# Programming with OpenCL™

## Introduction, Architecture and Programming

OpenCL

René Oertel (rene.oertel@cs.tu-chemnitz.de)

`http://www.tu-chemnitz.de/cs/ra`

Professorship of Computer Architecture
Faculty of Computer Science
Technische Universität Chemnitz



TECHNISCHE UNIVERSITÄT CHEMNITZ

# Lecture Outline

# Objectives of this lecture

- Introduces essential basics of OpenCL
- Introduces important programming aspects
- Establishes mapping between hardware and software
- Provides an well-known example

# Current Step

# What is it? I

- Standard of Apple Inc. and the Khronos Group [1]
- Specification versions:
  - v1.0 (December 9, 2008), revision 48 (October 6, 2009)
  - v1.1 (June 14, 2010), revision 44 (June 1, 2011)
  - v1.2 (November 15, 2011), revision 19 (November 14, 2012)
  - v2.0 (November 18, 2013), revision 26 (October 17, 2014)
  - v2.1 provisional revision 8 (January 29, 2015)
- Royalty free
- Open industry standard for programming a heterogeneous collection of CPUs, GPUs and other discrete computing devices
- Framework for parallel programming: Language, API, libraries and a runtime system
- Provides a low-level hardware abstraction

# What is it? II

## License Agreement

OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos.

OpenCL (Open Computing Language):

- Supports both data- and task-based parallel programming models
- Based on ISO C99
- Restrictions: C99 headers, function pointers, recursion, variable length arrays, ...
- Additions: vector types, work-items and -groups, address space qualifiers, synchronization, ...
- Defines consistent numerical requirements based on IEEE 754

# What is it? III



OpenCL uses a hierarchy of models:

1. Platform Model
2. Execution Model
3. Memory Model
4. Programming Model

# Current Step

# Implementations I



- Altera SDK for OpenCL
- AMD APP SDK
- Apple OS X OpenCL
- (†) NVIDIA OpenCL
- Intel OpenCL Code Builder (Intel SDK for OpenCL Applications)
- Intel Beignet Project[1]
- † PGI OpenCL Compiler for ARM
- IBM OpenCL Development Kit/Common Runtime
- Motorola, Mozilla, Nokia, Samsung WebCL[2] for browsers
- Portable Computing Language[3]

# Implementations II

1. ICDs (Installable Client Drivers): One or more OpenCL implementations
2. ICD registry: /etc/OpenCL/vendors/*.icd
3. ICD loader: libOpenCL.so

---

[1] `http://www.freedesktop.org/wiki/Software/Beignet/`

[2] WebCL v1.0 March 14, 2014: `http://www.khronos.org/webcl/`

[3] `http://www.portablecl.org/`

# Current Step

# Platform Model I

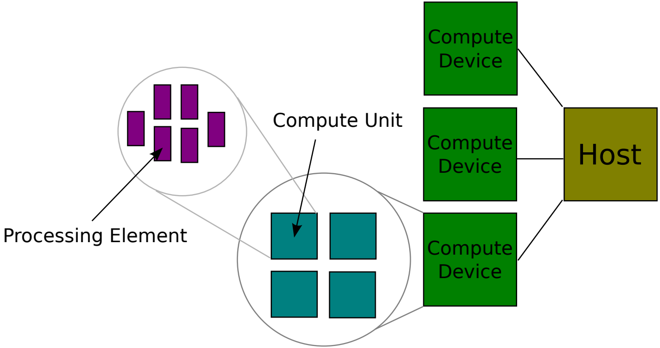


Figure: OpenCL Platform Model

# Platform Model II

One *host* =

- e.g. PC, embedded system or super computer

is connected to one or more *compute devices* =

- e.g. CPU, GPU device, DSP

with multiple *compute units (CUs)* =

- e.g. Streaming Multiprocessors (NVIDIA).

The compute units are comprised of several *processing elements (PEs)* =

- e.g. Scalar Processors (NVIDIA), Stream Cores with PEs (ATI/AMD).

