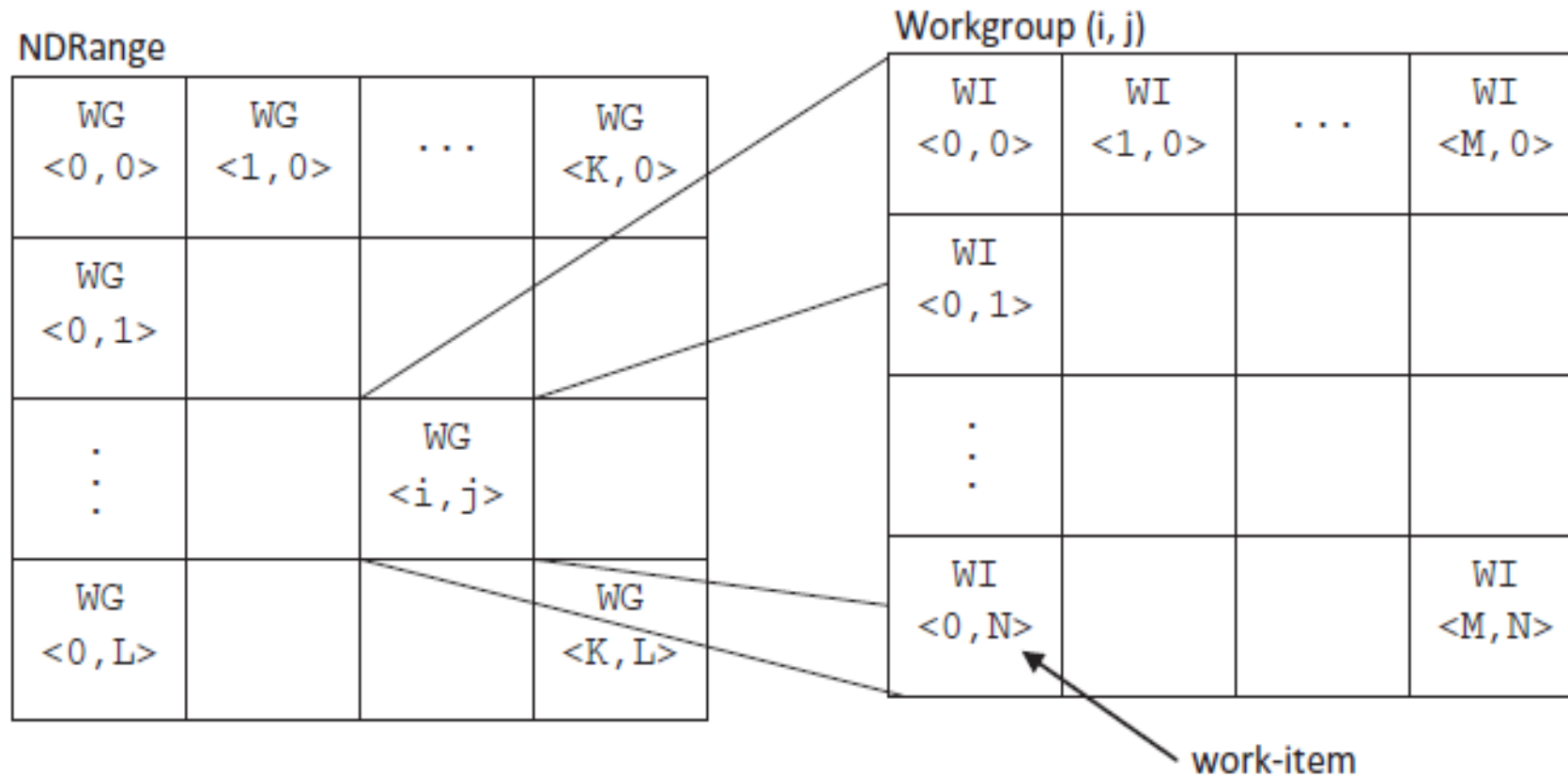

STEP 9: Set the kernel arguments

```
cl_int clSetKernelArg(  
    cl_kernel kernel,  
    cl_uint arg_index ,  
    size_t arg_size,  
    const void *arg_value);
```

```
// Associate the input and output buffers with the kernel
```

```
status = clSetKernelArg( kernel, 0, sizeof(cl_mem), &bufferA);  
status |= clSetKernelArg( kernel, 1, sizeof(cl_mem), &bufferB);  
status |= clSetKernelArg( kernel, 2, sizeof(cl_mem), &bufferC);
```

