What is OpenCL™?

André Heidekrüger

Sr. System Engineer Graphics, EMAE

Advanced Micro Devices, Inc.

Stream Computing Workshop, 2009

Stockholm, KTH

The future is fusion

Overview

What is OpenCL™?
- **Design Goals**
- **The OpenCL™ Execution Model**

What is OpenCL™? (continued)
- The OpenCL™ Platform and Memory Models

Resource Setup
- Setup and Resource Allocation

Kernel Execution
- Execution and Synchronization

Programming with OpenCL™ C
- Language Features
- Built-in Functions

Welcome to OpenCL™

With OpenCL™ you can…

- Leverage CPUs, GPUs, other processors such as Cell/B.E. processor and DSPs to accelerate parallel computation
- Get dramatic speedups for computationally intensive applications
- Write accelerated portable code across different devices and architectures

With AMD's OpenCL™ you can…

- Leverage AMD's CPUs,and AMD's GPUs, to accelerate parallel computation

OpenCL™ Execution Model

Kernel

- Basic unit of executable code - similar to a C function
- Data-parallel or task-parallel

Program

- Collection of kernels and other functions
- Analogous to a dynamic library

Applications queue kernel execution instances

- Queued in-order
- Executed in-order or out-of-order

Expressing Data-Parallelism in OpenCL™

Define N-dimensional computation domain (N = 1, 2 or 3)

- Each independent element of execution in N-D domain is called a work-item
- The N-D domain defines the total number of work-items that execute in parallel

E.g., process a 1024 x 1024 image: Global problem dimensions: 1024 x 1024 = 1 kernel execution per pixel: 1,048,576 total executions

Scalar

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

Data-Parallel

```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *result)
{
  int id = get_global_id(0);
  result[id] = a[id] * b[id];
}
// execute dp_mul over "n" work-items
```

Expressing Data-Parallelism in OpenCL™

Kernels executed across a global domain of work-items

- Global dimensions define the range of computation
- One work-item per computation, executed in parallel

Work-items are grouped in local workgroups

- Local dimensions define the size of the workgroups
- Executed together on one device
- Share local memory and synchronization

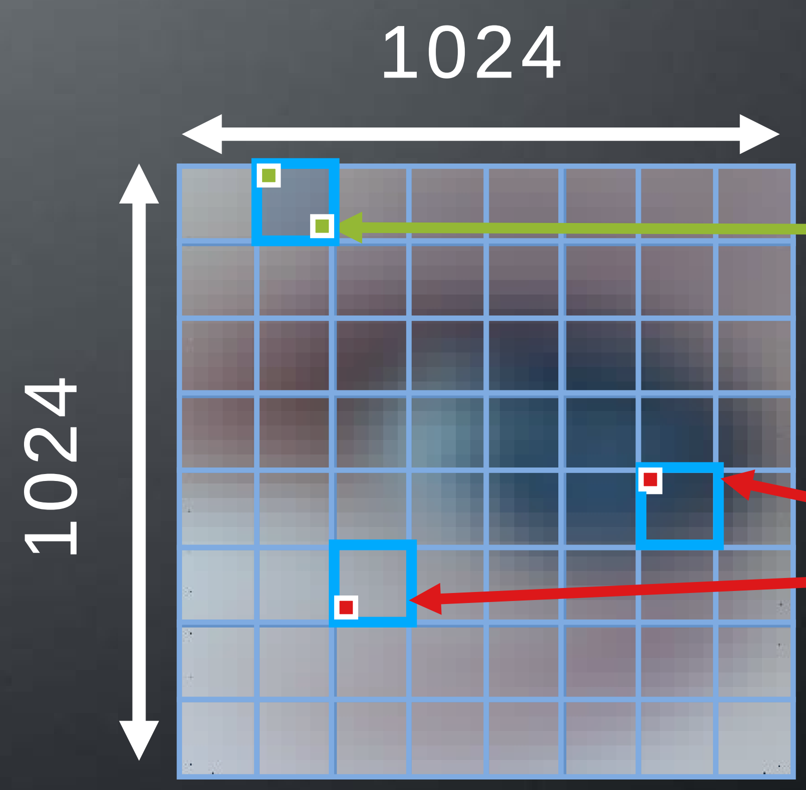
Caveats

- Global work-items must be independent: No global synchronization
- Synchronization can be done within a workgroup

Global and Local Dimensions

Global Dimensions: 1024 x 1024 (whole problem space)

Local Dimensions: 128 x 128 (executed together)



Synchronization between *work-items* possible only **within** *workgroups*: barriers and memory fences

Can **not** synchronize outside of a *workgroup*

Example Problem Dimensions

1D: 1 million elements in an array:

```
global_dim[3] = {1000000,1,1};
```

2D: 1920 x 1200 HD video frame, 2.3M pixels:

```
global_dim[3] = {1920, 1200, 1};
```

3D: 256 x 256 x 256 volume, 16.7M voxels:

```
global_dim[3] = {256, 256, 256};
```

Choose the dimensions that are “best” for your algorithm

- Maps well
- Performs well

Synchronization Within Work-Items

No global synchronization, only within workgroups

The work-items in each workgroup can:

- Use barriers to synchronize execution
- Use memory fences to synchronize memory accesses

You must adapt your algorithm to only require synchronization

- Within workgroups (e.g., reduction)
- Between kernels (e.g., multi-pass)

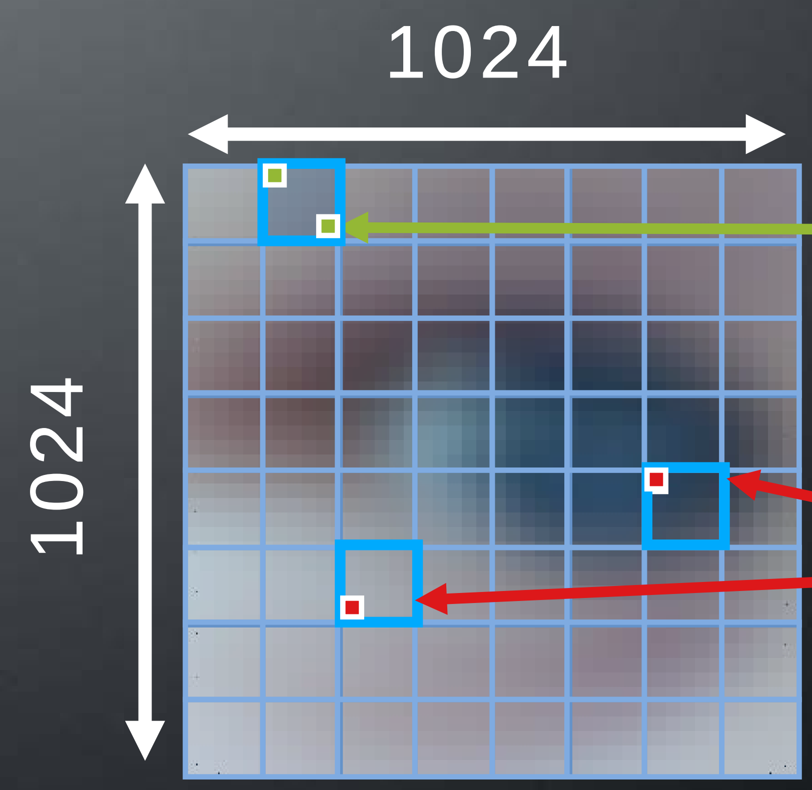
Part 2: What is OpenCL™? (continued)

The OpenCL™ Platform and Memory Models

Global and Local Dimensions

Global Dimensions: 1024 x 1024 (whole problem space)

Local Dimensions: 128 x 128 (executed together)



Synchronization between *work-items* possible only **within** *workgroups*: barriers and memory fences

