Introduction to OpenCL

Cliff Woolley, NVIDIA
Developer Technology Group

Welcome to the OpenCL Tutorial!

- OpenCL Platform Model
- OpenCL Execution Model
- Mapping the Execution Model onto the Platform Model
- Introduction to OpenCL Programming
- Additional Information and Resources

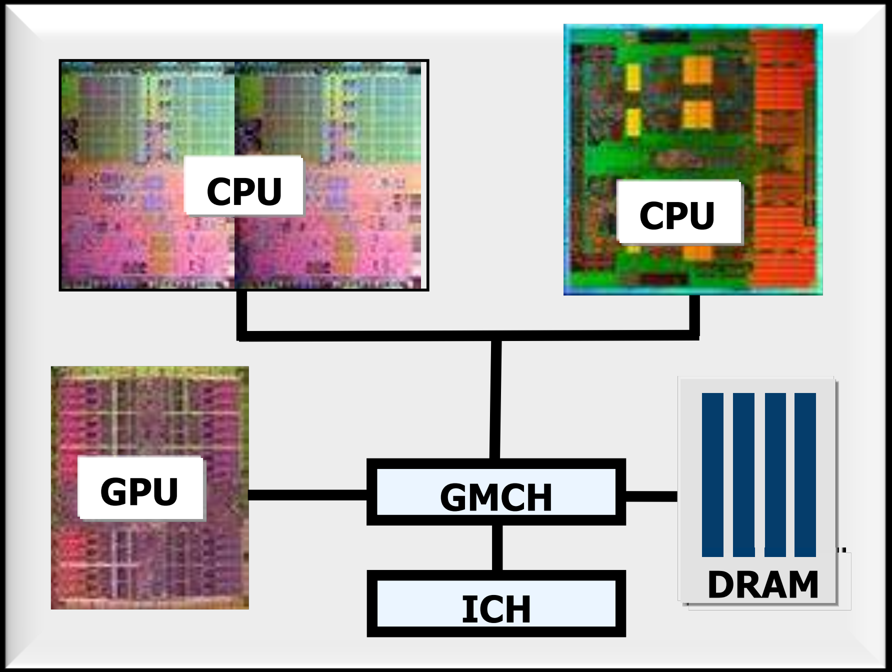
Design Goals of OpenCL

- Use all computational resources in the system
  - CPUs, GPUs and other processors as peers
- Efficient parallel programming model
  - Based on C99
  - Data- and task- parallel computational model
  - Abstract the specifics of underlying hardware
  - Specify accuracy of floating-point computations
- Desktop and Handheld Profiles

OPENCL PLATFORM MODEL

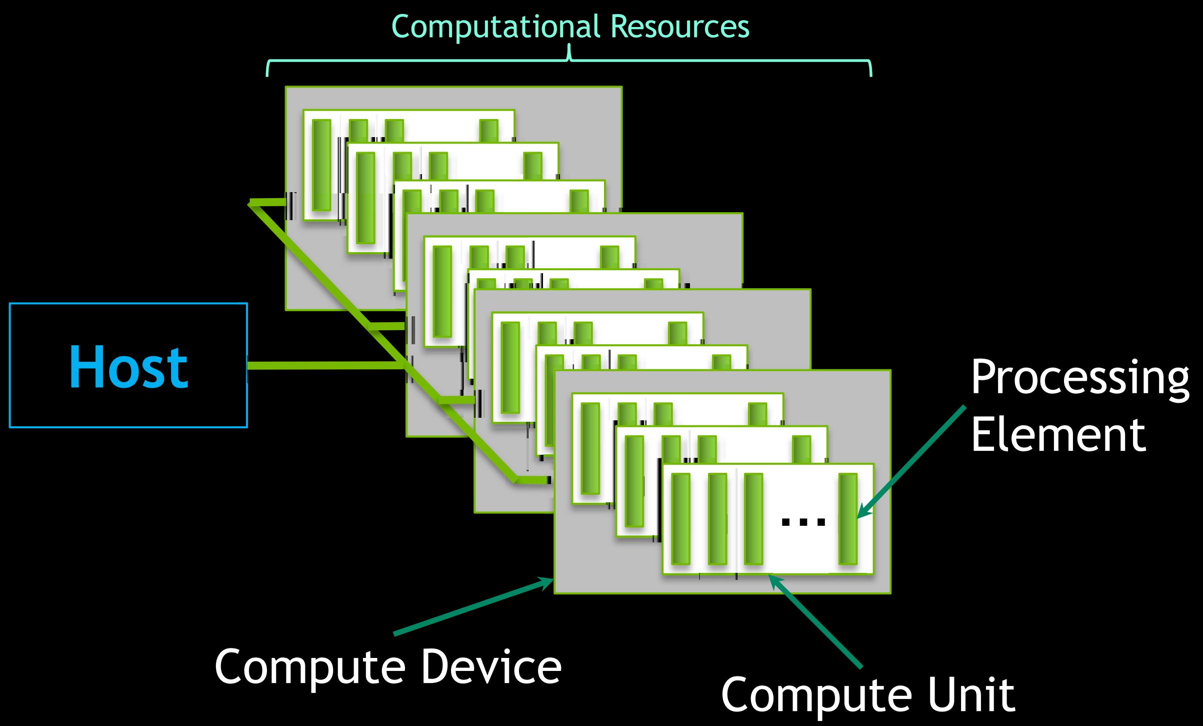
It’s a Heterogeneous World

- A modern platform includes:
  - One or more CPUs
  - One or more GPUs
  - Optional accelerators (e.g., DSPs)

