Basic OpenCL Programming

Pangfeng Liu
National Taiwan University

May 19, 2020

OpenCL

- Open Computing Language (OpenCL) is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors.[1]

[1] http://en.wikipedia.org/wiki/OpenCL

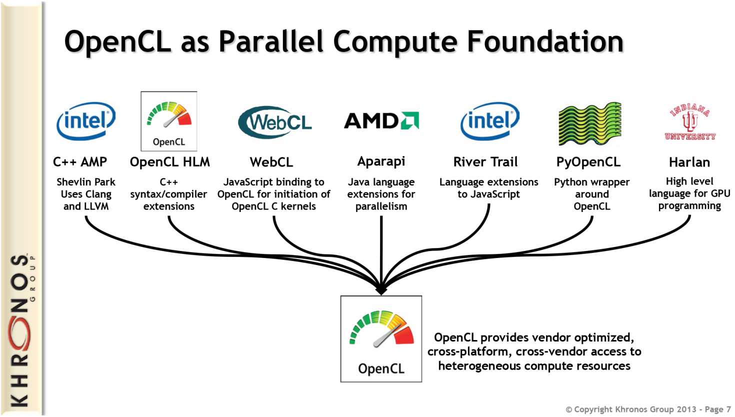
Model

- OpenCL specifies a language (based on C99) for programming these devices and application programming interfaces (APIs) to control the platform and execute programs on the compute devices.
- OpenCL provides parallel computing using task-based and data-based parallelism.

Standard

- OpenCL is an open standard maintained by the non-profit technology consortium Khronos Group.
- Conformant implementations are available from Altera, AMD, Apple, ARM Holdings, Creative Technology, IBM, Imagination Technologies, Intel, Nvidia, Qualcomm, Samsung, Vivante, Xilinx, and ZiiLABS.

OpenCL Vendors



2

[2] http://images.anandtech.com/doci/7161/SPIR5.png

Discussion

- Describe the relation between parallel programming and OpenCL.
- Google “SPIR” and describe its importance and its relation to OpenCL.
- What will happen if the implementation of an “integer” is different among the devices?

GPGPU

- General-purpose computing on graphics processing units (GPGPU) is the use of a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU).[3]

[3] http://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units

- The use of multiple graphics cards in one computer, or large numbers of graphics chips, further parallelizes the already parallel nature of graphics processing.
- In addition, even a single GPU-CPU framework provides advantages that multiple CPUs on their own do not offer due to specialization in each chip.

Advantages

- Fast vector processing for data parallel algorithms on regular data structures.
- High performance-to-price ratio for cost effective high performance computing.
- Leverage the fast growing 3D graphics rendering technology for high performance computing.

Discussion

- Compare the price, clock rate, and performance (measured in floating point operations per second) of GPU and CPU.
- What is the reason that GPU usually deal with "regular computation"? Give your reasoning.

History

- General-purpose computing on GPUs only became practical and popular after ca. 2001, with the advent of both programmable shaders and floating point support on graphics processors.
- In particular, problems involving matrices and/or vectors were easy to translate to a GPU, which acts with native speed and support on those types.

History

- The scientific computing community's experiments with the new hardware started with a matrix multiplication routine (2001); one of the first common scientific programs to run faster on GPUs than CPUs was an implementation of LU factorization (2005).

Pretend to be Pixels

- The early efforts to use GPUs as general-purpose processors required reformulating computational problems in terms of graphics primitives, as supported by the two major APIs for graphics processors, OpenGL and DirectX.
- This cumbersome translation was obviated by the advent of general-purpose programming languages and APIs such as Sh/RapidMind, Brook and Accelerator.

No Need to Pretend

- Nvidia's CUDA allows programmers to ignore the underlying graphical concepts in favor of more common high-performance computing concepts.
- Newer, hardware vendor-independent offerings include Microsoft's DirectCompute and Apple/Khronos Group's OpenCL.
- Modern GPGPU pipelines can act on any "big data" operation and leverage the speed of a GPU without requiring full and explicit conversion of the data to a graphical form.