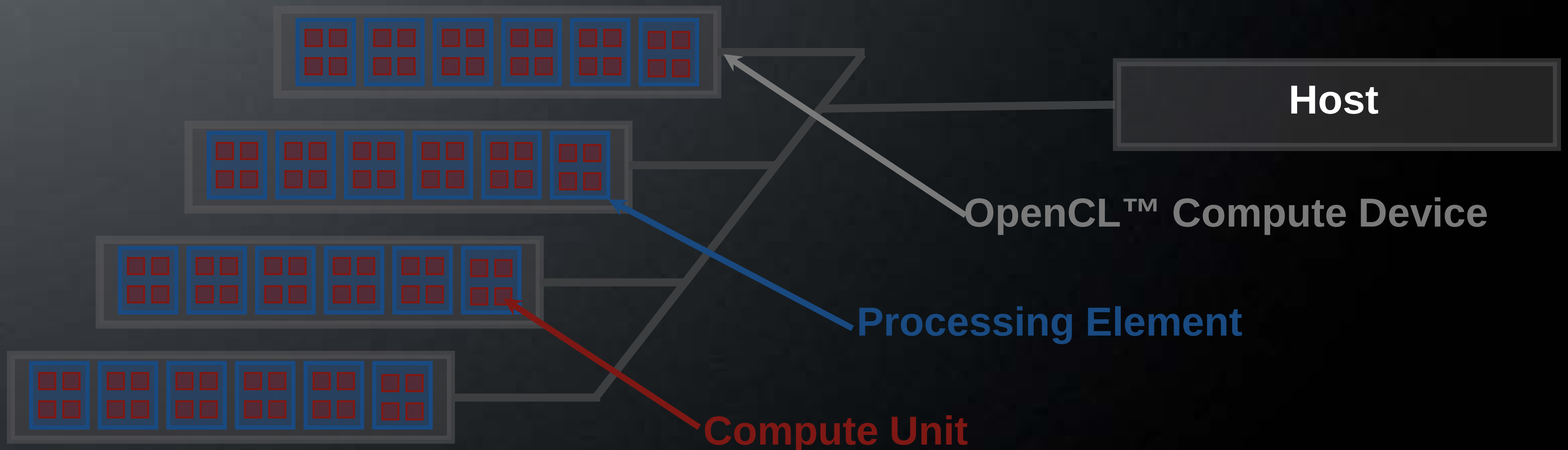
Can **not** synchronize outside of a *workgroup*

OpenCL™ Platform Model

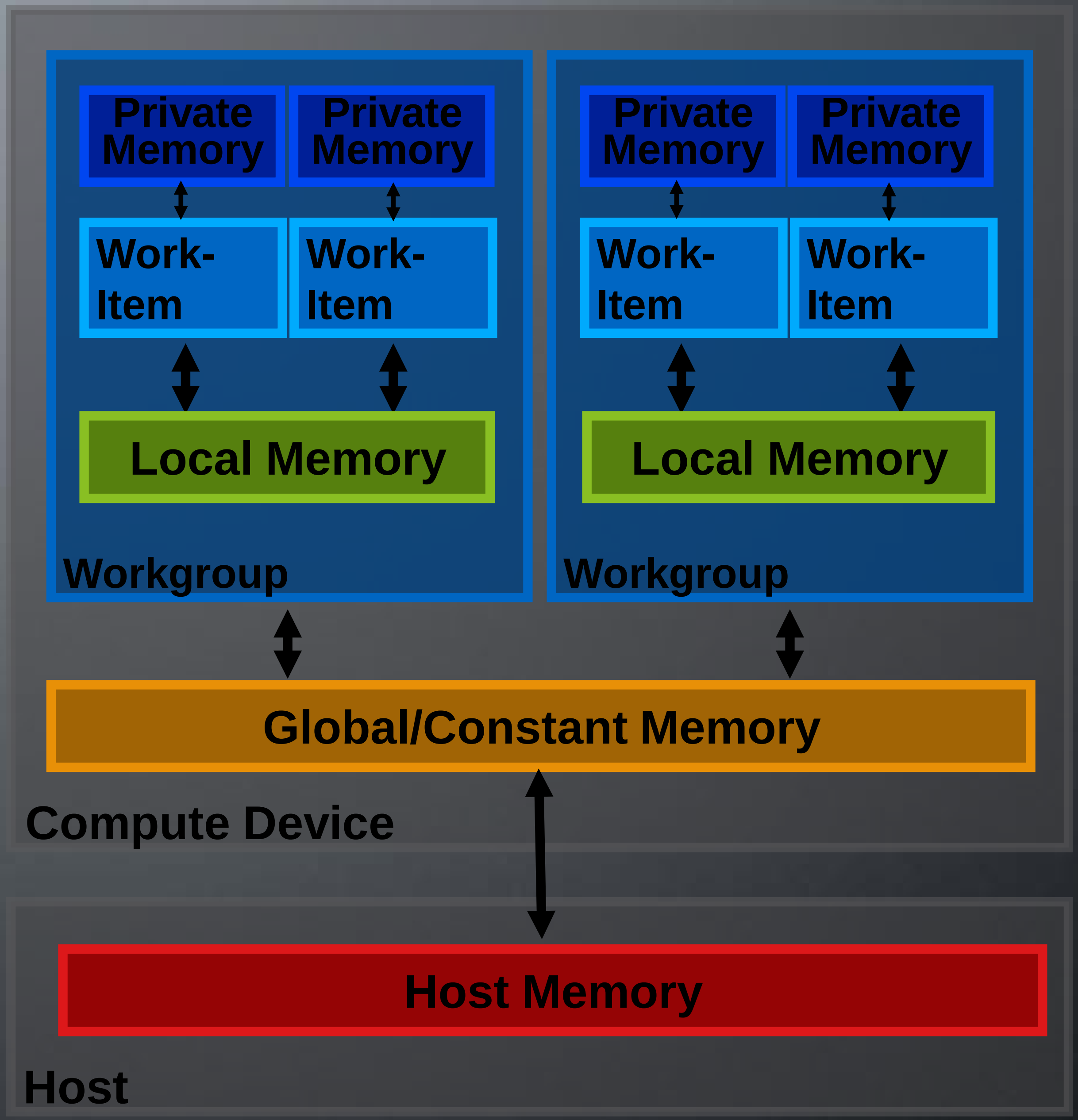
A host connected to one or more OpenCL™ devices

OpenCL™ devices:

- A collection of one or more compute units (cores)
- A compute unit
  - Composed of one or more processing elements
  - Processing elements execute code as SIMD or SPMD



OpenCL™ Memory Model



- Private Memory: Per *work-item*
- Local Memory: Shared within a *workgroup*
- Local Global/Constant Memory: Not synchronized
- Host Memory: On the CPU

Memory management is explicit

You must move data from host to global to local and back

AMD The future is fusion

OpenCL™ Objects

Setup

- Devices—GPU, CPU, Cell/B.E.
- Contexts—Collection of devices
- Queues—Submit work to the device

Memory

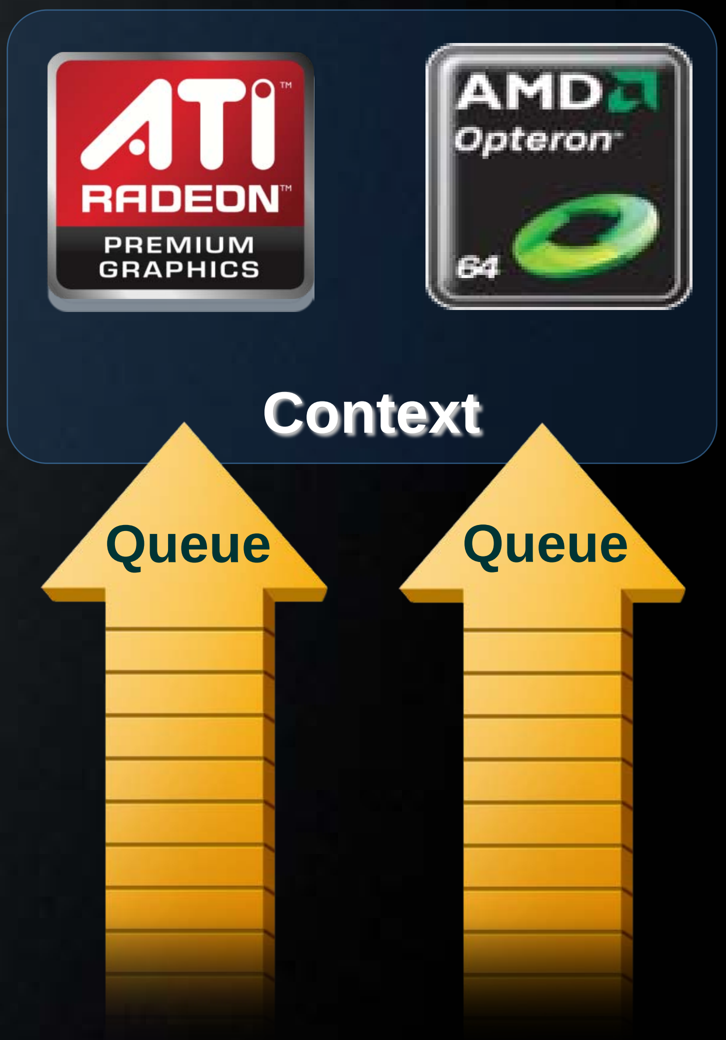
- Buffers—Blocks of memory
- Images—2D or 3D formatted images