# Platform Model III

- *Host* OpenCL application submits *commands* on the *PEs*
- *PEs* execute code as *SIMD* or *SPMD* (i.e. own program counter) units

# Current Step

# Execution Model I

- There are two different execution parts:
  1. Host program: Executes on the *host*
  2. Kernel: Executes on one or more *compute devices*
- Kernel execution creates an N-dimensional (N = 1,2,3) index space: *NDRange* (NVIDIA: grid of thread blocks)
- *Work-groups* (NVIDIA: thread block, ATI/AMD: group of wavefronts) are a coarse-grained decomposition if the index space
- An instance (*work-item*) (NVIDIA: thread, ATI/AMD: compute shader instance) of the kernel executes on each point of the index space
- Work-items of a work-group execute concurrently on the PEs of a CU
- Identifier of a work-item are the *global ID* or the *local ID* with the *group ID*

# Execution Model II

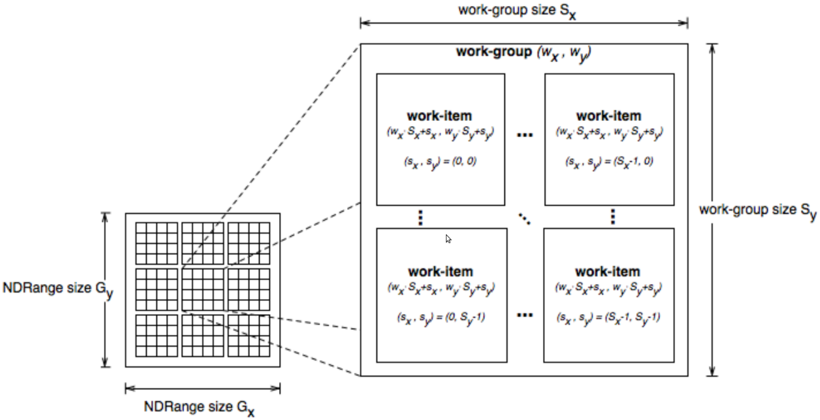


Figure: OpenCL Execution Model - NDRange mapping (Source: [1])

# Execution Model – Context

*Host* defines a context for the execution of the kernels:

- *Devices*: The collection of OpenCL devices to be used by the host
- *Kernels*: The OpenCL functions that run on OpenCL devices
- *Program Objects*: The program sources and executables that implement the kernels
- *Memory Objects*: A set of memory objects visible to the host and the OpenCL devices. Memory objects contain values that can be operated on by instances of a kernel.

OpenCL API functions are used to create and manipulate contexts.

# Execution Model – Command Queue



- *Command queue* is a data-structure for the coordination of kernel execution
- Host places commands into it within the context:
  1. Kernel execution commands
  2. Memory commands
  3. Synchronization commands
- Commands are scheduled and executed asynchronously between host and device with two modes:
  1. In-order: Serializes the execution order
  2. Out-of-order: Order constraints are enforced by the programmer through explicit synchronization

# Execution Model – Kernel Categories

Two categories of kernels are supported:

1. OpenCL kernels:
   - Written with the OpenCL C programming language
   - Compiled with the OpenCL compiler
2. Native kernels:
   - Accessed through a host function pointer
   - Ability to execute is an optional functionality
   - OpenCL API includes functions to query capability
   - E.g. an export of a library

# Current Step

# Memory Model I

*Work-items* have access to four distinct memory regions:

1. *Global Memory*:
   - Read/write access to all work-items in all work-groups
   - Work-items can read from or write to any element of a memory object
   - May be cached depending on the capabilities of the device
2. *Constant Memory*:
   - Region of global memory that remains constant during the execution of a kernel
   - Host allocates and initializes memory objects placed into constant memory

# Memory Model II

3. *Local Memory*:
   - Local to a work-group
   - Allocation of variables that are shared by all work-items in that work-group
   - May be implemented as a dedicated regions of memory
   - Alternatively, may be mapped onto sections of the global memory
4. *Private Memory*:
   - Accessed through a host function pointer
   - Private to a work-item
   - Variables are not visible to another work-item