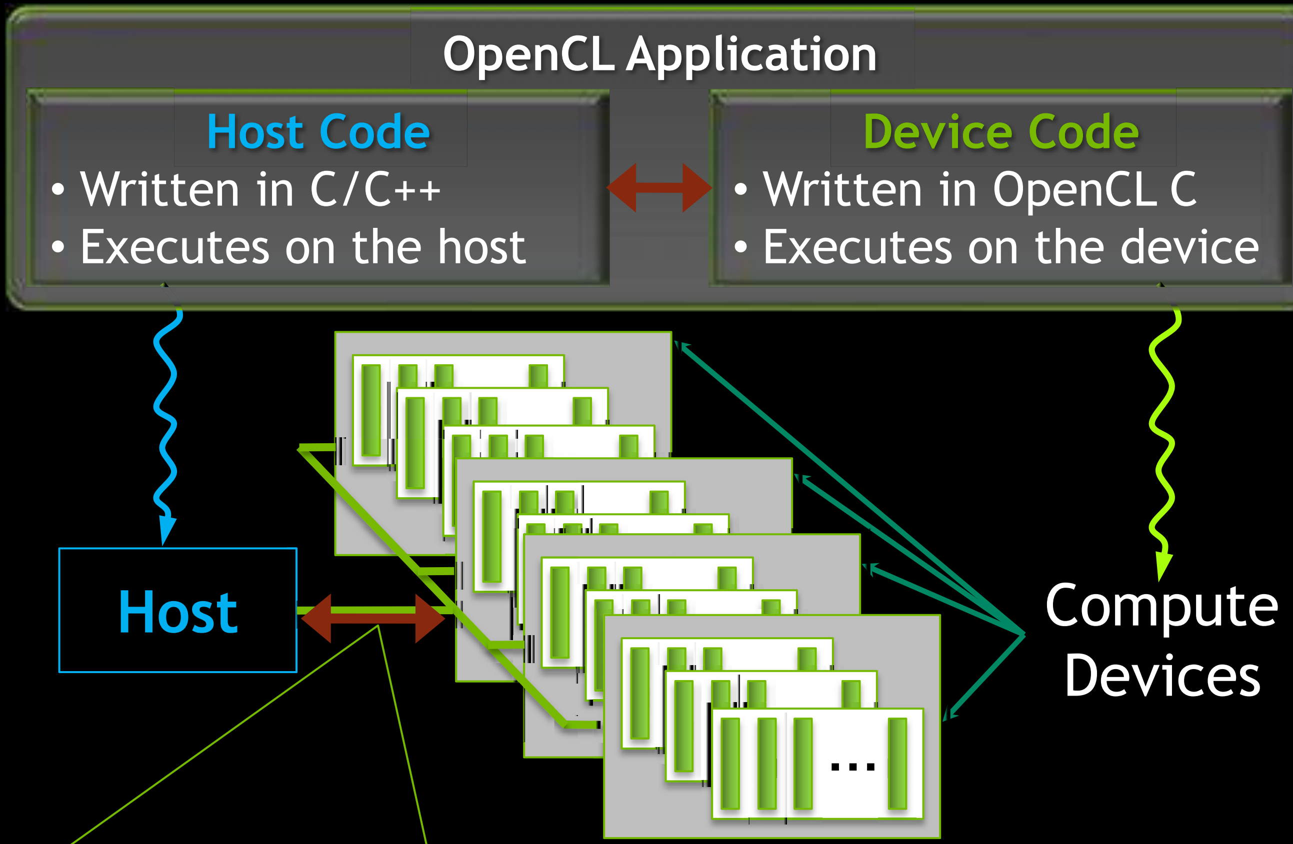


GMCH = graphics memory control hub
ICH = Input/output control hub

OpenCL Platform Model



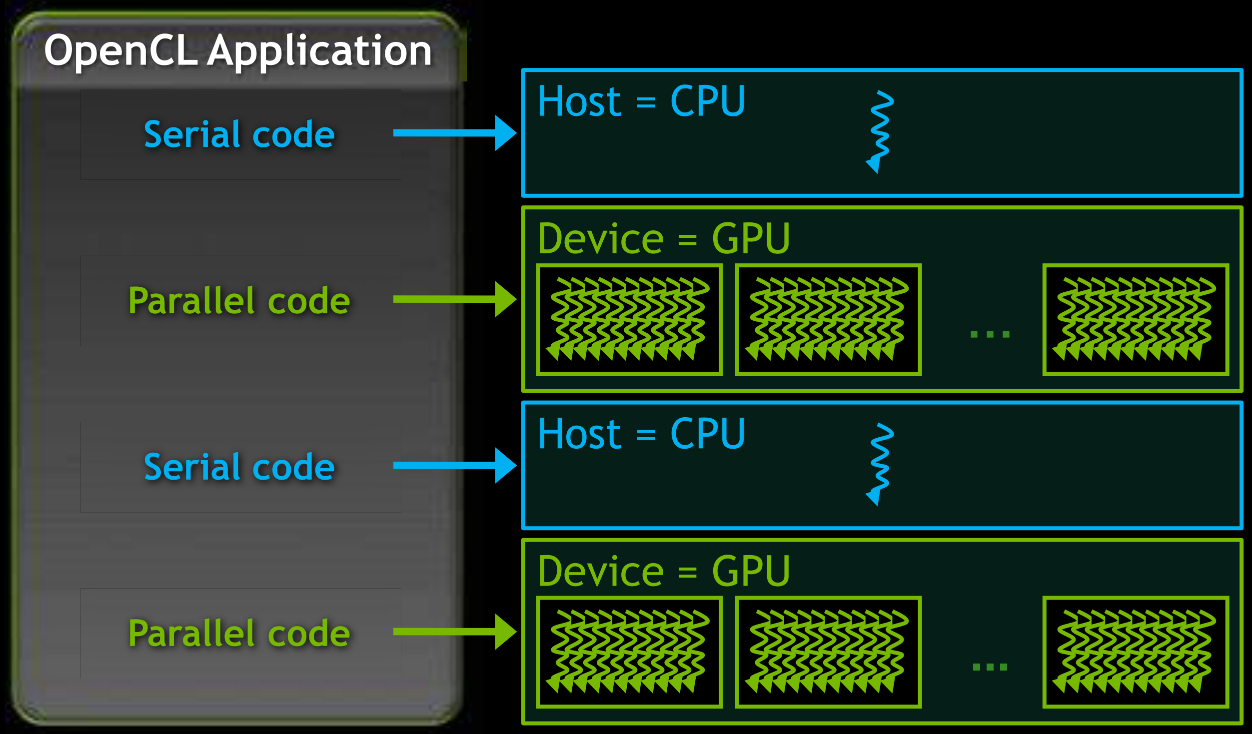
Anatomy of an OpenCL Application



Host code sends commands to the Devices:
… to transfer data between host memory and device memories
… to execute device code

Anatomy of an OpenCL Application

- Serial code executes in a Host (CPU) thread
- Parallel code executes in many Device (GPU) threads across multiple processing elements



OPENCL EXECUTION MODEL

Decompose task into *work-items*

- Define N-dimensional computation domain
- Execute a *kernel* at each point in computation domain

Traditional loop as a function in C

```c
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

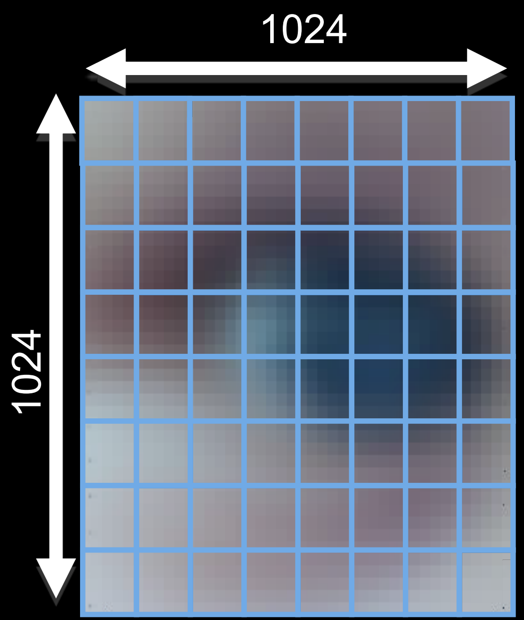
OpenCL C kernel

```c
__kernel void
dp_mul(__global const float *a,
       __global const float *b,
       __global float *c)

{
  int id = get_global_id(0);