Execution

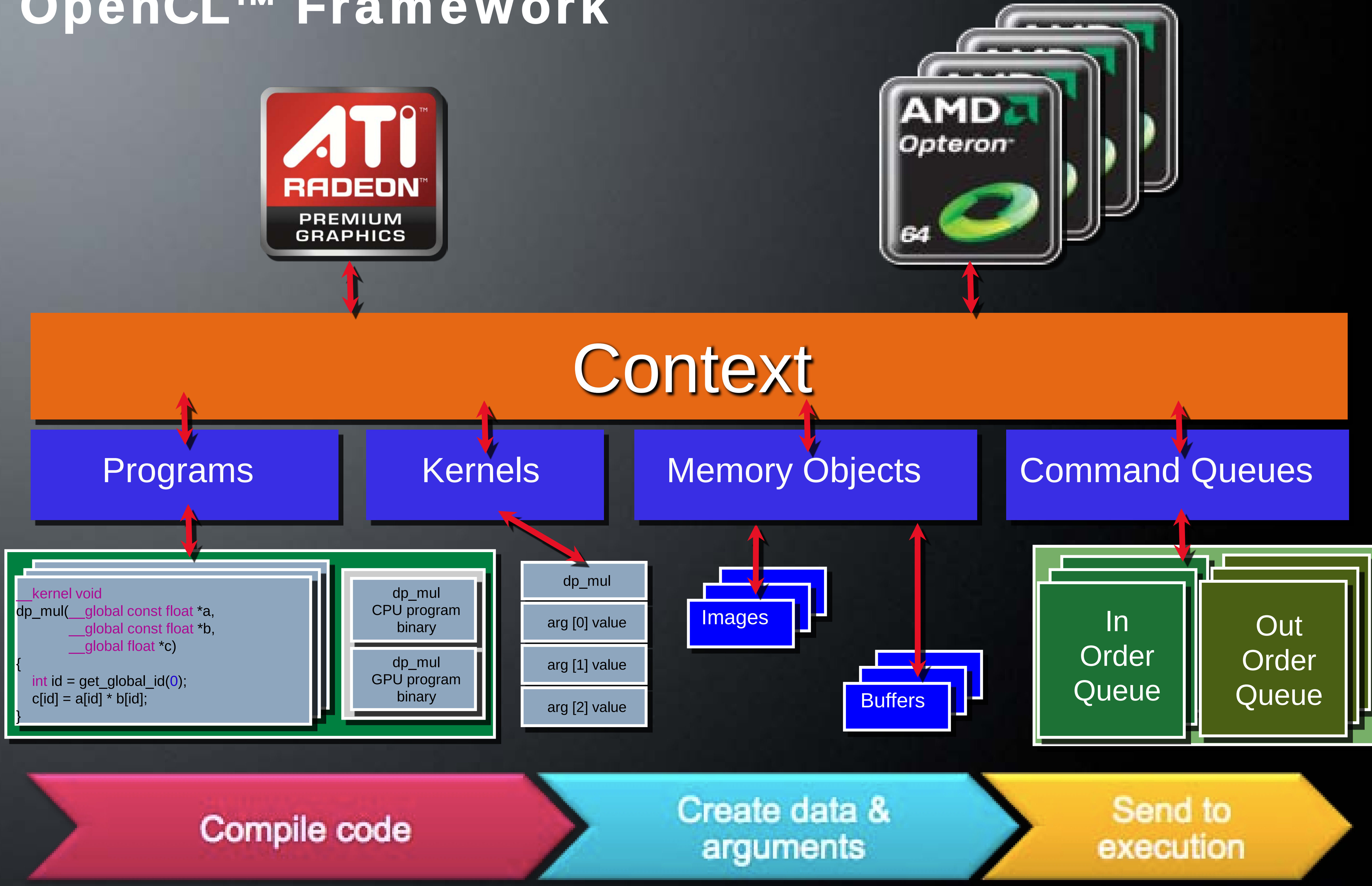
- Programs—Collections of kernels
- Kernels—Argument/execution instances