# Memory Model III

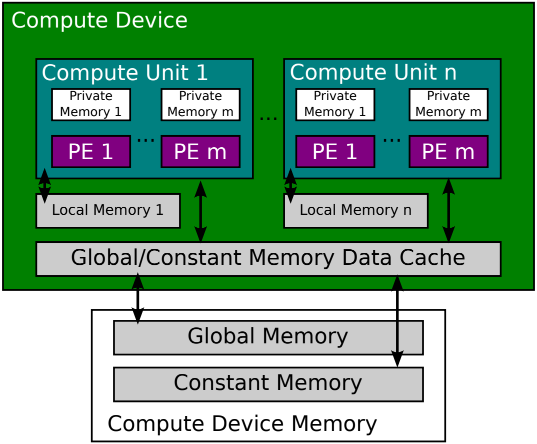


Figure: OpenCL Memory Model (Source: [1])

# Current Step

# Programming Models I

1. *Data Parallel Programming Model*:
   - Primary model of OpenCL
   - Defines a computation in terms of a sequence of instructions applied to multiple elements of a memory object
   - Index space defines the work-items and the data maps
   - OpenCL implements a relaxed version where a strict one-to-one mapping is not a requirement for the work-items
   - Support of *explicit model* (programmer defines number and the distribution of work-items to work-groups) and the *implicit model* (OpenCL implementation distributes the defined number of work-items)

2. *Task Parallel Programming Model*:
    - Single instance of a kernel is executed independent of any index space
    - Logically equivalent to executing a kernel with a work-group containing a single work-item
    - Parallelism e.g. with vector data types or enqueuing multiple tasks

# Synchronization



Two domains of synchronization in OpenCL:

1. Work-items in a single work-group (with *work-group barriers*)
2. Commands enqueued to command-queue(s) in a single context:
    - *Command-queue barriers*: Ensure that all previously queued commands have finished execution; Can only be used in a single command-queue
    - *Event waits*: All OpenCL API functions that enqueue commands return an event that identifies the command and memory objects it updates

# Framework

- *OpenCL Platform layer*: Allows the host program to discover OpenCL devices and their capabilities and to create contexts
- *OpenCL Runtime*: Allows the host program to manipulate contexts
- *OpenCL Compiler*: Creates program executables that contain OpenCL kernels

OpenCL C programming language implemented by the compiler supports a subset of the ISO C99 language with extensions for parallelism

# OpenCL Program Flow

1. Enumerate platforms
2. Enumerate devices
3. Create context
4. Create device-specific command queues
5. Create program
6. Allocate and initialize memory on host or device
7. Transfer data to the device
8. Set arguments and enqueue kernel
9. Sync
10. Read results back to the host
11. Clean up

# Simple Buffer Write

- Example of the AMD OpenCL Programming Guide ([3])
- Step-by-step of the above program flow
- Additionally, background information of the API and the OpenCL models
- C language, C++ bindings are available
- **No** error checks for simplicity ;)

# Current Step

1. Introduction

2. Architecture

3. Programming Workflow
   - Enumerate Platforms
   - Enumerate Devices
   - Create Context
   - Create Command Queue
   - Create Program
   - Memory Management
   - Kernel Enqueuing
   - Synchronizing
   - Gathering Results
   - Cleaning up

# Recall: Platform Model

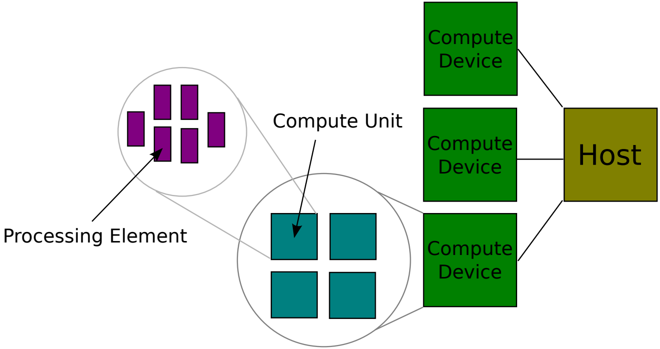


Figure: OpenCL Platform Model

# Enumerate Platforms I

Chapter 4.1 Querying Platform Info of OpenCL specification

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

- `num_entries`: Provides number of possible elements in `platforms`
- `platforms`: Returns list of found OpenCL platforms
- `num_platforms`: Returns number of found OpenCL platforms (optional)