  c[id] = a[id] * b[id];
} // execute over n "work items"
```

An N-dimension domain of work-items

Define the "best" N-dimensioned index space for your algorithm

- Kernels are executed across a global domain of *work-items*
- Work-items are grouped into local *work-groups*
  - Global Dimensions: 1024 x 1024 (whole problem space)
  - Local Dimensions: 32 x 32 (work-group … executes together)



OpenCL Execution Model

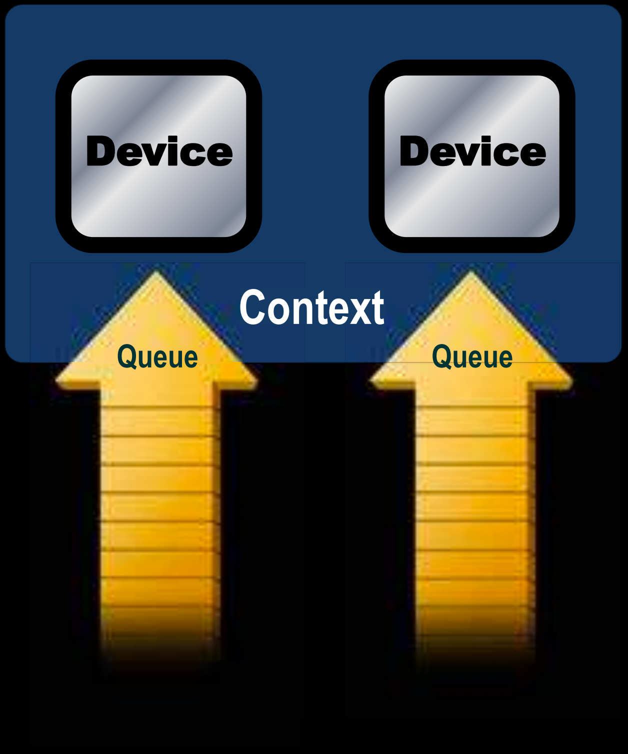
The application runs on a Host which submits work to the Devices

- **Work-item**: the basic unit of work on an OpenCL device
- **Kernel**: the code for a work-item (basically a C function)
- **Program**: Collection of kernels and other functions (analogous to a dynamic library)

OpenCL Execution Model

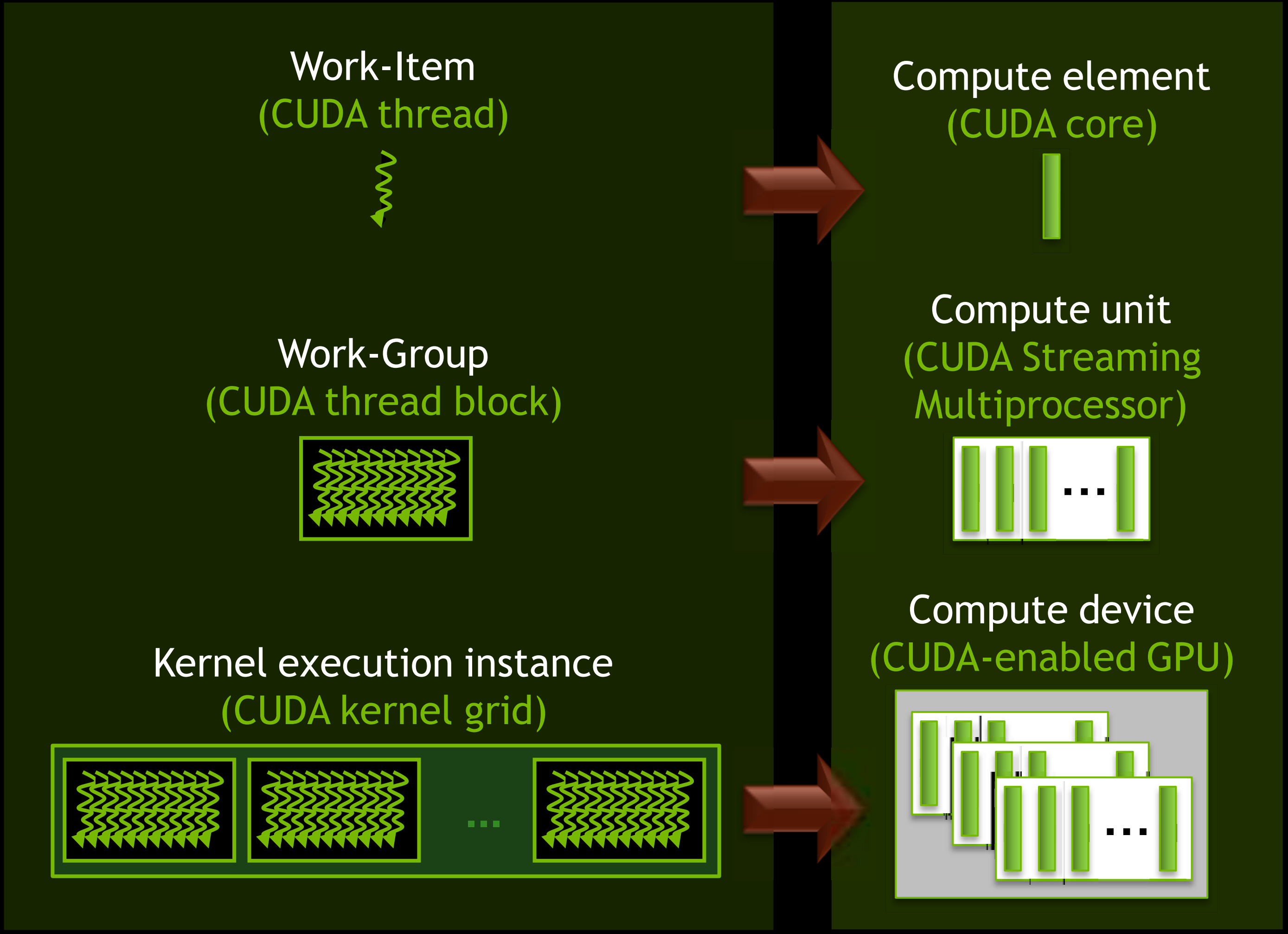
The application runs on a Host which submits work to the Devices

- **Context**: The environment within which work-items execute; includes devices and their memories and command queues
- **Command Queue**: A queue used by the Host application to submit work to a Device (e.g., kernel execution instances)
  - Work is queued in-order, one queue per device
  - Work can be executed in-order *or* out-of-order



MAPPING THE EXECUTION MODEL ONTO THE PLATFORM MODEL

Kernel Execution on Platform Model



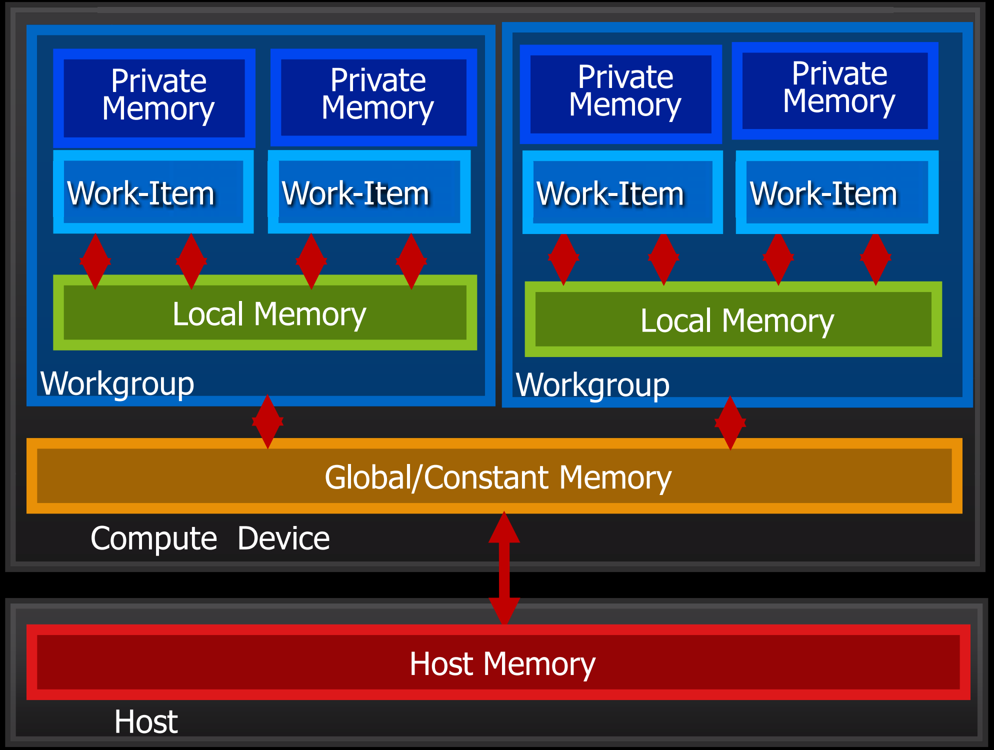
- Each work-item is executed by a compute element