Discussion

- Find out the full name of CUDA.
- What does the name imply?

Heterogeneity

- Modern computer systems consist of different types of processing units, e.g., CPU and GPU.
- Different processing units have different characteristics.
  - CPU is suitable for complex control, e.g., recursion, complex branching, etc.
  - GPU is suitable for regular computation patterns, e.g., linear algebra and matrix manipulation.

Advantages

- Different processing units for different applications according to their requirements.
- Flexibility for dispatching jobs.
- Energy conservation while maintaining quality of service simultaneously.

Management Issues

- Different processing units has different ISA's and requires different binary codes.
- A single programming model may not be flexible enough to cover all the processing units.
- Serious compiler support for heterogeneity among processing units.

Discussion

- Why does a heterogeneous computing environment is more complicated than a homogeneous one? Give your reasoning.

Heterogeneous System Architecture

- Heterogeneous System Architecture (HSA) is a computer processor architecture that integrates central processing units and graphics processors on the same bus, with shared memory and tasks.[4]
- Heterogeneous computing itself refers to systems that contain multiples processing units central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), or any type of application-specific integrated circuits (ASICs).

[4] http://en.wikipedia.org/wiki/Heterogeneous_System_Architecture

Unified View

- The HSA is being developed by the HSA Foundation, which includes (among many others) AMD and ARM.
- HSA is to reduce communication latency between CPUs, GPUs and other compute devices, and make these various devices more compatible from a programmer's perspective, relieving the programmer of the task of planning the moving of data between devices' disjoint memories, which is done by OpenCL or CUDA right now.

Features

- HSA defines a unified virtual address space space for compute devices: where GPUs traditionally have their own memory, separate from the main (CPU) memory.
- HSA requires these devices to share page tables so that devices can exchange data by sharing pointers, which is to be supported by custom memory management units.
- To render interoperability possible and also to ease various aspects of programming, HSA is intended to be ISA-agnostic for both CPUs and accelerators, and to support high-level programming languages.

Discussion

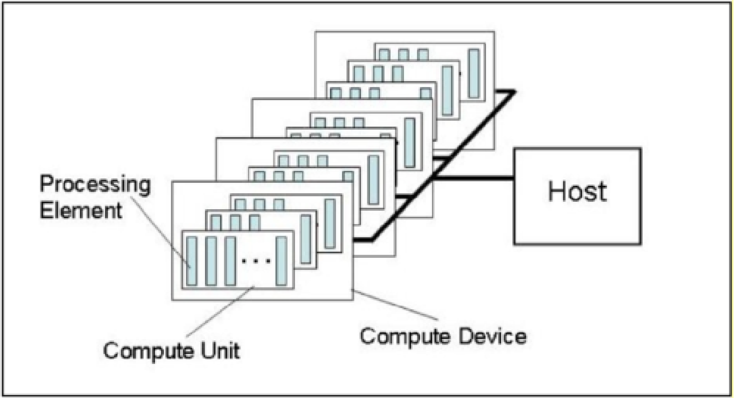
- What do you think are the key issues in the success of HSA? Give your reasoning.

Hierarchy

- An OpenCL computing system consisting of a *host* and *computing resources*.
- The computing resources consist of a number of *compute devices*[5]
- A single compute device typically consists of many *compute units*.
- A single compute unit typically consists of *processing elements* (PEs).

[5] might be central processing units (CPUs) or "accelerators" such as graphics processing units (GPUs).

OpenCL Hardware Model



6

6 http://www.rastergrid.com/blog/wp-content/uploads/2010/11/opencl_platform_model.png

Host

- The host machine is a CPU.
- The computation is dispatched to computing resources for execution.

Computational Resources

1. Compute device 1
    1. Compute unit 1
        1. Processing unit 1
        2. Processing unit 2
        3. ...
    2. Compute unit 2
    3. ...
2. Compute device 2
3. ...