# Enumerate Platforms II

*Example*:

```
int main(int argc, char** argv)
{
    // 1. Get a platform.

    cl_platform_id platform;
    clGetPlatformIDs( 1, &platform, NULL );

    //...
```

# Enumerate Platforms III

```
cl_int clGetPlatformInfo (cl_platform_id platform,
                          cl_platform_info param_name,
                          size_t param_value_size,
                          void *param_value,
                          size_t *param_value_size_ret)
```

- `platform`: Platform to query
- `param_name`: Platform information being queried (e.g. CL_PLATFORM_NAME, CL_PLATFORM_VENDOR, CL_PLATFORM_VERSION, ...)
- `param_value_size`: Size of the buffer to hold information
- `param_value`: Pointer to memory location of the buffer
- `param_value_size_ret`: Actual number of bytes of the information

# Enumerate Platforms IV



*Examples*:

```
char pbuff[100];

clGetPlatformInfo( platform,
                   CL_PLATFORM_NAME,
                   sizeof(pbuff), pbuff, NULL);

printf("CL_PLATFORM_NAME:␣%s\n", pbuff);

clGetPlatformInfo( platform,
                   CL_PLATFORM_VENDOR,
                   sizeof(pbuff), pbuff, NULL);

printf("CL_PLATFORM_VENDOR:␣%s\n", pbuff);
```

*Example output*:

```
CL_PLATFORM_NAME: ATI Stream
CL_PLATFORM_VENDOR: Advanced Micro Devices, Inc.
```

# Current Step

1. Introduction
2. Architecture
3. Programming Workflow
   - Enumerate Platforms
   - Enumerate Devices
   - Create Context
   - Create Command Queue
   - Create Program
   - Memory Management
   - Kernel Enqueuing
   - Synchronizing
   - Gathering Results
   - Cleaning up

# Enumerate Devices I

Chapter 4.2 Querying Devices of OpenCL specification

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

- `platform`: Platform to query
- `device_type`: Type of OpenCL devices to query (e.g. CL_DEVICE_TYPE_GPU, CL_DEVICE_TYPE_ACCELERATOR, CL_DEVICE_TYPE_ALL, ...)
- `num_entries`: Size of the device buffer
- `devices`: Pointer to memory location of the device buffer
- `num_devices`: Actual number of devices of device_type

# Enumerate Devices II

*Example*:

```
// ...

// 2. Find a gpu device.

cl_device_id device;

clGetDeviceIDs( platform, CL_DEVICE_TYPE_GPU,
                          1,
                          &device,
                          NULL);

// ...
```

# Enumerate Devices III

```
cl_int clGetDeviceInfo (cl_device_id device,
                        cl_device_info param_name,
                        size_t param_value_size,
                        void *param_value,
                        size_t *param_value_size_ret)
```

- `device`: Device to query
- `param_name`: Device information being queried (e.g. CL_DEVICE_TYPE, CL_DEVICE_NAME, CL_DEVICE_MAX_CLOCK_FREQUENCY, CL_DEVICE_GLOBAL_MEM_SIZE, ...)
- `param_value_size`: Size of the buffer to hold information
- `param_value`: Pointer to memory location of the buffer
- `param_value_size_ret`: Actual number of bytes of the information

# Enumerate Devices IV

*Example*:

```
// ...
char dbuff[100];

clGetDeviceInfo(device, CL_DEVICE_NAME,
                sizeof(dbuff), dbuff, NULL);

printf("CL_DEVICE_NAME:␣%s\n", dbuff);

cl_ulong global_mem;

clGetDeviceInfo(device, CL_DEVICE_GLOBAL_MEM_SIZE,
                sizeof(global_mem), &global_mem, NULL);

printf("CL_DEVICE_GLOBAL_MEM_SIZE:␣%llu\n",
       (long long unsigned int)global_mem);
```