- Each work-group is executed on a compute unit
- Several concurrent work-groups can reside on one compute unit depending on work-group's memory requirements and compute unit's memory resources

- Each kernel is executed on a compute device

OpenCL Memory Model

- **Private Memory**
  - Per work-item
- **Local Memory**
  - Shared within a workgroup
- **Global/Constant Memory**
  - Visible to all workgroups
- **Host Memory**
  - On the CPU



Memory management is Explicit

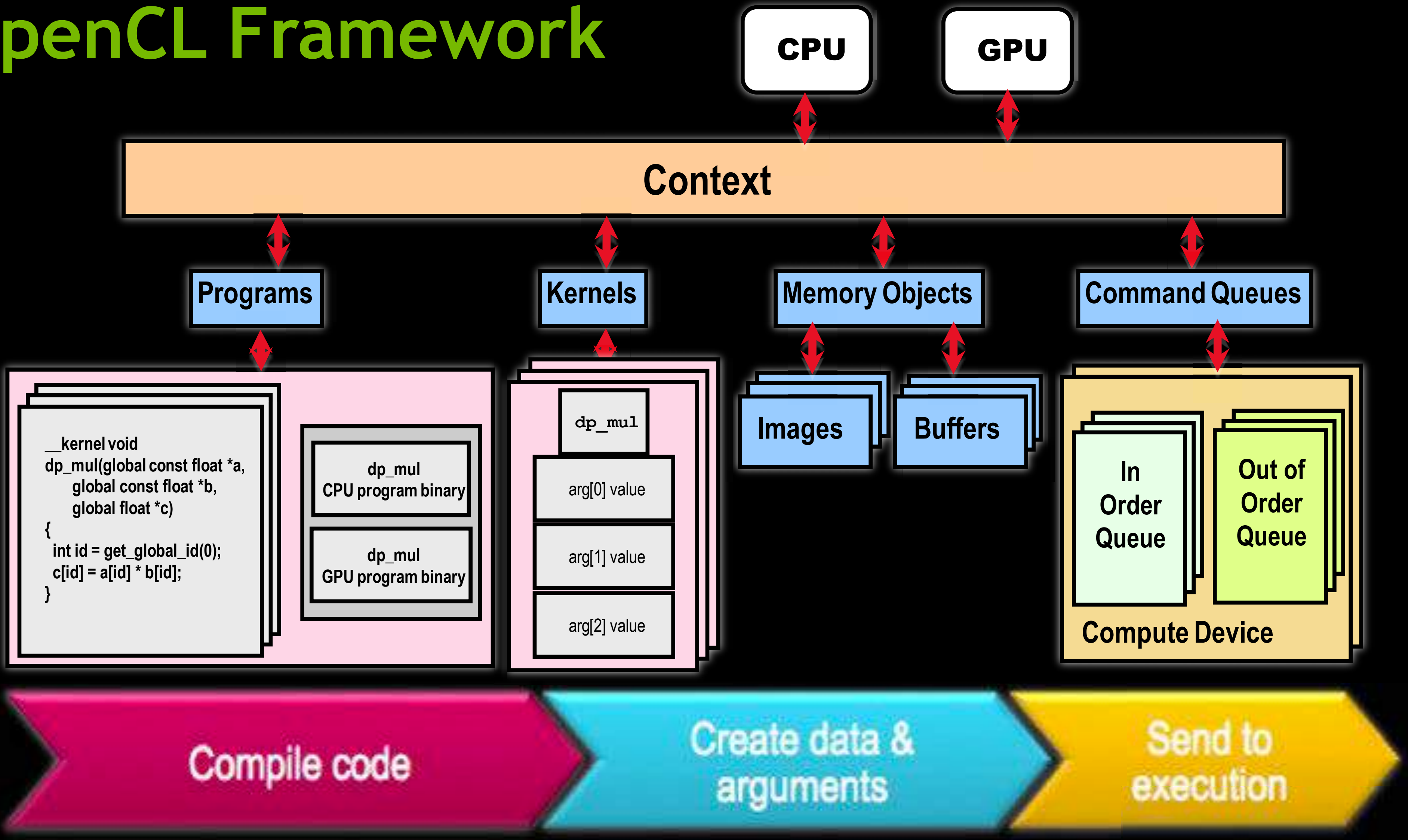
You must move data from host -> global -> local … and back

INTRODUCTION TO OPENCL PROGRAMMING

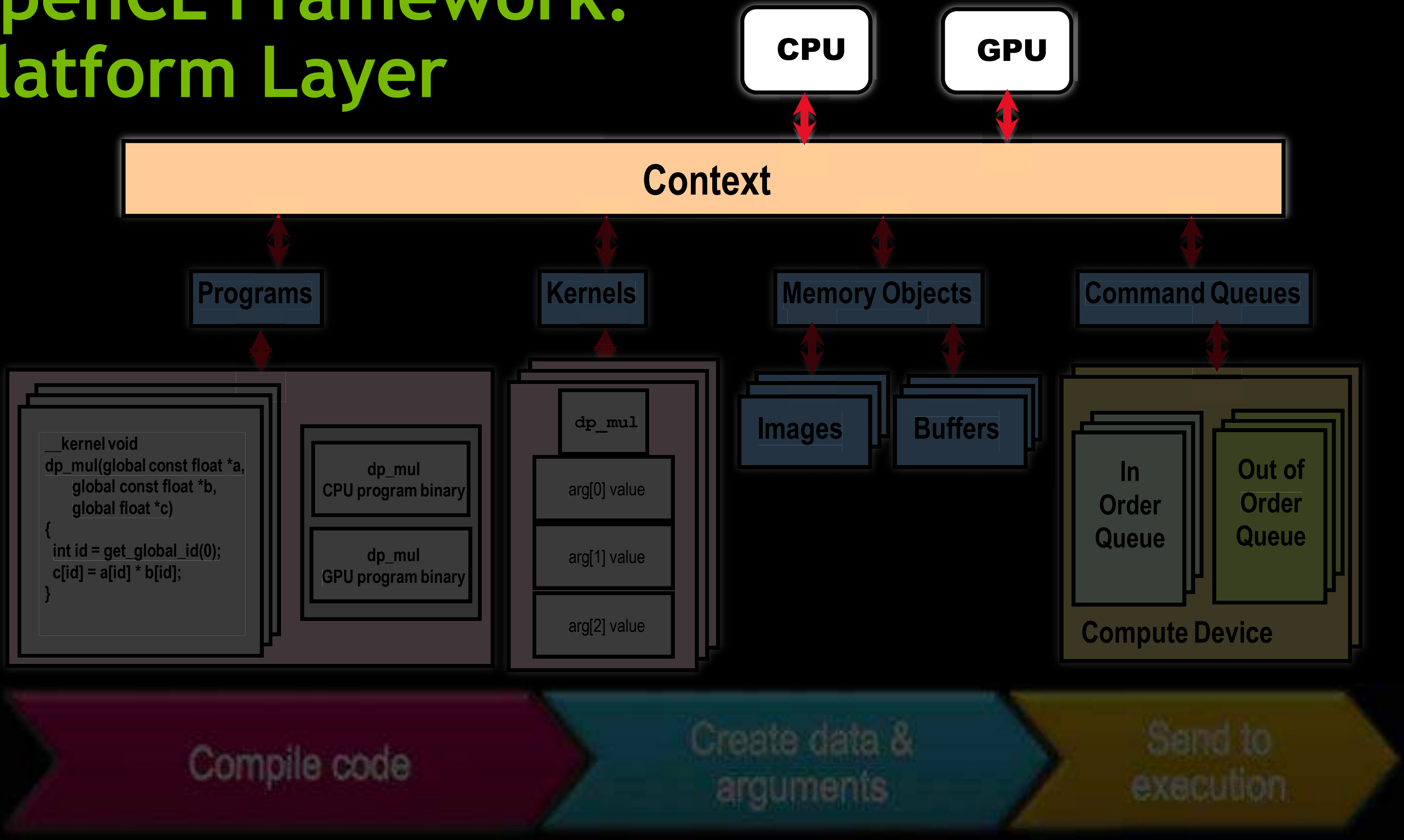
OpenCL Framework

- Platform layer
  - Platform query and context creation
- Compiler for OpenCL C
- Runtime
  - Memory management and command execution within a context

OpenCL Framework



OpenCL Framework: Platform Layer



OpenCL Framework: Platform Layer

- Query platform information
  - clGetPlatformInfo(): profile, version, vendor, extensions
  - clGetDeviceIDs(): list of devices
  - clGetDeviceInfo(): type, capabilities
- Create an OpenCL context for one or more devices

Context =
cl_context

One or more devices
cl_device_id

Memory and device code shared by these devices
cl_mem cl_program

Command queues to send commands to these devices
cl_command_queue

Platform Layer:
Context Creation (simplified)