Work-items & Work groups



STEP 10: Configure the work-item structure

```
// Define an index space (global work size) of work items for  
execution. // A workgroup size (local work size) is not required,  
but can be used. size_t globalWorkSize[1];  
// There are 'elements(1024)' work-items  
globalWorkSize[0] = elements;
```

STEP 11: Enqueue the kernel for execution

```
cl_int clEnqueueNDRangeKernel(  
    cl_command_queue command_queue,  
    cl_kernel kernel,  
    cl_uint work_dim,  
    const size_t *global_work_offset,  
    const size_t *global_work_size,  
    const size_t *local_work_size,  
    cl_uint num_events_in_wait_list,  
    const cl_event *event_wait_list,  
    cl_event *event)
```

//Execute the kernel

```
status = clEnqueueNDRangeKernel( cmdQueue,  
                                kernel,  
                                1,  
                                NULL,  
                                globalWorkSize,  
                                NULL,  
                                0,  
                                NULL,  
                                NULL);
```

STEP 12: Read the output buffer back to the host

```
cl_int clEnqueueReadBuffer (  
    cl_command_queue command_queue,  
    cl_mem buffer,  
    cl_bool blocking_write,  
    size_t offset,  
    size_t cb,  
    const void *ptr,  
    cl_uint num_events_in_wait_list,  
    const cl_event *event_wait_list,  
    cl_event *event)
```

```
// Read the OpenCL output buffer (bufferC) to the host output array (C)
```

```
clEnqueueReadBuffer( cmdQueue, bufferC, CL_TRUE, 0, datasize, C, 0, NULL, NULL);
```

```
// Verify the output
```

```
bool result = true;
```

```
for(int i = 0; i < elements; i++)
```

```
{  
    if(C[i] != i+i)  
    {  
        result = false;  
        break;  
    }  
}
```

```
if(result)  
    printf("Output is correct\n");
```

```
else  
    printf("Output is incorrect\n");
```

STEP 13: Release OpenCL resources


```
// Free OpenCL resources
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cmdQueue);
clReleaseMemObject(bufferA);
clReleaseMemObject(bufferB);
clReleaseMemObject(bufferC);
clReleaseContext(context);
// Free host resources
free(A);
free(B);
free(C);
free(platforms);
free(devices);
```

Calculation of Time taken for kernel function execution

Profiling of OpenCL commands can be enabled by using a command-queue created with CL_QUEUE_PROFILING_ENABLE flag set in properties argument to [clCreateCommandQueue](#)

CL_QUEUE_PROFILING_ENABLE

```
cl_command_queue clCreateCommandQueue (
    cl_context context,
    cl_device_id device,
    cl_command_queue_properties properties,
    cl_int *errcode_ret )
```



Calculation of Time taken for kernel function execution

```
cl_int clGetEventProfilingInfo  
(  
    cl_event      event,  
    cl_profiling_info param_name,  
    size_t        param_value_size,  
    void          *param_value,  
    size_t        *param_value_size_ret  
);
```


Calculation of Time taken for kernel function execution

cl_profiling_info	Return Type	Info. returned in param_value
CL_PROFILING_COMMAND_START	cl_ulong	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by event starts execution on the device.
CL_PROFILING_COMMAND_END	cl_ulong	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by event has finished execution on the device.

Calculation of Time taken for kernel function execution

- `cl_event event;`
- `ret = clEnqueueNDRangeKernel (command_queue, kernel, 1, NULL, &global_item_size, &local_item_size, 0, NULL, & event);`
- `ret = clFinish(command_queue);`
- `cl_ulong time_start, time_end;`
- `double total_time;`
- `clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, sizeof(time_start), & time_start, NULL);`

Calculation of Time taken for kernel function execution

- `clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(time_end), & time_end, NULL);`
- `total_time = double (time_end - time_start);`
- `total_time=total_time/1000000; // m secs`

Barrier Operations for a command queue

- Two types of barrier operations for a command queue:

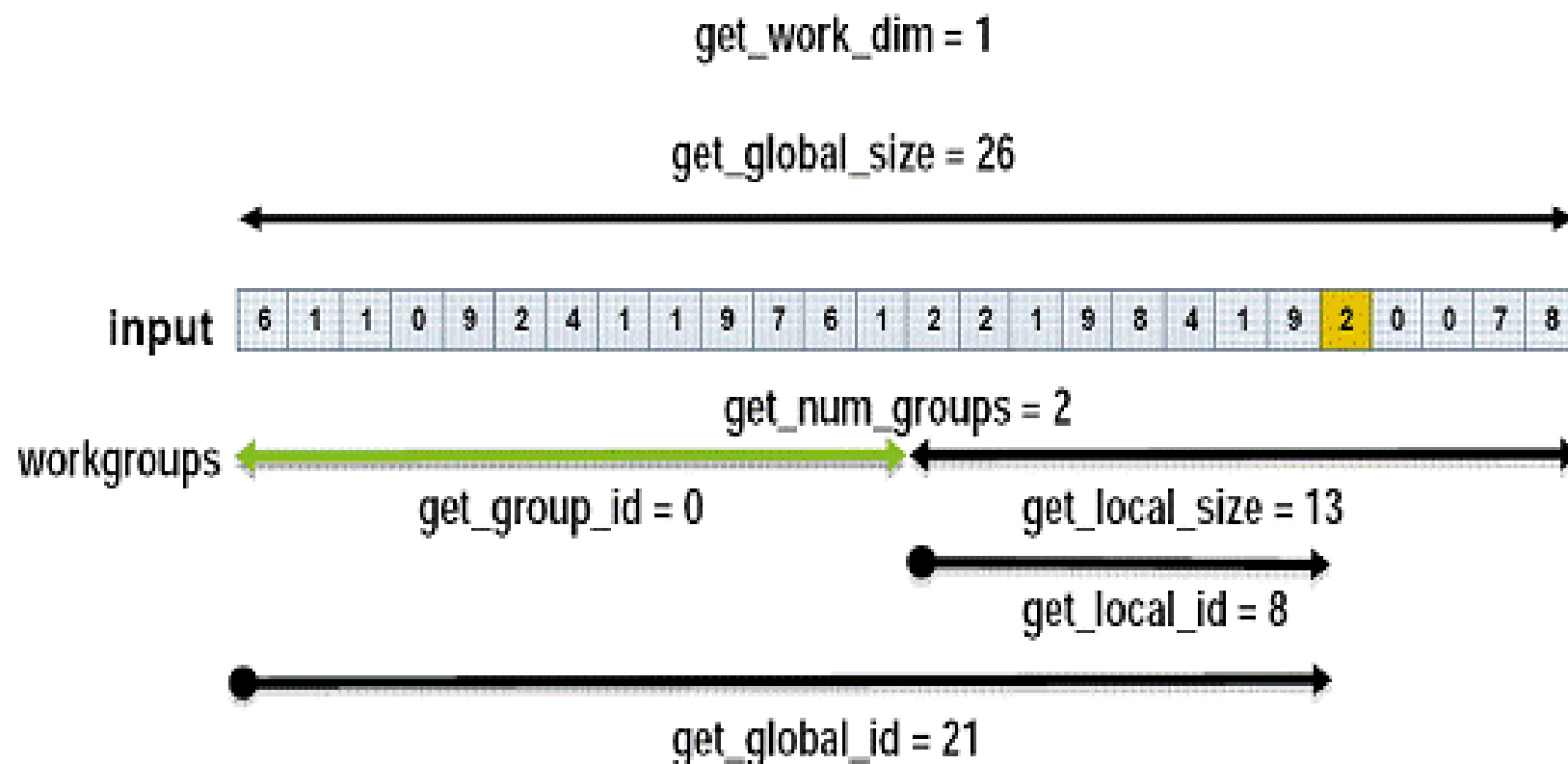
- ✓ `cl_int clFinish (cl_command_queue cmdQueue);`

This function blocks until all of the commands in a command queue have completely executed.

- ✓ `cl_int clFlush (cl_command_queue cmdQueue);`

This function blocks until all of the commands in a command queue have been removed from the command queue.

Work groups & Work items



OpenCL - Built-in Functions

Function	Property returned
<code>uint get_work_dim()</code>	The number of dimensions
<code>size_t get_global_id(uint dimidx)</code>	The ID of the current work-item [0,WI) in dimension dimidx
<code>size_t get_global_size(uint dimidx)</code>	The total number of work-items (WI) in dimension dimidx
<code>size_t get_global_offset(uint dimidx)</code>	The offset as specified in the enqueueNDRangeKernel API in dimension dimidx
<code>size_t get_group_id(uint dimidx)</code>	The ID of the current work-group [0, WG) in dimension dimidx
<code>size_t get_local_id(uint dimidx)</code>	The ID of the work-item within the work-group [0, WI/WG) in dimension dimidx
<code>size_t get_local_size (uint dimidx)</code>	The number of work-items per work-group = WI/WG in dimension dimidx
<code>size_t get_num_groups(uint dimidx)</code>	The total number of work-groups (WG) in dimension dimidx