Host v.s. Device

- Host
  - Sequential code written in C/C++
  - For coordination
  - Single threaded
- Device
  - Parallel code written in OpenCL
  - For computation
  - Multi-threaded

Discussion

- Describe the three layer architecture in OpenCL hardware architecture.

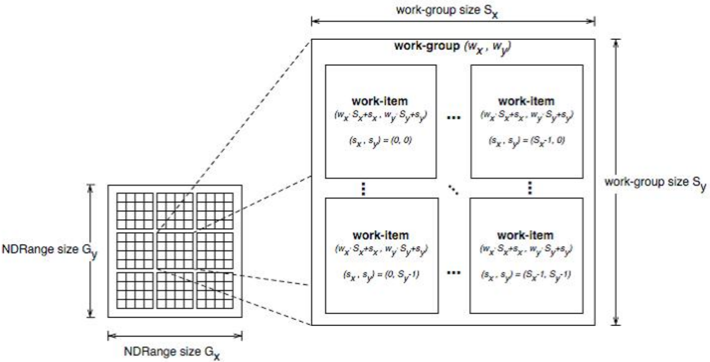
Domain and Kernel

- The computation is defined on a N-dimensional *domain* called `NDRange`.
- You can think of domain as a N-dimensional array, and each array element stores a data.
- We perform *kernel* computation on these array elements in a data parallel fashion.

Hierarchy

- The entire computation domain is divided into *work groups*.
- A work group is divided into *work items*.
- A kernel describes the work to be done for a work item.

Hierarchy



[7]

[7] https://software.intel.com/sites/landingpage/opencl/optimization-guide/OG_files/Basic_Concepts.jpg

Mapping

- The entire computation domain is run on the OpenCL devices.
- A work group is run on a compute unit.
- A work item is run on a compute element.

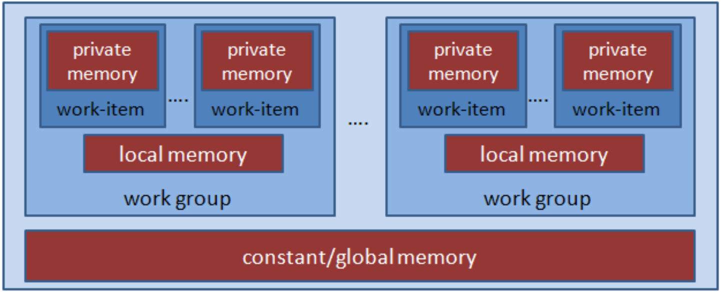
Discussion

- Describe the three layer architecture in OpenCL data domain.

Memory

- The OpenCL memory is consistent with the hardware hierarchy.
- A processing unit has its own private memory.
- Processing units in the same compute unit share a local memory.
- Processing units in the same compute device share a global memory.

Memory



8

---

8 http://1.bp.blogspot.com/-KLUr3r0t0t8/UoIWBwwqSGI/AAAAAAAABzA/WWXaDG4KAkg/s1600/img10.png

Host v.s. Device

- Different memory system from the host.
- Host cannot address the memory on device and device cannot address the memory on host, so memory copying is necessary.

Performance

- The global memory is larger than the local memory, which is larger than the private memory.
- The private memory is faster than the local memory, which is faster than the global memory.

Memory Movement

- Data must be given to the host by I/O operation.
- Data is then transfer from the host to the device global memory for processing.
- Optionally one can move the data from global into local memory for faster access.
- Even more optionally one can move the data from the local memory into Private memory for even faster processing.

Memory Movement

- OK now data processing finishes, you need to move data *all the way back* to the host for visualization.
- This is tedious – hopefully later architecture like HSA will give us a unified, and better, view on memory, from both GPU and CPU.

Mapping

hardware	programming	memory
host	host data	host memory
compute device	computation domain	global memory
compute unit	work group	local memory
processing unit	work item	private memory

Discussion

- Describe the three layer architecture in OpenCL memory view.