```
// Get the platform ID
cl_platform_id platform;
clGetPlatformIDs(1, &platform, NULL);

// Get the first GPU device associated with the platform
cl_device_id device;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);

// Create an OpenCL context for the GPU device
cl_context context;
context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
```

Number returned

Context properties

Error callback

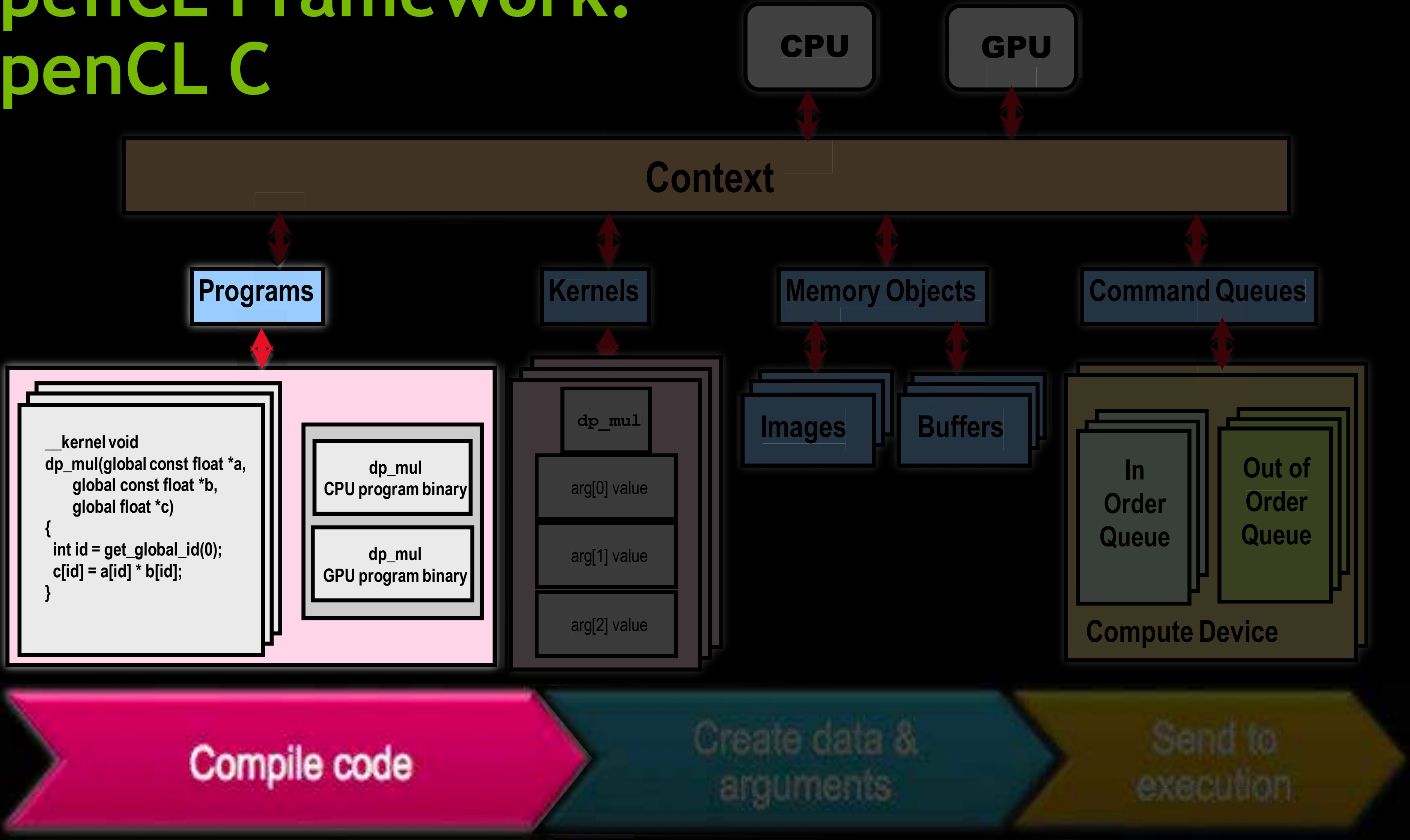
User data

Error code

Platform Layer: Error Handling, Resource Deallocation

- Error handling:
  - All host functions return an error code
  - Context error callback
- Resource deallocation
  - Reference counting API: clRetain*(), clRelease*()
- Both are removed from code samples for clarity
  - Please see SDK samples for complete code

OpenCL Framework: OpenCL C



OpenCL C

- Derived from ISO C99 (with some restrictions)
- Language Features Added
  - Work-items and work-groups
  - Vector types
  - Synchronization
  - Address space qualifiers
- Also includes a large set of built-in functions
  - Image manipulation
  - Work-item manipulation
  - Math functions

OpenCL C Language Restrictions

- Pointers to functions are not allowed
- Pointers to pointers allowed within a kernel, but not as an argument
- Bit-fields are not supported
- Variable-length arrays and structures are not supported
- Recursion is not supported
- Writes to a pointer to a type less than 32 bits are not supported*
- Double types are not supported, but reserved
- 3D Image writes are not supported

Some restrictions are addressed through extensions

OpenCL C Optional Extensions

- Extensions are optional features exposed through OpenCL
- The OpenCL working group has already approved many extensions to the OpenCL specification:
  - Double precision floating-point types (Section 9.3)
  - Built-in functions to support doubles
  - Atomic functions (Section 9.5, 9.6, 9.7)
  - Byte-addressable stores (write to pointers to types < 32-bits) (Section 9.9)
  - 3D Image writes (Section 9.8)
  - Built-in functions to support half types (Section 9.10)

Now core features in OpenCL 1.1

Work-items and work-groups

- A *kernel* is a function executed for each work-item