Synchronization/profiling

- Events



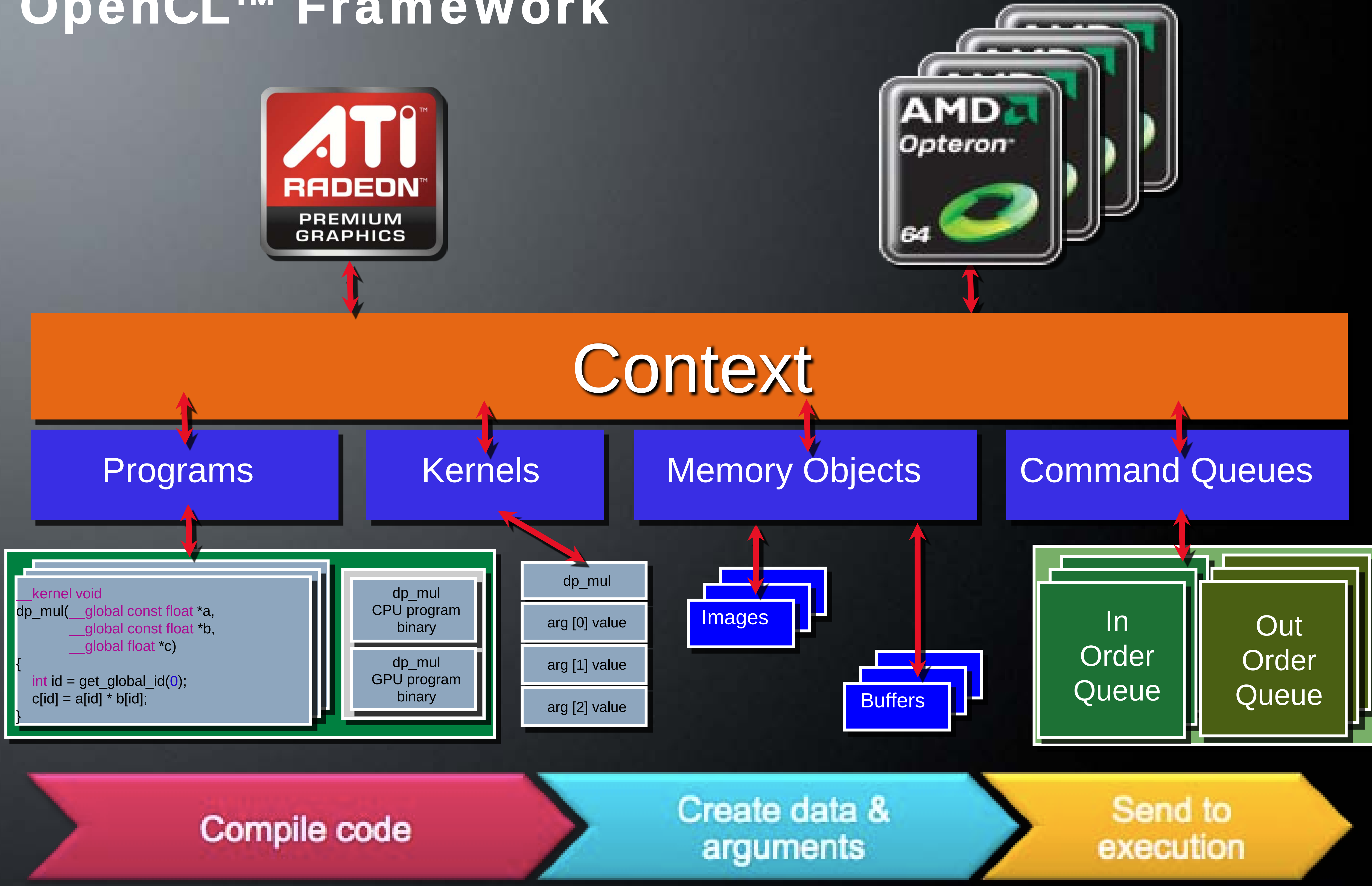
OpenCL™ Framework



Part 3: Resource Setup

- Setup and Resource Allocation

OpenCL™ Framework

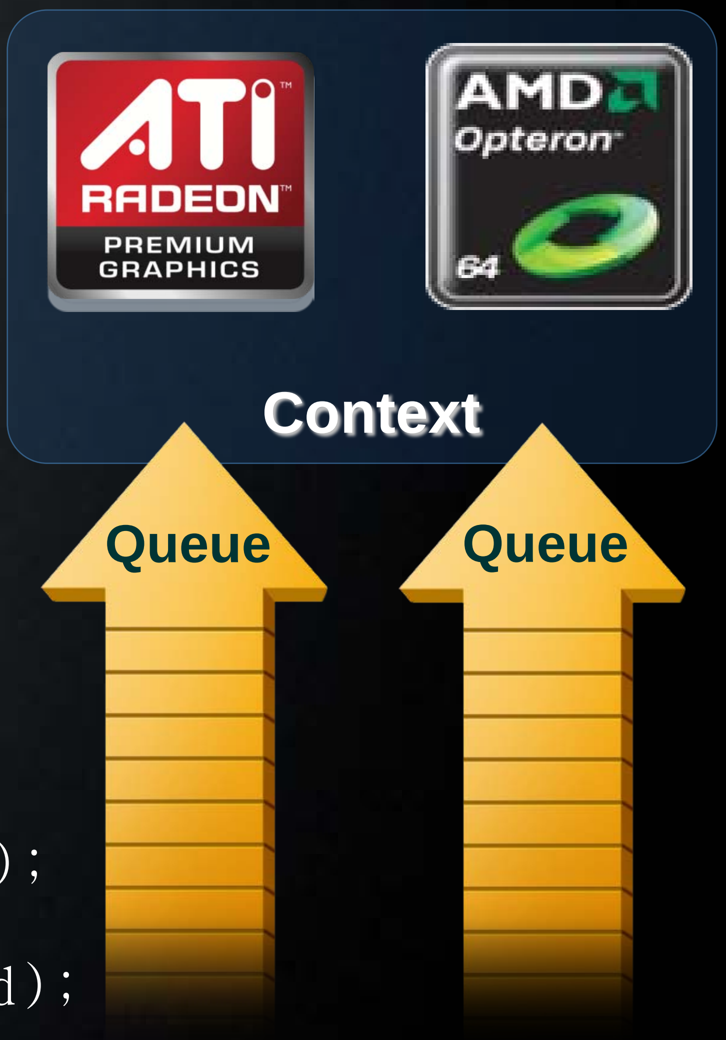


Setup

Get the device(s)

Create a context

Create command queue(s)




```
cl_uint num_devices_returned;
cl_device_id devices[2];
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_GPU, 1,
                     &devices[0], num_devices_returned);
err = clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1,
                     &devices[1], &num_devices_returned);


cl_context context;
context = clCreateContext(0, 2, devices, NULL, NULL, &err);

cl_command_queue queue_gpu, queue_cpu;
queue_gpu = clCreateCommandQueue(context, devices[0], 0, &err);
queue_cpu = clCreateCommandQueue(context, devices[1], 0, &err);
```

Setup: Notes

Devices

- Multiple cores on CPU or GPU together are a single device
- OpenCL™ executes kernels across all cores in a data-parallel manner

Contexts

- Enable sharing of memory between devices
- To share between devices, both devices must be in the same context

Queues

- All work submitted through queues
- Each device must have a queue

Choosing Devices

A system may have several devices—which is best?

The “best” device is algorithm- and hardware-dependent

Query device info with: `clGetDeviceInfo(device, param_name, *value)`

- Number of compute units `CL_DEVICE_MAX_COMPUTE_UNITS`
- Clock frequency `CL_DEVICE_MAX_CLOCK_FREQUENCY`
- Memory size `CL_DEVICE_GLOBAL_MEM_SIZE`
- Extensions (double precision, atomics, etc.)

Pick the best device for your algorithm

- Sometimes CPU is better, other times GPU is better

Memory Resources

Buffers

- Simple chunks of memory
- Kernels can access however they like (array, pointers, structs)
- Kernels can read and write buffers

Images

- Opaque 2D or 3D formatted data structures
- Kernels access only via read_image() and write_image()
- Each image can be read or written in a kernel, but not both

Image Formats and Samplers

Formats

- Channel orders: `CL_A, CL_RG, CL_RGB, CL_RGBA`, etc.
- Channel data type: `CL_UNORM_INT8, CL_FLOAT`, etc.
- `clGetSupportedImageFormats()` returns supported formats

Samplers (for reading images)

- Filter mode: linear or nearest
- Addressing: clamp, clamp-to-edge, repeat, or none
- Normalized: true or false

Benefit from image access hardware on GPUs

Allocating Images and Buffers

```
cl_image_format format;
format.image_channel_data_type = CL_FLOAT;
format.image_channel_order = CL_RGBA;

 cl_mem input_image;
 input_image  = clCreateImage2D(context, CL_MEM_READ_ONLY, &format,
                              image_width, image_height, 0, NULL, &err);
cl_mem output_image;
output_image  = clCreateImage2D(context, CL_MEM_WRITE_ONLY, &format,
                              image_width, image_height, 0, NULL, &err);


cl_mem input_buffer;
input_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY,
                              sizeof(cl_float)*4*image_width*image_height, NULL, &err);
cl_mem output_buffer;
output_buffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                              sizeof(cl_float)*4*image_width*image_height, NULL, &err);
```

Reading and Writing Memory Object Data

Explicit commands to access memory object data

- Read from a region in memory object to host memory
  - `clEnqueueReadBuffer(queue, object, blocking, offset, size, *ptr, ...)`
- Write to a region in memory object from host memory
  - `clEnqueueWriteBuffer(queue, object, blocking, offset, size, *ptr, ...)`
- Map a region in memory object to host address space
  - `clEnqueueMapBuffer(queue, object, blocking, flags, offset, size, ...)`
- Copy regions of memory objects
  - `clEnqueueCopyBuffer(queue, srcobj, dstobj, src_offset, dst_offset, ...)`

Operate synchronously (`blocking = CL_TRUE`) or asynchronously

Introduction to OpenCL™: part 4

- Execution and Synchronization

Program and Kernel Objects

Program objects encapsulate

- A program source or binary
- List of devices and latest successfully built executable for each device
- A list of kernel objects