*Example output*:

```
CL_DEVICE_NAME: Tesla K20c
CL_DEVICE_GLOBAL_MEM_SIZE: 5368512512
```

# Current Step

1. Introduction
2. Architecture
3. Programming Workflow
   - Enumerate Platforms
   - Enumerate Devices
   - Create Context
   - Create Command Queue
   - Create Program
   - Memory Management
   - Kernel Enqueuing
   - Synchronizing
   - Gathering Results
   - Cleaning up

# Create Context I

Chapter 4.3 Contexts of OpenCL specification

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

- `properties`: Platform to use, `NULL` for auto-select
- `num_devices`: Number of devices
- `devices`: List of unique devices of `clGetDeviceIDs`
- `pfn_notify`: Pointer of callback function for error notifications
- `user_data`: Data for callback function
- `errcode_ret`: Returned error code

# Create Context II

*Example*:

```
// ...

// 3. Create a context on that device.

cl_context context = clCreateContext( NULL,
                                      1,
                                      &device,
                                      NULL, NULL, NULL);

// ...
```

# Current Step

1. Introduction
2. Architecture
3. Programming Workflow
   - Enumerate Platforms
   - Enumerate Devices
   - Create Context
   - **Create Command Queue**
   - Create Program
   - Memory Management
   - Kernel Enqueuing
   - Synchronizing
   - Gathering Results
   - Cleaning up

# Create Command Queue I

Chapter 5.1 Command Queues of OpenCL specification

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

- `context`: Valid OpenCL context
- `device`: Device associated with the context
- `properties`: Properties of the command queue (e.g. out-of-order execution and profiling)
- `errcode_ret`: Returned error code

# Create Command Queue II

*Example*:

```
// ...

// 3. Create a command queue on that device.

cl_command_queue queue = clCreateCommandQueue( context,
                                               device,
                                               0, NULL );

// ...
```

# Current Step

1. Introduction

2. Architecture

3. Programming Workflow
   - Enumerate Platforms
   - Enumerate Devices
   - Create Context
   - Create Command Queue
   - Create Program
   - Memory Management
   - Kernel Enqueuing
   - Synchronizing
   - Gathering Results
   - Cleaning up

Chapter 5.4.1 Creating Program Objects of OpenCL specification

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

- `context`: Valid OpenCL context
- `count`: Number of `strings`
- `strings`: Null-terminated source code string
- `lengths`: Length of source code if not null-terminated
- `errcode_ret`: Returned error code

*Example*:

```
// ...
const char *source =
"__kernel␣void␣memset(␣__global␣uint␣*dst␣)␣␣␣\n"
"{␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n"
"␣␣␣dst[get_global_id(0)]␣=␣get_global_id(0);␣\n"
"}␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n";
// ...

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

// ...
```

# Build Program I

Chapter 5.4.2 Building Program Executables of OpenCL specification

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

- `program`: Resulting program object
- `num_devices`: Number of devices in `device_list`
- `device_list`: List of devices
- `options`: Pointer to string of build options
- `pfn_notify`: Pointer of callback function called after build
- `user_data`: Data for callback function

# Build Program II

*Example*:

```
// ...
clBuildProgram( program, 1, &device, NULL, NULL, NULL );
// ...
```

# Create Kernel I

Chapter 5.5.1 Creating Kernel Objects of OpenCL specification

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

- `program`: Valid program object of `clBuildProgram`
- `kernel_name`: Kernel function (declared with `__kernel`)
- `errcode_ret`: Appropriate error code

Returns a valid non-zero kernel object.