```
__kernel void square(__global float* input, __global float* output)
{
  int i = get_global_id(0);
  output[i] = input[i] * input[i];
}
```

Function qualifier

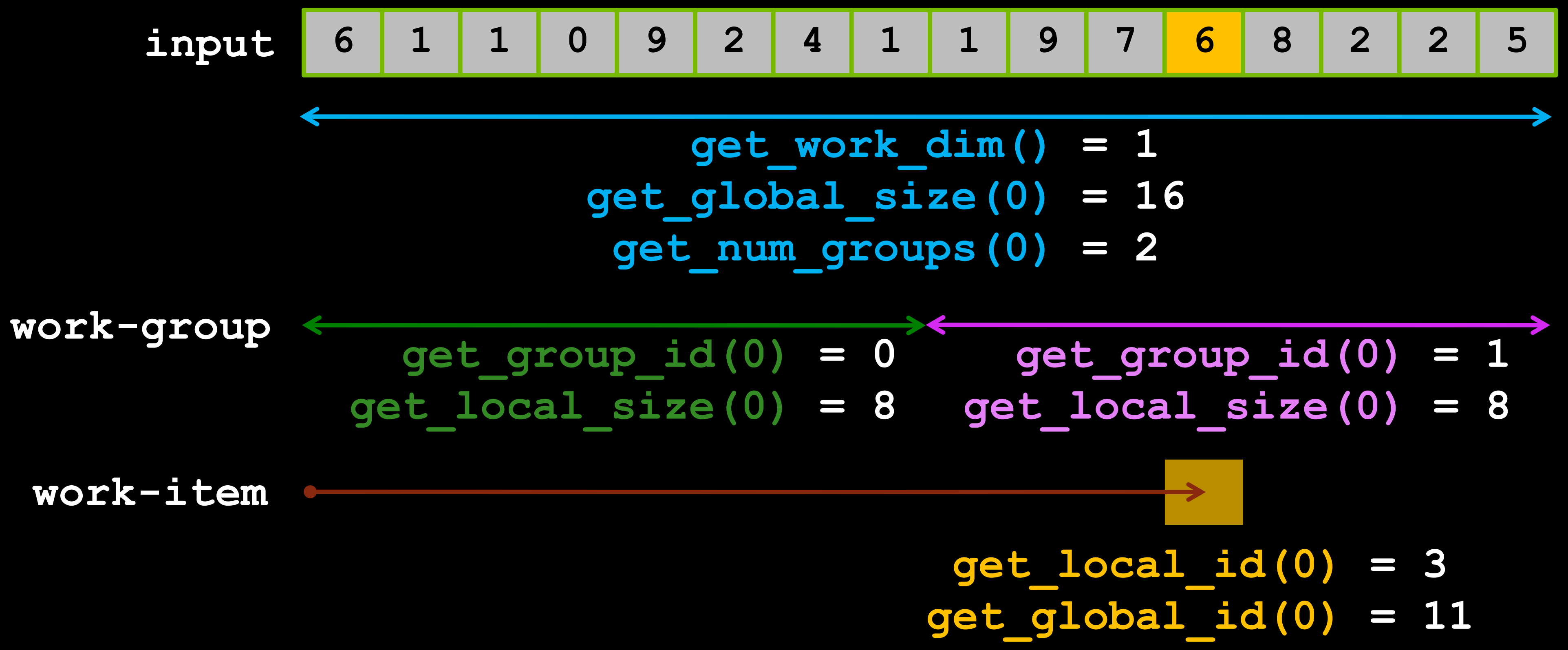
Built-in function

Address space qualifier

`get_global_id(0) = 7`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
input	6	1	1	0	9	2	4	1	1	9	7	6	8	2	2	5
output	36	1	1	0	81	4	16	1	1	81	49	36	64	4	4	25

Work-items and work-groups



OpenCL C Data Types

- Scalar data types
  - char, uchar, short, ushort, int, uint, long, ulong, float
  - bool, intptr_t, ptrdiff_t, size_t, uintptr_t, void, half (storage)
- Image types
  - image2d_t, image3d_t, sampler_t
- Vector data types
  - Vector lengths 2, 3, 4, 8, 16 (char2, ushort4, int8, float16, double2, …)
  - Endian safe
  - Aligned at vector length
  - Vector operations

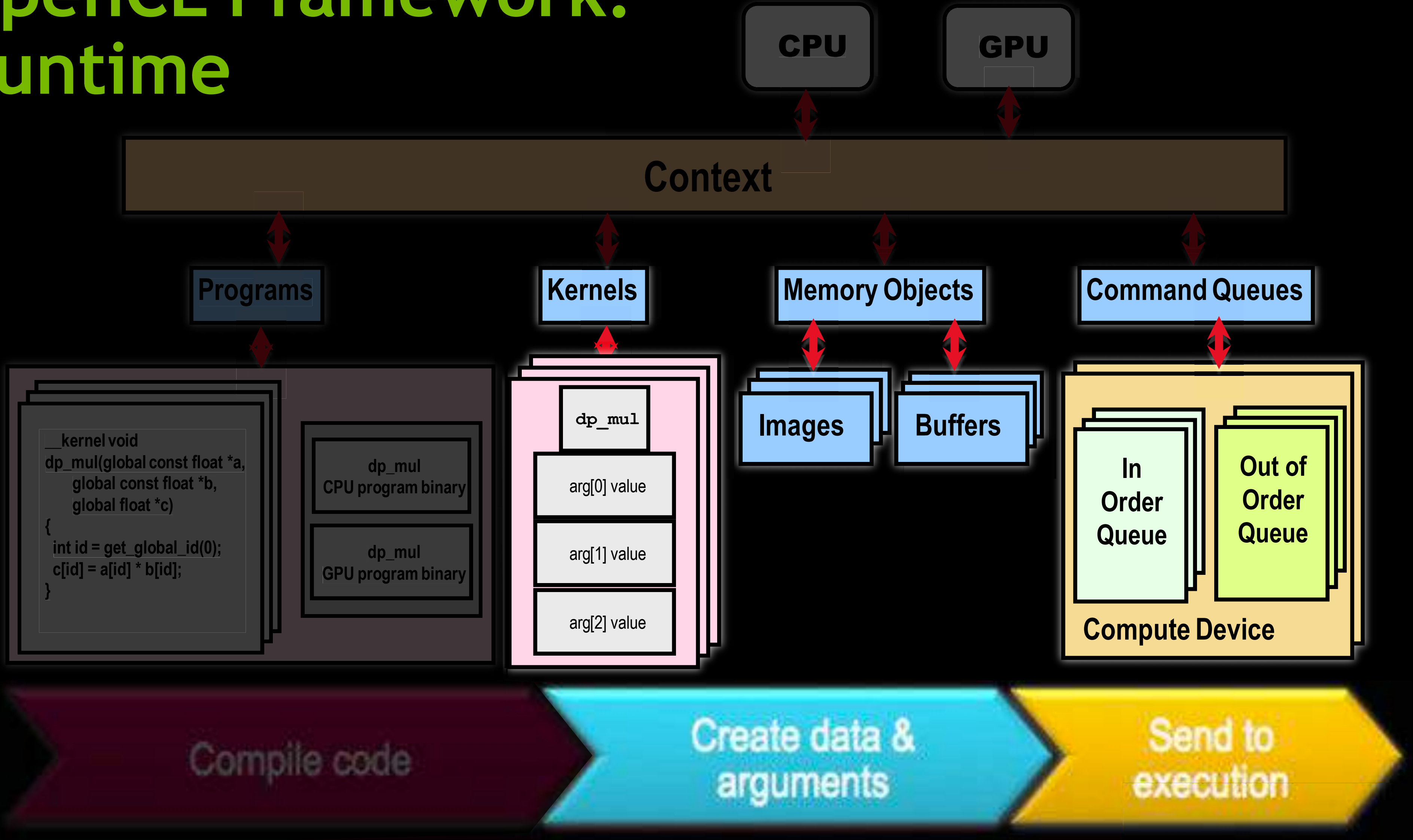
3-vectors new in OpenCL 1.1

double is an optional type in OpenCL

OpenCL C Kernel Example