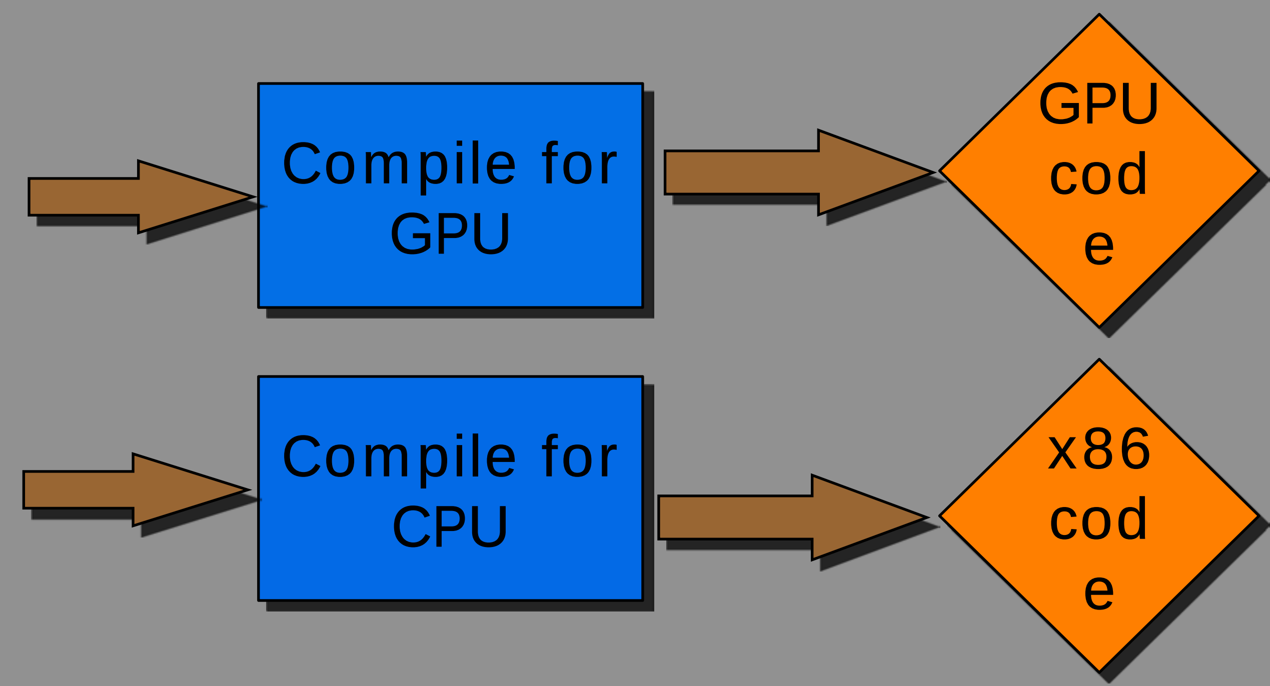
Kernel objects encapsulate

- A specific kernel function in a program
  - Declared with the kernel qualifier
- Argument values
- Kernel objects can only be created after the program executable has been built

Program

Kernel Code

```
kernel void
horizontal_reflect(read_only image2d_t src,
                   write_only image2d_t dst)
{
  int x = get_global_id(0);  // x-coord  int y = get_global_id(1);  // y-coord
int width = get_image_width(src);  float4 src_val = read_imagef(src,
sampler,
                   (int2)(width-1-x, y));  write_imagef(dst, (int2)(x, y), src_val);
}
```



Programs build executable code for multiple devices

Execute the same code on different devices

Compiling Kernels

Create a program

- Input: String (source code) or precompiled binary
- Analogous to a dynamic library: A collection of kernels

Compile the program

- Specify the devices for which kernels should be compiled
- Pass in compiler flags
- Check for compilation/build errors

Create the kernels

- Returns a kernel object used to hold arguments for a given execution

Creating a Program

File: kernels.cl

```
// ---------------------------------
// Images Kernel
// ---------------------------------
kernel average_images(read_only image2d_t input, write_only image2d_t output)
{
    sampler_t sampler = CLK_ADDRESS_CLAMP | CLK_FILTER_NEAREST | CLK_NORMALIZED_COORDS_FALSE;
    int x = get_global_id(0);
    int y = get_global_id(1);
    float4 sum = (float4)0.0f;

    int2 pixel;
    for (pixel.x=x-SIZE; pixel.x<=x+SIZE; pixel.x++)
        for (pixel.y=y-SIZE; pixel.y<=y+SIZE; pixel.y++)
            sum += read_imagef(input, sampler, pixel);

    write_imagef(output, (int2)(x, y), sum/TOTAL);
};
```

```
cl_program program;
program = clCreateProgramWithSource(context, 1, &source, NULL, &err);
```

Compiling and Creating a Kernel

```
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

if (err) {
  char log[10240] = "";
  err = clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
                                          sizeof(log), log, NULL);
  printf("Program build log:\n%s\n", log);
}

kernel = clCreateKernel(program, "average_images", &err);
```

Executing Kernels

Set the kernel arguments

Enqueue the kernel

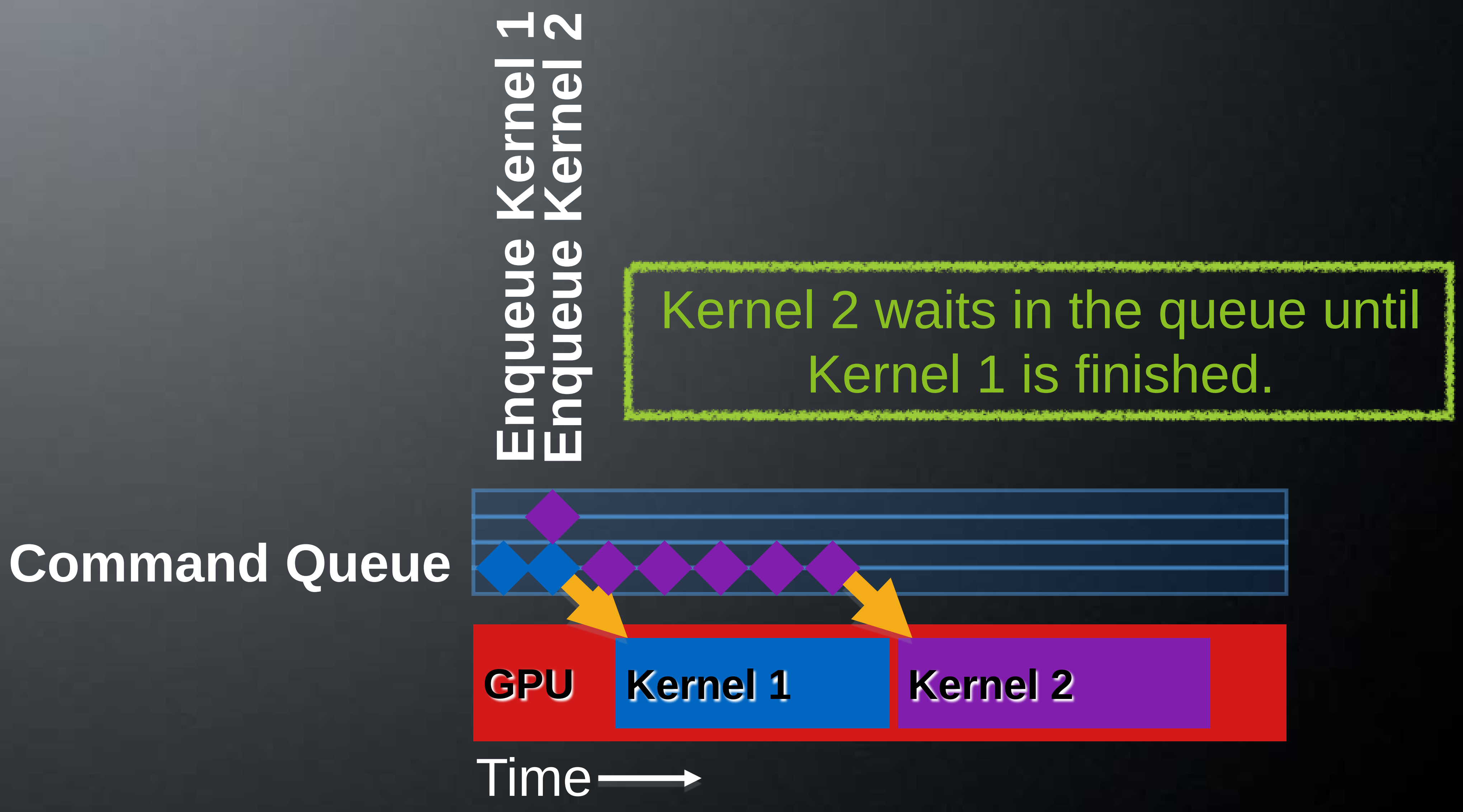
```
err = clSetKernelArg(kernel, 0, sizeof(input), &input);
err = clSetKernelArg(kernel, 1, sizeof(output), &output);

size_t global[3] = {image_width, image_height, 0};
err = clEnqueueNDRangeKernel(queue, kernel, 2, NULL, global, NULL, 0, NULL, NULL);
```

- Note: Your kernel is executed ***asynchronously***
  - Nothing may happen—you have only enqueued your kernel
  - Use a blocking read clEnqueueRead*(... CL_TRUE ...)
  - Use events to track the execution status