Platform and Device

- We start with querying the system to understand its configuration.
- The first thing we do is to know the number of platforms the system has.
- A system can have many platforms, and each platform can have many devices.

Prototype 1: clGetPlatformIDs.h

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

clGetPlatformIDs

`num_entries` The size of platform ids provided by the next parameter `platforms`, where the platform ids will be added into. It must be positive if `platforms` is not `NULL`.

`platforms` An array of `cl_platform_id` found.

`num_platforms` The number of platforms available in the system. `NUll` will be ignored.

clGetPlatformIDs

Example 2: (getPlatformID.c)

```
#include <stdio.h>
#include <assert.h>
#include <CL/cl.h>
#define MAXPLATFORM 5
int main(int argc, char *argv[])
{
  printf("Hello, OpenCL\n");
  cl_platform_id platform_id[MAXPLATFORM];
  cl_uint platform_id_got;
  clGetPlatformIDs(MAXPLATFORM, platform_id,
                   &platform_id_got);
  printf("%d platform found\n", platform_id_got);
  return 0;
}
```

Compilation

- Our platform is an AMD implementation of OpenCL.
- In order to compile the OpenCL on AMD platform we need to specify the directory for the OpenCL headers and library.

Makefile

Example 3: (Makefile)

```
CC = gcc
CFLAGS = -std=c99
CL_TARGET = getPlatformID-cl getPlatformInfo-cl getDeviceID-cl get
PDF_TARGET = buffer.pdf command_queue.pdf context.pdf deviceID.pdf
LIBS = -lOpenCL -lm
all: $(CL_TARGET) $(PDF_TARGET)
%-cl: %.c
	$(CC) $(CFLAGS) $< -o $@ $(LIBS)
%.pdf: %.dot
	dot -Tpdf $< -o $@
tar:
	tar -cvf PP-OpenCL-basic.tar *.c kernel.cl Makefile
clean:
	rm -f $(CL_TARGET)
```

Compilation

`INCDIRS`	Directories for the OpenCL headers.
`LIBS`	Libraries for OpenCL.
`LINKOPT`	Linker options.

Demonstration

- Run the `getPlatformID-cl` program.

Discussion

- How many platform do we have in the system?

Platform Information

- After knowing the number of platform and their ids, we want to know their information.
- OpenCL provides a function `clGetPlatformInfo` for this.

Prototype 4: clGetPlatformInfo.h

```
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);
```

Parameters

- `platform` The id of the platform you want to query.
- `param_name` The property you want to query.
- `param_value_size` The length of `param_value` in bytes.
- `param_value` The location for the query answer.
- `param_value_size_ret` The number of bytes returned from the query.

Request

We may request the following information from a platform.

CL_PLATFORM_PROFILE

CL_PLATFORM_VERSION

CL_PLATFORM_NAME

CL_PLATFORM_VENDOR

CL_PLATFORM_EXTENSIONS

Platform ID

- We first call `clGetPlatformIDs` to get the ids into array `platform_id`.
- The actual number of platforms found will be in `platform_id_got`.
- We then use the platform id to find out its properties.

Example 5: (getPlatformInfo.c)

```
#include <stdio.h>
#include <assert.h>
#include <CL/cl.h>
#define MAXB 256
#define MAXPLATFORM 5
int main(int argc, char *argv[])
{
  printf("hello, OpenCL\n");
  cl_platform_id platform_id[MAXPLATFORM];
  cl_uint platform_id_got;
  clGetPlatformIDs(MAXPLATFORM, platform_id,
                   &platform_id_got);
  printf("%d platform found\n", platform_id_got);
```

```
  for (int i = 0; i < platform_id_got; i++) {
    char buffer[MAXB];
    size_t length;
    clGetPlatformInfo(platform_id[i], CL_PLATFORM_NAME,
                      MAXB, buffer, &length);
    buffer[length] = '\0';
    printf("platform name %s\n", buffer);
    clGetPlatformInfo(platform_id[i], CL_PLATFORM_VENDOR,
                      MAXB, buffer, &length);
    buffer[length] = '\0';
    printf("platform vendor %s\n", buffer);
    clGetPlatformInfo(platform_id[i], CL_PLATFORM_VERSION,
                      MAXB, buffer, &length);
    buffer[length] = '\0';
    printf("OpenCL version %s\n", buffer);
    clGetPlatformInfo(platform_id[i], CL_PLATFORM_PROFILE,
                      MAXB, buffer, &length);
    buffer[length] = '\0';
    printf("platform profile %s\n", buffer);
  }
  return 0;
}
```