# Create Kernel II

*Example*:

```
// ...
cl_kernel kernel = clCreateKernel( program, "memset", NULL );
// ...
```

# Current Step

1. Introduction
2. Architecture
3. Programming Workflow
   - Enumerate Platforms
   - Enumerate Devices
   - Create Context
   - Create Command Queue
   - Create Program
   - Memory Management
   - Kernel Enqueuing
   - Synchronizing
   - Gathering Results
   - Cleaning up

Chapter 5.2.1 Creating Buffer Objects of OpenCL specification

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

- `context`: Valid OpenCL context
- `flags`: Usage and region flags (e.g. CL_MEM_READ_ONLY, CL_MEM_COPY_HOST_PTR)
- `size`: Size in bytes of the memory object
- `host_ptr`: Pointer to preallocated host memory buffer
- `errcode_ret`: Returned error code

*Example 1*: Creating the result buffer

```
// ...

// 5. Create a data buffer.

cl_mem buffer = clCreateBuffer( context,
                                CL_MEM_WRITE_ONLY,
                                NWITEMS * sizeof(cl_uint),
                                NULL, NULL );


// ...
```

# Memory Management III

*Example 2*: Use host buffer to allocate device memory (binary search example of AMD APP SDK v2.4)

```
// ...

inputBuffer = clCreateBuffer(
                    context,
                    CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
                    sizeof(cl_uint) * length,
                    input,
                    &status);

// ...
```

CL_MEM_USE_HOST_PTR vs.
CL_MEM_ALLOC_HOST_PTR vs.
CL_MEM_COPY_HOST_PTR

# Current Step

# Recall: Execution Model

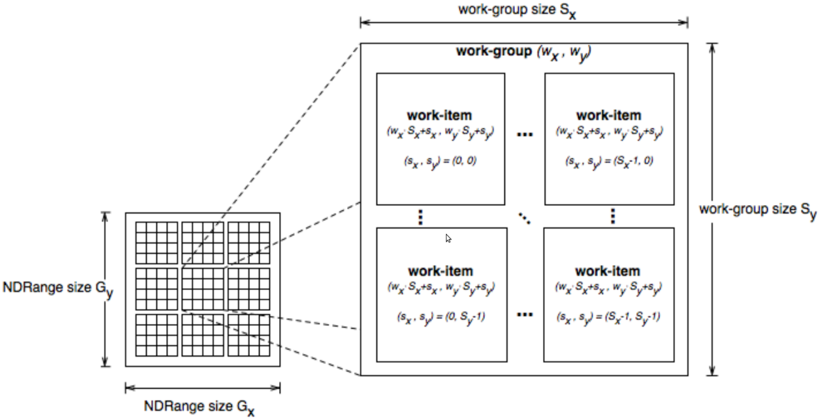


Figure: OpenCL Execution Model - NDRange mapping (Source: [1])

Chapter 5.5.2 Setting Kernel Arguments of OpenCL specification

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

- `kernel`: Valid kernel object
- `arg_index`: Parameter index of argument
- `arg_size`: Size in bytes of the argument
- `arg_value`: Value of argument

`clSetKernelArg` must be repeated for all arguments separately!

# Kernel Enqueuing II



*Example*:

```
// ...
clSetKernelArg(kernel, 0, sizeof(buffer), (void*) &buffer);
// ...
```

Chapter 5.6 Executing Kernels of OpenCL specification

```
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)
```

- `command_queue`: Valid command-queue
- `kernel`: Valid kernel object
- `work_dim`: Number of dimensions to specify work-items and -groups
- `global_work_offset`: Currently `NULL`

# Kernel Enqueuing IV

- `global_work_size`: Array with `work_dim` entries with the number of work-items per dimension
- `local_work_size`: Array with values for work-group size or `NULL` for auto-select
- `num_events_in_wait_list`: Number of events
- `event_wait_list`: Events that need to complete before this particular command can be executed
- `event`: Unique event object for the execution instance

# Kernel Enqueuing V

*Example*:

```
// ...

// 6. Launch the kernel. Let OpenCL pick the local work size.

size_t global_work_size = NWITEMS;

clEnqueueNDRangeKernel( queue,
                        kernel,
                        1,
                        NULL,
                        &global_work_size,
                        NULL, 0, NULL, NULL);

// ...
```

# Current Step



1. Introduction
2. Architecture
3. Programming Workflow
   - Enumerate Platforms
   - Enumerate Devices
   - Create Context
   - Create Command Queue
   - Create Program
   - Memory Management
   - Kernel Enqueuing
   - Synchronizing
   - Gathering Results
   - Cleaning up

# Synchronizing I

Chapter 5.10 Flush and Finish of OpenCL specification

```
cl_int clFinish (cl_command_queue command_queue)
```

- `command_queue`: Valid command queue

`clFinish` does not return until all queued commands in command_queue have been processed and completed!