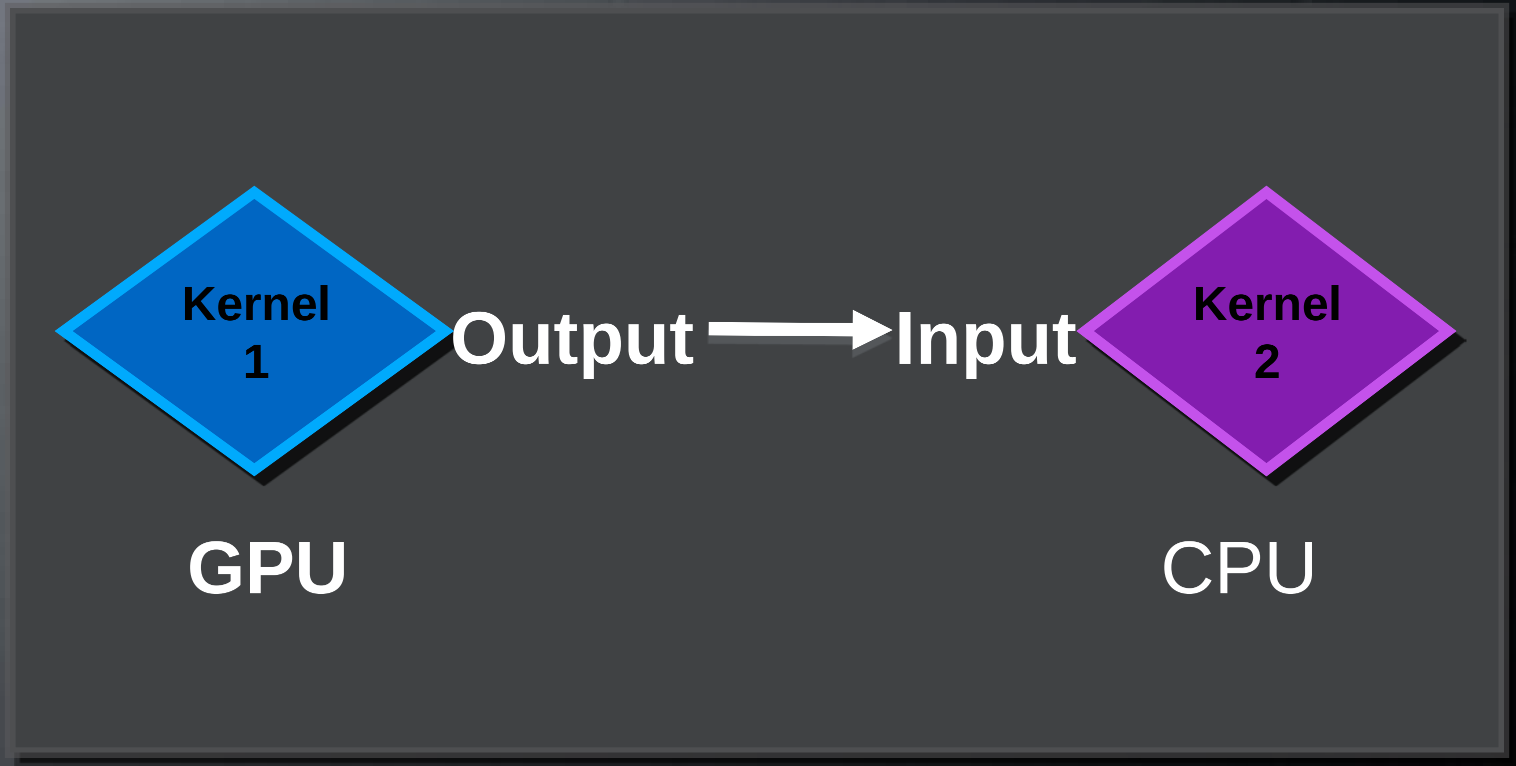
Synchronization Between Commands

Synchronization: One Device/Queue

- Example: Kernel 2 uses the results of Kernel 1

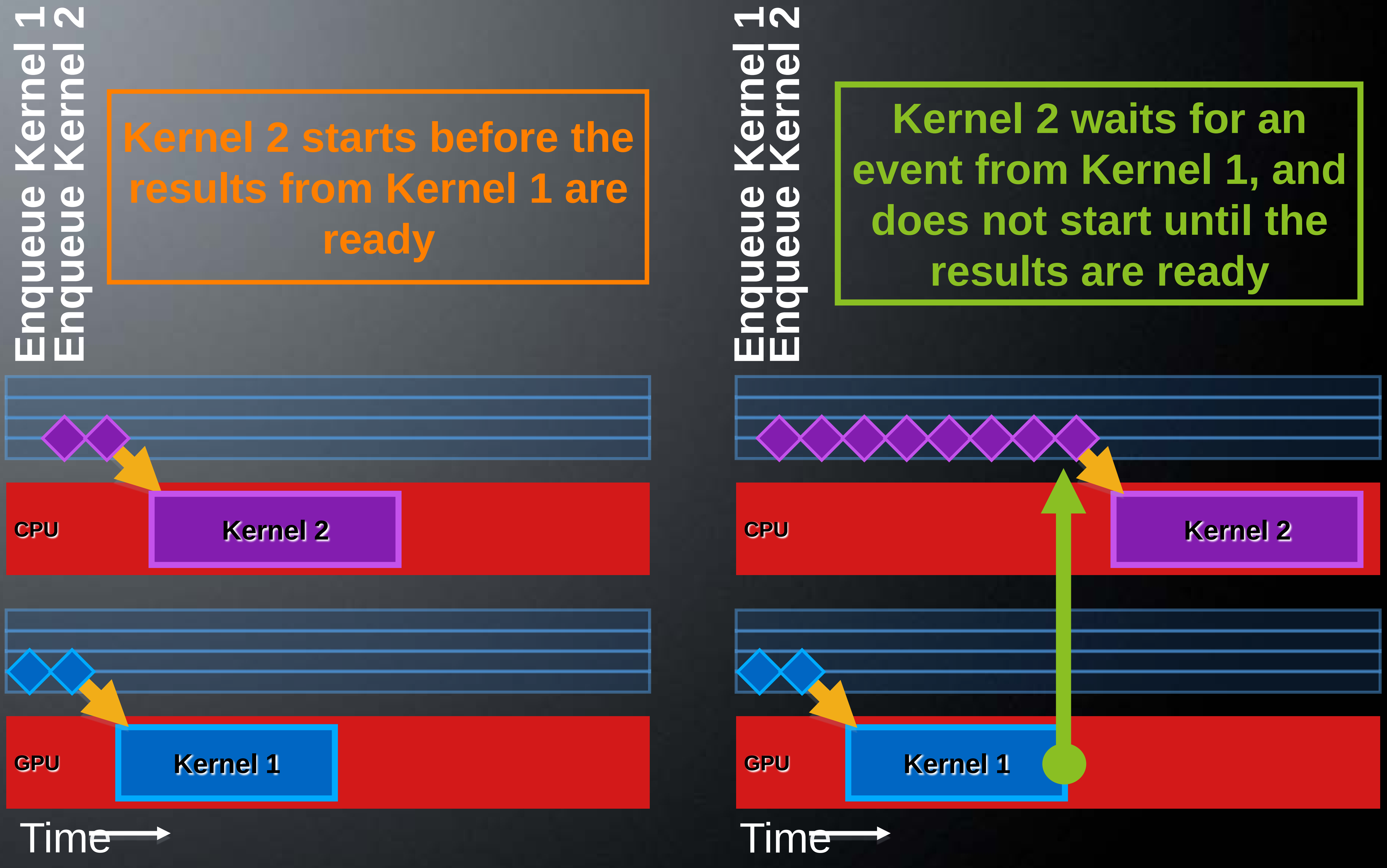


Synchronization: Two Devices/Queues



Explicit dependency: Kernel 1 must finish before Kernel 2 starts

Synchronization: Two Devices/Queues



Using Events on the Host

clWaitForEvents(num_events, *event_list)

- Blocks until events are complete

clEnqueueMarker(queue, *event)

- Returns an event for a marker that moves through the queue

clEnqueueWaitForEvents(queue, num_events, *event_list)

- Inserts a "WaitForEvents" into the queue

clGetEventInfo()

- Command type and status

  CL_QUEUED, CL_SUBMITTED, CL_RUNNING, CL_COMPLETE, or error code

clGetEventProfilingInfo()

- Command queue, submit, start, and end times

Part 5: OpenCL™ C

- Language Features
- Built-in Functions

OpenCL™ C Language

Derived from ISO C99

- No standard C99 headers, function pointers, recursion, variable length arrays, and bit fields

Additions to the language for parallelism

- Work-items and workgroups
- Vector types
- Synchronization

Address space qualifiers

Optimized image access

Built-in functions

Address space

- __global – memory allocated from global address space, images are global by default
- __constant – is like global, but read only
- __local – memory shared by work-group
- __private – private per work-item memory
- __read_only – only for images
- __write_only – only for images

Kernel args have to be global, constant or local.