```
__kernel void dp_mul(__global const float *a,
                     __global const float *b,
                     __global float *c,
                     int N)
{
    int id = get_global_id (0);
    if (id < N)
        c[id] = a[id] * b[id];
}
```

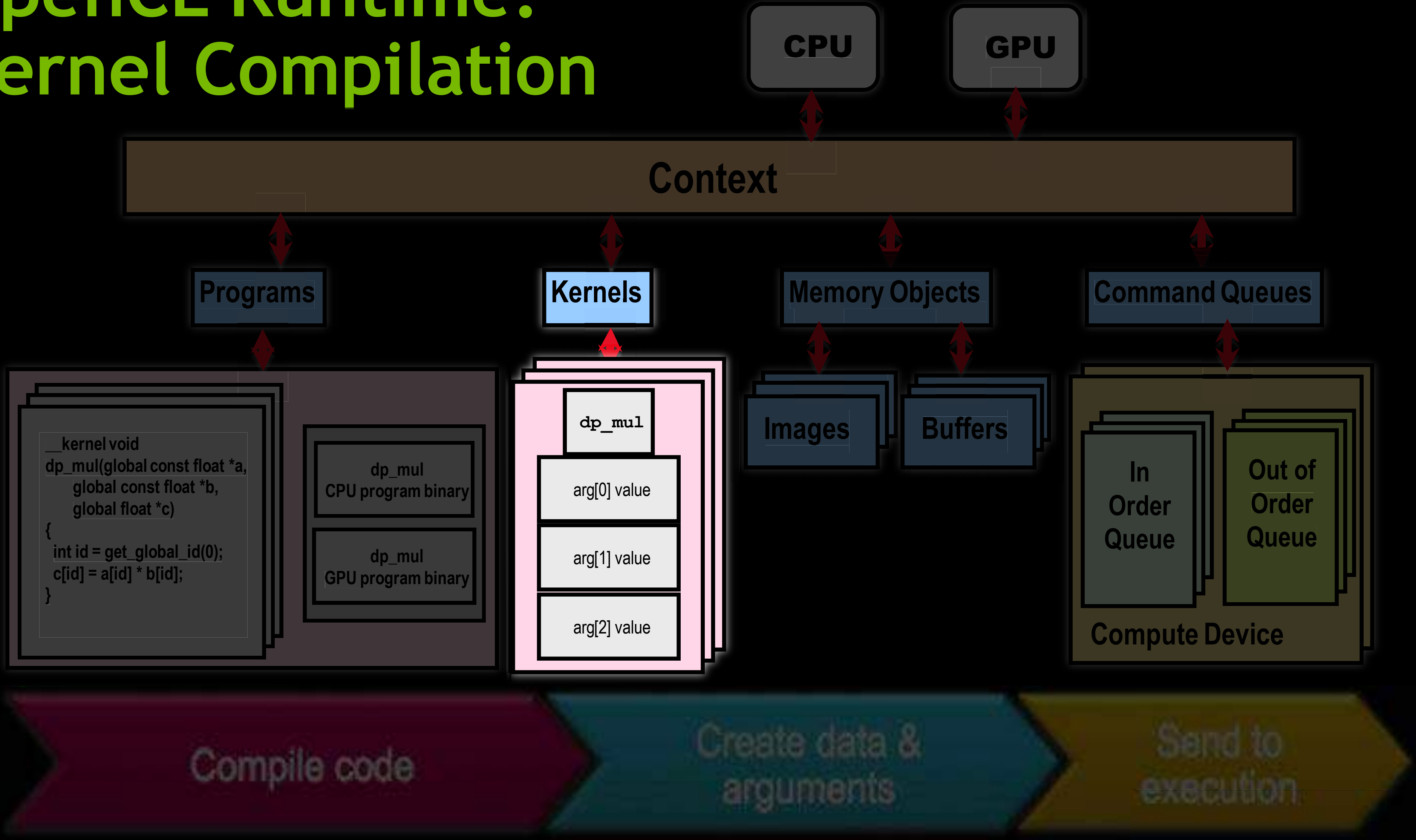
OpenCL Framework: Runtime



OpenCL Framework: Runtime

- Command queues creation and management
- Device memory allocation and management
- Device code compilation and execution
- Event creation and management (synchronization, profiling)

OpenCL Runtime: Kernel Compilation



Kernel Compilation

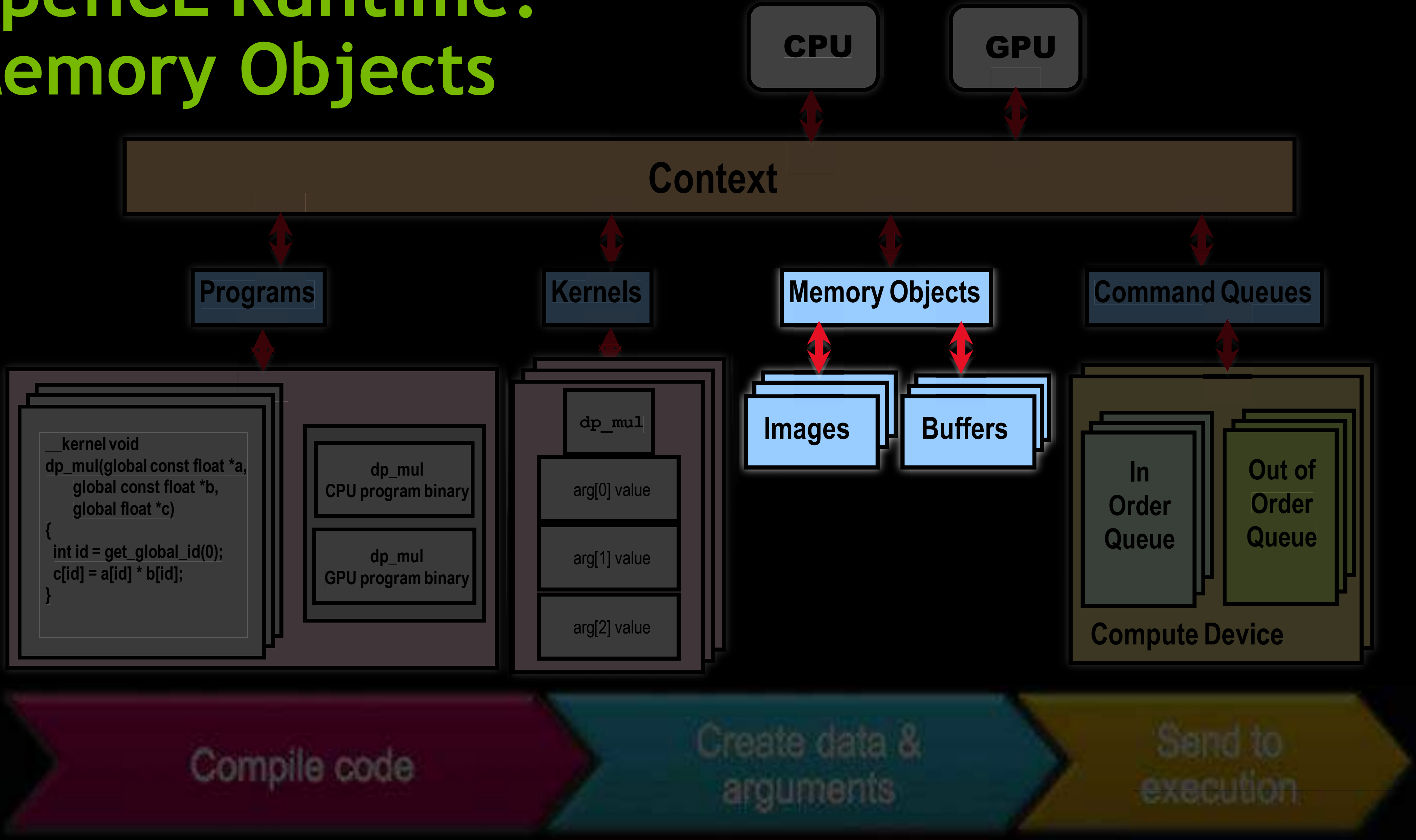
- A cl_program object encapsulates some source code (with potentially several kernel functions) and its last successful build
  - clCreateProgramWithSource() // Create program from source
  - clBuildProgram() // Compile program

- A cl_kernel object encapsulates the values of the kernel's arguments used when the kernel is executed
  - clCreateKernel() // Create kernel from successfully compiled program
  - clSetKernelArg() // Set values of kernel's arguments

Kernel Compilation