Platform Information

- The actual number of platforms found is in `platform_id_got`.
- The platform ids are in `platform_id`.
- We loop through all platform ids and use `clGetPlatformInfo` to retrieve the property we need.
- The id will be in the `buffer` we provided, and the length of query result will be in `length`.
- We are not sure about anything after the specified length so we place a zero character at the end. This may not be necessary.

Platform Property

- We request the name, the vendor, the OpenCL version, and the profile of the platform.
- A full profile means a full implementation of OpenCL.

Demonstration

- Run the `getPlatformInfo-cl` program.

Discussion

- Does the implementation place an zero character at the end of the answer?
- Google `CL_PLATFORM_EXTENSIONS` and find out its meaning.

Device ID and Information

- After knowing platform ids we can find out the ids of the devices in that platform.
- After knowning the device id we can know the information about a device.

Prototype 6: clGetDeviceIDs.h

```
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` The id of the platform you want to query.
- `device_type` The type of device you want to query.
- `num_entries` The number of devices that can be added into the buffer provided by the next parameter `devices`.
- `devices` A pointer to the buffer to store the devices.
- `num_devices` The actual number of devices returned.

Device Types

There are five types of devices one can query.

CL_DEVICE_TYPE_CPU

CL_DEVICE_TYPE_GPU

CL_DEVICE_TYPE_ACCELERATOR

CL_DEVICE_TYPE_DEFAULT

CL_DEVICE_TYPE_ALL

Example 7: (getDeviceID.c)

```
#define MAXB 256
#define MAXPLATFORM 5
#define MAXDEVICE 10
int main(int argc, char *argv[])
{
  printf("Hello, OpenCL\n");
  cl_platform_id platform_id[MAXPLATFORM];
  cl_device_id device_id[MAXDEVICE];
  cl_uint platform_id_got;
  clGetPlatformIDs(MAXPLATFORM, platform_id,
                   &platform_id_got);
  printf("%d platform found\n", platform_id_got);
```

```c
  for (int i = 0; i < platform_id_got; i++) {
    char buffer[MAXB];
    size_t length;
    clGetPlatformInfo(platform_id[i], CL_PLATFORM_NAME,
                      MAXB, buffer, &length);
    buffer[length] = '\0';
    printf("Platform name %s\n", buffer);
    clGetPlatformInfo(platform_id[i], CL_PLATFORM_VENDOR,
                      MAXB, buffer, &length);
    buffer[length] = '\0';
    printf("Platform vendor %s\n", buffer);
    clGetPlatformInfo(platform_id[i], CL_PLATFORM_VERSION,
                      MAXB, buffer, &length);
    buffer[length] = '\0';
    printf("OpenCL version %s\n", buffer);
    clGetPlatformInfo(platform_id[i], CL_PLATFORM_PROFILE,
                      MAXB, buffer, &length);
    buffer[length] = '\0';
    printf("Platform profile %s\n", buffer);
```