Can’t assign to different pointer type.

Workgroups

- uint **get_work_dim () (1 to 3)**
- size_t **get_global_size (uint dimindx)**
- size_t **get_global_id (uint dimindx)**
- size_t **get_local_size (uint dimindx)**
- size_t **get_local_id (uint dimindx)**
- size_t **get_num_groups (uint dimindx)**
- size_t **get_group_id (uint dimindx)**

num_groups * local_size = global_size

local_id + group_id * local_size = global_id

global_size % local_size = 0

Synchronization

`barrier()` function. All work-items must reach the barrier before they execute further. It must be encountered by all work-items in work-group.

Flags: LOCAL_MEM_FENCE, GLOBAL_MEM_FENCE – flush and ensure ordering for local or global memory.

`mem_fence()`, `read_mem_fence()`, `write_mem_fence()` – ensure memory loads and stores ordering within work-item.

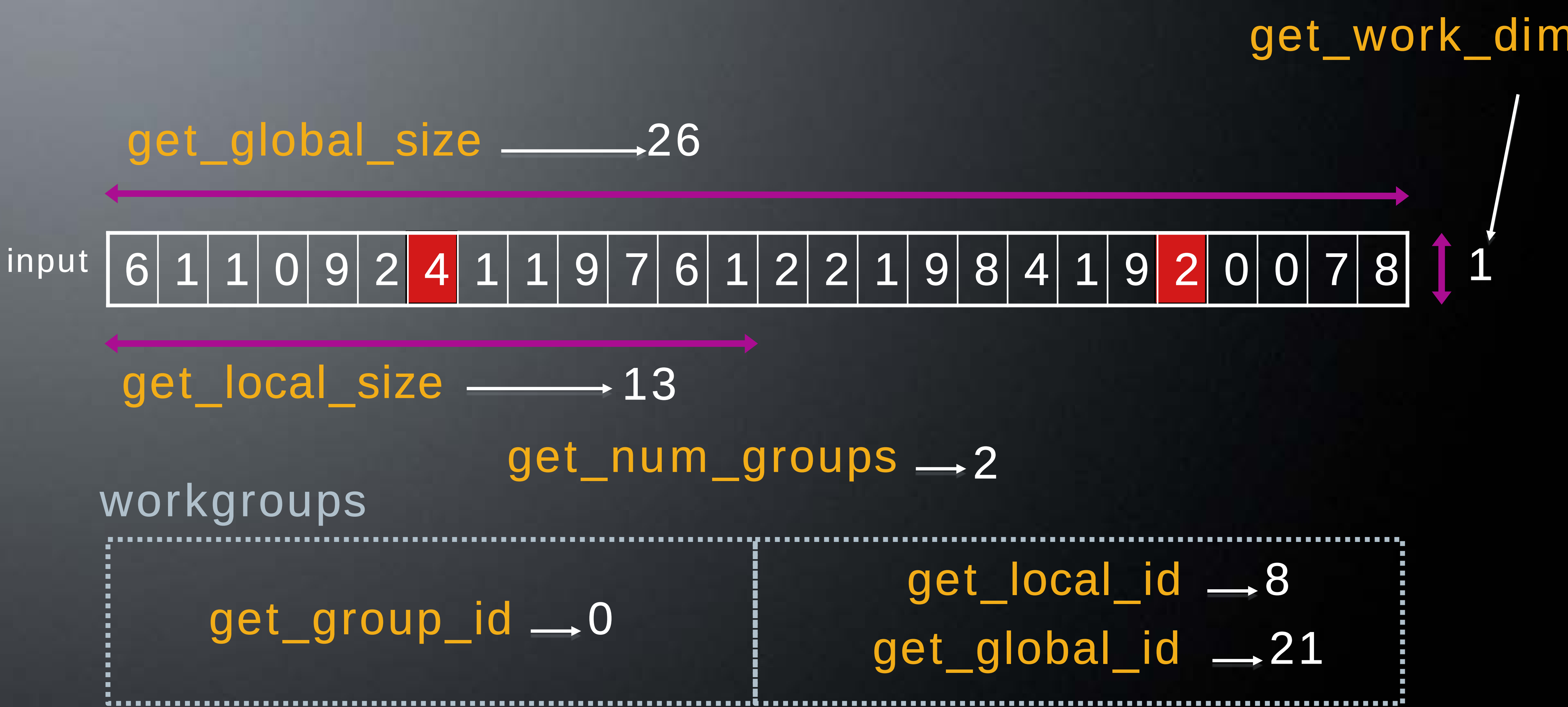
Kernel

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

get_global_id(0) → i==11

input	6	1	1	0	9	2	4	1	1	9	7	6	1	2	2	1	9	8	4	1	9	2	0	0	7	8
output	36	1	1	0	81	4	16	1	1	81	49	36	1	4	4	1	81	64	16	1	81	4	0	0	49	64

Work-Items and Workgroup Functions



Data Types

Scalar data types

- char , uchar, short, ushort, int, uint, long, ulong
- bool, intptr_t, ptrdiff_t, size_t, uintptr_t, void,
- half (storage)

Image types

- image2d_t, image3d_t, sampler_t

Vector data types

Data Types

Portable

Vector length of 2, 4, 8, and 16

- char2, ushort4, int8, float16, double2, …

Endian safe

Aligned at vector length

Vector operations and built-in functions

Vector Operations

- Vector literal

```
int4 vi0 = (int4) -7;
int4 vi1 = (int4)(0, 1, 2, 3);
```

-7	-7	-7	-7

0	1	2	3

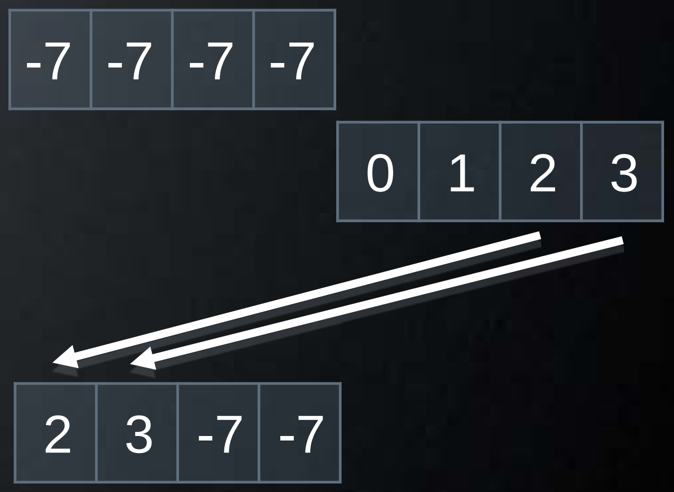
Vector Operations

- Vector literal

```
int4 vi0 = (int4) -7;

int4 vi1 = (int4)(0, 1, 2, 3);
```

- Vector components

```
vi0.lo = vi1.hi;
```



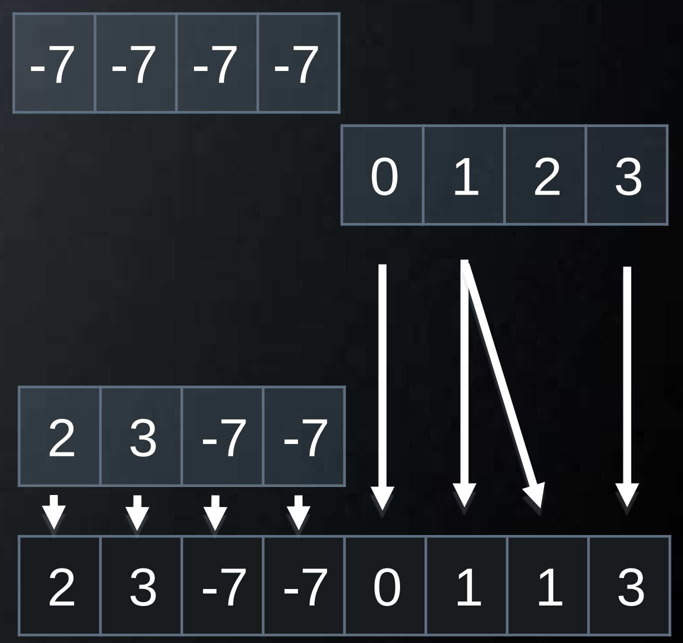
Vector Operations

- Vector literal

```
int4 vi0 = (int4) -7;

int4 vi1 = (int4)(0, 1, 2, 3);
```

- Vector components

```
vi0.lo = vi1.hi;

int8 v8 = (int8)(vi0.s0123, vi1.odd);
```