```
// Build program object and set up kernel arguments
const char* source = "__kernel void dp_mul(__global const float *a, \n"
                     "                     __global const float *b, \n"
                     "                     __global float *c, \n"
                     "                     int N) \n"
                     "{ \n"
                     "     int id = get_global_id (0); \n"
                     "     if (id < N) \n"
                     "          c[id] = a[id] * b[id]; \n"
                     "} \n";
cl_program program = clCreateProgramWithSource(context, 1, &source, NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
cl_kernel kernel = clCreateKernel(program, “dp_mul", NULL);
clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&d_buffer);
clSetKernelArg(kernel, 1, sizeof(int), (void*)&N);
```

OpenCL Runtime: Memory Objects



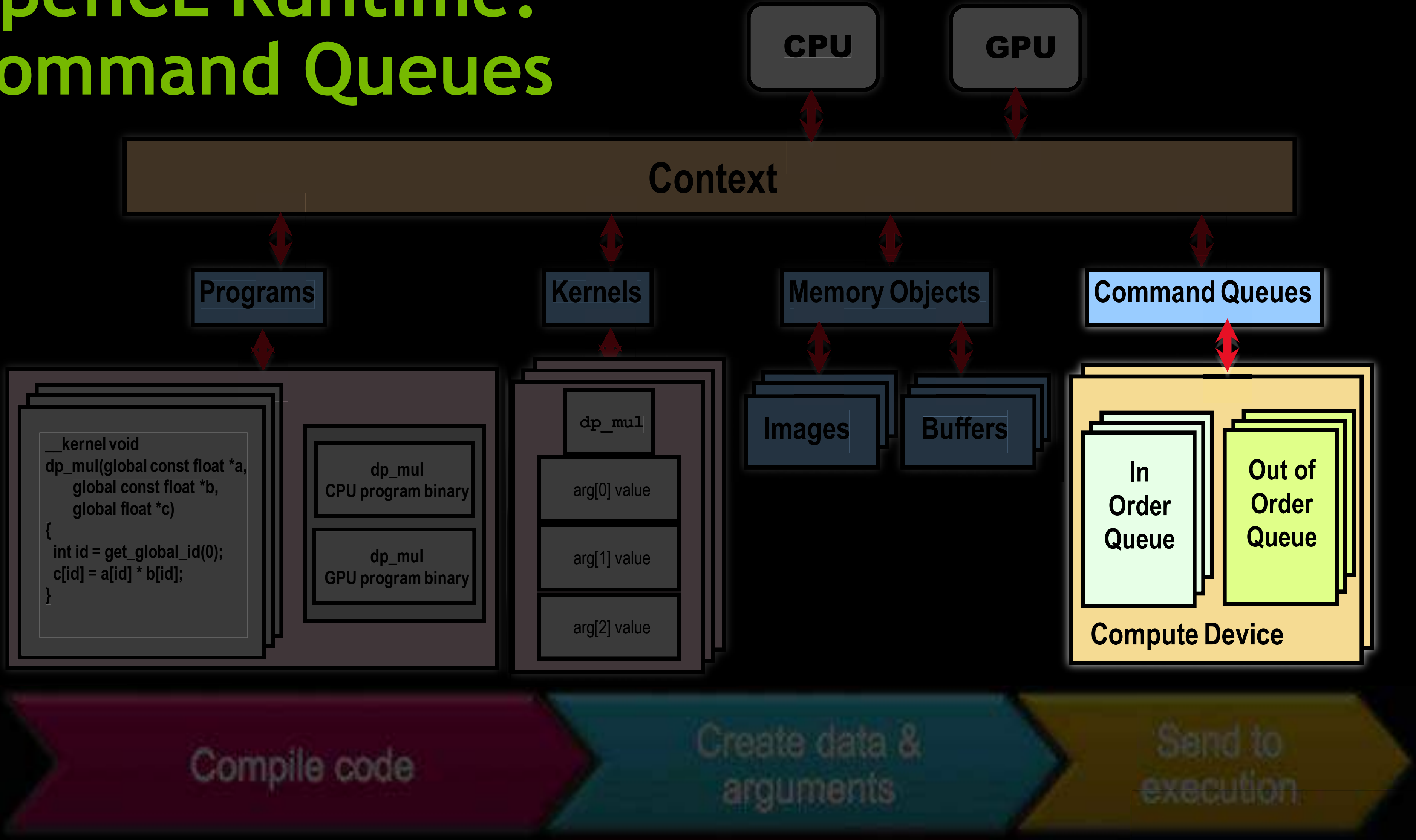
Memory Objects

- Two types of memory objects (cl_mem):
  - **Buffer** objects
  - **Image** objects
- Memory objects can be copied to host memory, from host memory, or to other memory objects
- Regions of a memory object can be accessed from host by mapping them into the host address space

Buffer Object

- One-dimensional array
- Elements are scalars, vectors, or any user-defined structures
- Accessed within device code through pointers

OpenCL Runtime: Command Queues



Commands

- Memory copy or mapping
- Device code execution
- Synchronization point

Command Queue

- Sequence of commands scheduled for execution on a specific device
  - Enqueuing functions: clEnqueue*()
  - Multiple queues can execute on the same device
- Two modes of execution:
  - In-order: Each command in the queue executes only when the preceding command has completed (including memory writes)
  - Out-of-order: No guaranteed order of completion for commands

Error code

```
// Create a command-queue for a specific device
cl_command_queue  cmd_queue = clCreateCommandQueue(context, device_id, 0, NULL);
```

Properties

Data Transfer between Host and Device

```
// Create buffers on host and device
size_t size = 100000 * sizeof(int);
int* h_buffer = (int*)malloc(size);
cl_mem d_buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, size, NULL, NULL);
…
// Write to buffer object from host memory
clEnqueueWriteBuffer(cmd_queue, d_buffer, CL_FALSE, 0, size, h_buffer, 0, NULL, NULL);
…
// Read from buffer object to host memory
clEnqueueReadBuffer(cmd_queue, d_buffer, CL_TRUE, 0, size, h_buffer, 0, NULL, NULL);
```

Blocking? Offset Event synch

Kernel Execution: NDRange

- Host code invokes a kernel over an index space called an *NDRange*
  - NDRange = “N-Dimensional Range” of work-items
  - NDRange can be a 1-, 2-, or 3-dimensional space
  - Work-group dimensionality matches work-item dimensionality

Kernel Invocation

```
// Set number of work-items in a work-group
size_t localWorkSize = 256;
int numWorkGroups = (N + localWorkSize – 1) / localWorkSize; // round up
size_t globalWorkSize = numWorkGroups * localWorkSize; // must be evenly divisible by localWorkSize
clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL, &globalWorkSize, &localWorkSize, 0, NULL, NULL);
```

NDRange