```
    cl_device_id devices[MAXDEVICE];
    cl_uint device_id_got;
    clGetDeviceIDs(platform_id[i], CL_DEVICE_TYPE_ALL,
     MAXDEVICE, devices, &device_id_got);
    printf("There are %d devices\n", device_id_got);
    clGetDeviceIDs(platform_id[i], CL_DEVICE_TYPE_CPU,
     MAXDEVICE, devices, &device_id_got);
    printf("There are %d CPU devices\n", device_id_got);
    clGetDeviceIDs(platform_id[i], CL_DEVICE_TYPE_GPU,
     MAXDEVICE, devices, &device_id_got);
    printf("There are %d GPU devices\n", device_id_got);
  }
  return 0;
}
```

Device IDs

- The number of device id returned is in `device_id_got`, and the ids are in `devices`.
- We test all device types, i.e. CPU type, and GPU type.

Demonstration

- Run the `getDeviceID-cl` program.

Discussion

- Does the platform have anything other than CPU and GPU?

clGetDeviceInfo

- After receiving the device ids, we can query the detailed information about a device.
- The procedure is very similar to `clGetPlatformInfo`.

Prototype 8: clGetDeviceInfo.h

```
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);
```

clGetDeviceInfo

- `device` The id of the device you want to query.
- `param_name` The property you want to query.
- `param_value_size` The size of the answer that will be returned.
- `param_value` The buffer for the answer.
- `param_value_size_ret` The actual size of answer returned.

Example 9: (getDeviceInfo.c)

```c
    for (int j = 0; j < device_id_got; j++) {
      clGetDeviceInfo(devices[j], CL_DEVICE_NAME,
                      MAXB, buffer, &length);
      buffer[length] = '\0';
      printf("Device name %s\n", buffer);
      cl_ulong number;
      clGetDeviceInfo(devices[j], CL_DEVICE_GLOBAL_MEM_SIZE,
                      sizeof(cl_ulong), &number, NULL);
      printf("Global memory size %lld\n", (long long)number);
      clGetDeviceInfo(devices[j], CL_DEVICE_LOCAL_MEM_SIZE,
                      sizeof(cl_ulong), &number, NULL);
      printf("Local memory size %lld\n", (long long)number);
      clGetDeviceInfo(devices[j], CL_DEVICE_MAX_COMPUTE_UNITS,
                      sizeof(cl_ulong), &number, NULL);
      printf("# of compute units %lld\n", (long long)number);
      clGetDeviceInfo(devices[j], CL_DEVICE_MAX_WORK_GROUP_SIZE,
                      sizeof(cl_ulong), &number, NULL);
      printf("max # of work items in a work group %lld\n",
             (long long)number);
    }
```

Device Information

- We know the number of devices from the previous `device_id_got`.
- We first query the name of the device, which is given as a character buffer.
- The we query the global memory size, local memory size, the maximum number of compute units, and the maximum number of work groups.
- These numbers are of type `cl_ulong`, which is defined by OpenCL.

Demonstration

- Run the `getDeviceInfo-cl` program.

Discussion

- Google what can be queried from `clGetDeviceInfo`. Make a list of property that you think is useful.

Vector Add

- We will use a simple example to work through the execution of OpenCL program.
- This example will add two vectors with GPU.
- We first query the system to get the platform and device information we need.

Platform and Device

Example 10: (vectorAdd.c)

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

#define MAXGPU 10
#define MAXK 1024
#define N (1024 * 1024)
```

Headers

- We need to include OpenCL header file.
- The length of the vector is `N`.
- The maximum length of the kernel source is `MAXK`

Platform and Device

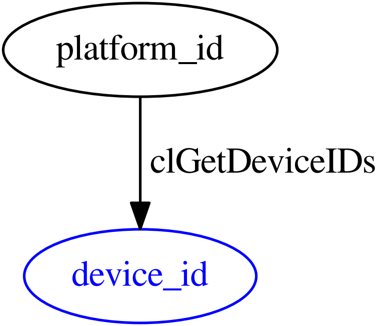
Example 11: (vectorAdd.c)

```
int main(int argc, char *argv[])
{
  printf("Hello, OpenCL\n");
  cl_int status;
  cl_platform_id platform_id;
  cl_uint platform_id_got;
  status = clGetPlatformIDs(1, &platform_id,
                            &platform_id_got);
  assert(status == CL_SUCCESS && platform_id_got == 1);
  printf("%d platform found\n", platform_id_got);
  cl_device_id GPU[MAXGPU];
  cl_uint GPU_id_got;
  status = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU,
                          MAXGPU, GPU, &GPU_id_got);
  assert(status == CL_SUCCESS);
  printf("There are %d GPU devices\n", GPU_id_got);
```

Platform and Device

- We then get only one device, and all of its GPUs.
- Note that we need to check the return status.