Vector Operations

- Vector literal

```
int4 vi0 = (int4) -7;

int4 vi1 = (int4)(0, 1, 2, 3);
```

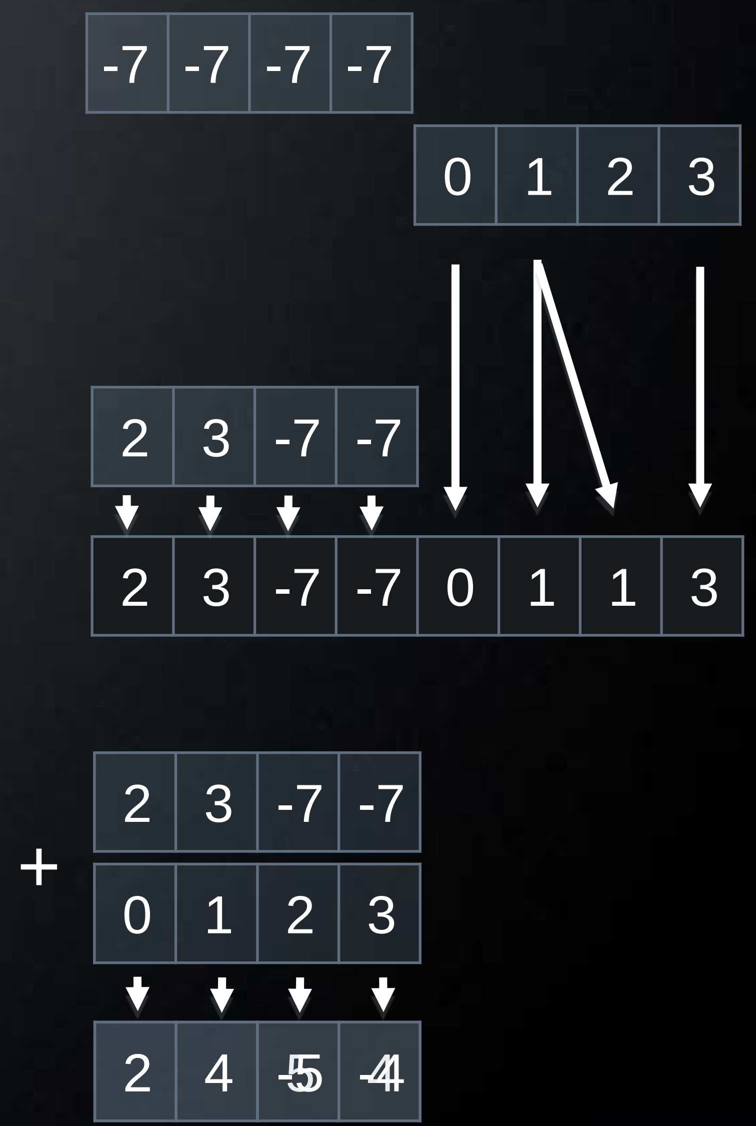
- Vector components

```
vi0.lo = vi1.hi;

int8 v8 = (int8)(vi0.s0123, vi1.odd);
```

- Vector ops

```
vi0 += vi1;
```

```
vi0 = abs(vi0);
```



Address Spaces

- Kernel pointer arguments must use global, local, or constant

```
kernel void distance(global float8* stars, local float8* local_stars)
kernel void sum(private int* p)   // Illegal because is uses private
```

- Default address space for arguments and local variables is private

```
kernel void smooth(global float* io) {
  float temp;
  ...
```

- image2d_t and image3d_t are always in global address space

```
kernel void average(read_only global image_t in, write_only
image2d_t out)
```

Address Spaces

- Program (global) variables must be in constant address space

```
constant float bigG = 6.67428E-11;
global   float time;              // Illegal non constant
kernel void force(global float4 mass) { time = 1.7643E18f; }
```

- Casting between different address spaces is undefined

```
kernel void calcEMF(global float4* particles) {
  global float* particle_ptr = (global float*) particles;
  float* private_ptr = (float*) particles; // Undefined behavior -
  float particle = * private_ptr;          //  different address
}
```

Conversions

Scalar and pointer conversions follow C99 rules

- No implicit conversions for vector types

```
float4 f4 = int4_vec;            // Illegal implicit conversion
```

- No casts for vector types (different semantics for vectors)

```
float4 f4 = (float4) int4_vec;  // Illegal cast
```

- Casts have other problems

```
float x;
int i = (int)(x + 0.5f);         // Round float to nearest integer
```

Wrong for:

0.5f - 1 ulp (rounds up not down)

negative numbers (wrong answer)

- There is hardware to do it on nearly every machine

Conversions

Explict conversions:

convert_*destType*<_saturate><_roundingMode>

- Scalar and vector types
- No ambiguity

```
uchar4 c4 = convert_uchar4_sat_rte(f4);
```

f4	-5.0f	254.5f	254.6	1.2E9f
c4	0	254	255	255
	Saturate to 0	Round down to nearest even	Round up to nearest value	Saturated to 255

Reinterpret Data: *as_typen*

Reinterpret the bits to another type

Types must be the same size

```
// f[i] = f[i] < g[i] ? f[i] : 0.0f
float4 f, g;
int4 is_less = f < g;
f = as_float4(as_int4(f) & is_less);
```

f	-5.0f	254.5f	254.6f	1.2E9f
g	254.6f	254.6f	254.6f	254.6f
is_less	ffffffff	ffffffff	00000000	00000000
as_int	c0a00000	42fe0000	437e8000	4e8f0d18
&	c0a00000	42fe0000	00000000	00000000
f	-5.0f	254.5f	0.0f	0.0f

OpenCL™ provides a select built-in

Built-in Math Functions

IEEE 754 compatible rounding behavior for single precision floating-point

IEEE 754 compliant behavior for double precision floating-point

Defines maximum error of math functions as ULP values

Handle ambiguous C99 library edge cases

Commonly used single precision math functions come in three flavors

- eg. log(x)
  - Full precision <= 3ulps
  - Half precision/faster. half_log—minimum 11 bits of accuracy, <= 8192 ulps
  - Native precision/fastest. native_log: accuracy is implementation defined
- Choose between accuracy and performance

Built-in Work-group Functions

```
kernel read(global int* g, local int* shared) {
   if (get_global_id(0) < 5)
     barrier(CLK_GLOBAL_MEM_FENCE);
   else
     k = array[0];
}
```

work-item 0

work-item 6

Illegal since not all work-items encounter barrier

AMD The future is fusion

Built-in Functions

Integer functions

- abs, abs_diff, add_sat, hadd, rhadd, clz, mad_hi, mad_sat, max, min, mul_hi, rotate, sub_sat, upsample

Image functions

- read_image[f | i | ui]
- write_image[f | i | ui]
- get_image_[width | height | depth]

Common, Geometric and Relational Functions

Vector Data Load and Store Functions

- eg. vload_half, vstore_half, vload_halfn, vstore_halfn, ...

Extensions

Atomic functions to global and local memory

- add, sub, xchg, inc, dec, cmp_xchg, min, max, and, or, xor
- 32-bit/64-bit integers

Select rounding mode for a group of instructions at compile time

- For instructions that operate on floating-point or produce floating-point values
- #pragma OpenCL_select_rounding_mode rounding_mode
- All 4 rounding modes supported

Extension: Check `clGetDeviceInfo` with `CL_DEVICE_EXTENSIONS`

OpenCL™ Language

Show the SDK