# Synchronizing II

*Example*:

```
// ...
clFinish( queue );
// ...
```

# Current Step

1. Introduction
2. Architecture
3. Programming Workflow
   - Enumerate Platforms
   - Enumerate Devices
   - Create Context
   - Create Command Queue
   - Create Program
   - Memory Management
   - Kernel Enqueuing
   - Synchronizing
   - Gathering Results
   - Cleaning up

Chapter 5.2.8 Mapping and Unmapping Memory Objects of OpenCL specification

```
void * clEnqueueMapBuffer (cl_command_queue command_queue,
                          cl_mem buffer,
                          cl_bool blocking_map,
                          cl_map_flags map_flags,
                          size_t offset,
                          size_t cb,
                          cl_uint num_events_in_wait_list,
                          const cl_event *event_wait_list,
                          cl_event *event,
                          cl_int *errcode_ret)
```

- `command_queue`: Valid command queue
- `buffer`: Valid OpenCL buffer object
- `blocking_map`: Mode of the map operation (non-/blocking)
- `map_flags`: Type of mapping (read and/or write)

# Gathering Results II

- `offset/cb`: Offset and size in bytes of the region
- `num_events_in_wait_list`: Number of events
- `event_wait_list`: Events that need to complete before this particular command can be executed
- `event`: Unique event object for the execution instance
- `errcode_ret`: Returned error code

Returns a pointer to the mapped region.

*Example*:

```
// ...

// 7. Look at the results via synchronous buffer map.

cl_uint *ptr;

ptr = (cl_uint *) clEnqueueMapBuffer( queue,
                                      buffer,
                                      CL_TRUE,
                                      CL_MAP_READ,
                                      0,
                                      NWITEMS * sizeof(cl_uint),
                                      0, NULL, NULL, NULL );

// ...
```

# Current Step

# Cleaning up I



Chapters of the creation functions

```
cl_int clReleaseKernel (cl_kernel kernel)
cl_int clReleaseProgram (cl_program program)
cl_int clReleaseMemObject (cl_mem memobj)
```

- `kernel`: Valid OpenCL kernel object
- `program`: Valid OpenCL program object
- `memobj`: Valid OpenCL memory object

Usage necessary if used in loops, else the memory management of the operating system will free memory after the **whole** program has finished!

# Cleaning up II

*Example*:

```
// ...
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseMemObject(buffer);
// ...
```

# Matrix Multiplication I

Simple version of the NVIDIA OpenCL Programming Guide ([4])

*Host code*:

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.width + col)
typedef struct {
    int width;
    int height;
    cl_mem elements;
} Matrix;
// Thread block size
#define BLOCK_SIZE 16
// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMulHost(const Matrix A, const Matrix B, Matrix C,
                const cl_context context,
                const cl_kernel matMulKernel,
                const cl_command_queue queue)
{
  // Load A and B to device memory
  Matrix d_A;
  d_A.width = A.width; d_A.height = A.height;
```

```
size_t size = A.width * A.height * sizeof(float);
d_A.elements = clCreateBuffer(context,
                      CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                      size, A.elements, 0);
Matrix d_B;
d_B.width = B.width; d_B.height = B.height;
size = B.width * B.height * sizeof(float);
d_B.elements = clCreateBuffer(context,
                      CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                      size, B.elements, 0);
// Allocate C in device memory
Matrix d_C;
d_C.width = C.width; d_C.height = C.height;
size = C.width * C.height * sizeof(float);
d_C.elements = clCreateBuffer(context,
                      CL_MEM_WRITE_ONLY, size, 0, 0);
// Invoke kernel
cl_uint i = 0;
clSetKernelArg(matMulKernel, i++,
               sizeof(d_A.width),    (void*)&d_A.width);
clSetKernelArg(matMulKernel, i++,
               sizeof(d_A.height),   (void*)&d_A.height);
```

```
clSetKernelArg(matMulKernel, i++,
               sizeof(d_A.elements), (void*)&d_A.elements);
clSetKernelArg(matMulKernel, i++,
               sizeof(d_B.width),     (void*)&d_B.width);
clSetKernelArg(matMulKernel, i++,
               sizeof(d_B.height),    (void*)&d_B.height);
clSetKernelArg(matMulKernel, i++,
               sizeof(d_B.elements), (void*)&d_B.elements);
clSetKernelArg(matMulKernel, i++,
               sizeof(d_C.width),     (void*)&d_C.width);
clSetKernelArg(matMulKernel, i++,
               sizeof(d_C.height),    (void*)&d_C.height);
clSetKernelArg(matMulKernel, i++,
               sizeof(d_C.elements), (void*)&d_C.elements);
size_t localWorkSize[] = { BLOCK_SIZE, BLOCK_SIZE };
size_t globalWorkSize[] =
             { B.width / dimBlock.x, A.height / dimBlock.y };
clEnqueueNDRangeKernel(queue, matMulKernel, 2, 0,
                        globalWorkSize, localWorkSize,
                        0, 0, 0);
// Read C from device memory
clEnqueueReadBuffer(queue, d_C.elements, CL_TRUE, 0, size,
```

```
                        C.elements, 0, 0, 0);
    // Free device memory
    clReleaseMemObject(d_A.elements);
    clReleaseMemObject(d_C.elements);
    clReleaseMemObject(d_B.elements);
}
```

*Kernel code*:

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.width + col)
typedef struct {
    int width;
    int height;
    __global float* elements;
} Matrix;
// Thread block size
#define BLOCK_SIZE 16
// Matrix multiplication function called by MatMulKernel()
void MatMul(Matrix A, Matrix B, Matrix C)
{
    float Cvalue = 0;
    int row = get_global_id(1);
```

```
    int col = get_global_id(0);
    for (int e = 0; e < A.width; ++e)
        Cvalue += A.elements[row * A.width + e]
                * B.elements[e * B.width + col];
    C.elements[row * C.width + col] = Cvalue;
}
// Matrix multiplication kernel called by MatMulHost()
__kernel void MatMulKernel(
      int Awidth, int Aheight, __global float* Aelements,
      int Bwidth, int Bheight, __global float* Belements,
      int Cwidth, int Cheight, __global float* Celements)
{
    Matrix A = { Awidth, Aheight, Aelements };
    Matrix B = { Bwidth, Bheight, Belements };
    Matrix C = { Cwidth, Cheight, Celements };
    MatMul(A, B, C); // << Error in old NVIDIA documentation
}
```

# Security Considerations

- Paper: You Can Type, but You Can't Hide: A Stealthy GPU-based Keylogger[4]
- GPU rootkit Jellyfish Proof-of-Concept[5]
- GPU keylogger Demon PoC[6]
- ...

[4] http://www.cs.columbia.edu/~mikepo/papers/gpukeylogger.eurosec13.pdf

[5] https://github.com/x0r1/jellyfish

[6] https://github.com/x0r1/Demon

# References I


[1] Khronos OpenCL Working Group, The OpenCL Specification Version 1.1 Rev. 36, September 2010. → PDF

[2] Khronos Group, OpenCL API 1.1 Quick Reference Card Rev. 0610, June 2010. → PDF

[3] Advanced Micro Devices, AMD Accelerated Parallel Processing OpenCL Programming Guide (rev2.7), November 2013. → PDF

[4] NVIDIA Corporation, NVIDIA OpenCL Programming Guide for the CUDA Architecture Version 4.2, March 2012. → PDF

[5] NVIDIA Corporation, NVIDIA OpenCL Best Practices Guide Version 4.2, February 2011. → PDF

Thank you!
Questions?