Platform and Device



Discussion

- Read the code and identify the part that we will use only GPU.

clCreateContext

- After knowing the platform and device information, we can build a *context* to run our OpenCL applications.
- The context consists mostly the devices that will participate this computation.

Prototype 12: clCreateContext.h

```
cl_context
clCreateContext(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);
```

Parameters

- `properties` A list of properties for this context.
- `num_devices` The number of devices this context will use. The devices are in the next parameter `devices`.
- `devices` The devices used by this context.
- `pfn_notify` A callback routine for error.
- `user_data` A parameter that will be supplied to the callback routine `pfn_notify`.
- `errcode_ret` The error code.

Simplicity

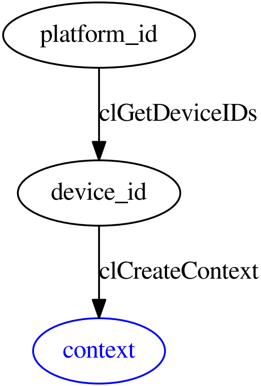
- For simplicity in this lecture we will not use callback function and property.

Context

Example 13: (vectorAdd.c)

```
cl_context context =
  clCreateContext(NULL, GPU_id_got, GPU, NULL, NULL,
                  &status);
assert(status == CL_SUCCESS);
```

Context


platform_id
clGetDeviceIDs
device_id
clCreateContext
context

Discussion

- Read the code and identify the part that the context we build uses all GPUs.

Command Queue

- Within a context the kernel will start execution and send commands to devices.
- We need to build a command queue from the host to a device so that the command from the kernel can be sent there.
- A command queue connects to a device.

Prototype 14: clCreateCommandQueue.h

```
cl_command_queue
clCreateCommandQueueWithProperties(
          cl_context context,
          cl_device_id device,
          const cl_queue_properties
          *properties,
          cl_int *errcode_ret);
```

Parameters

- `context` The context this command queue belongs to.
- `device` The device connected by this command queue. It has to be one of the devices in the context.
- `properties` A pointer to the properties.
- `errcode_ret` The error code.

Simplicity

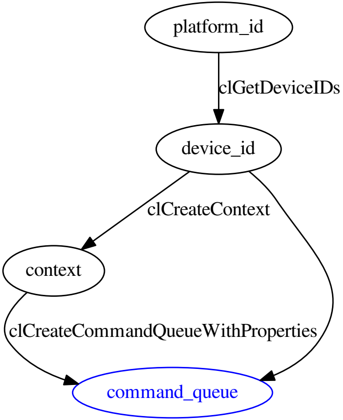
- For simplicity in this lecture we will not set property.
- Note that the property is a pointer so we can set it to NULL.

Command Queue

Example 15: (vectorAdd.c)

```
cl_command_queue commandQueue =
  clCreateCommandQueueWithProperties(context, GPU[0], NULL, &sta
assert(status == CL_SUCCESS);
```

Command Queue



Discussion

- Which GPU does this command queue connect to?

Kernel Program

- We must specify the computation as a kernel.
- The source of a kernel is in the form of string, not file.
- OpenCL provides functions to build kernel program from strings.
- For ease of kernel modification we will place the kernel source into a file "kernel.cl".

Prototype 16: clCreateProgramWithSource.h

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

Parameters

`context` The context for which to build the program.

`count` The number of strings in the program.

`strings` A pointer array to store the strings. That is, your kernel program may consist of many strings.

`lengths` An array of string lengths for the pointer array `strings`.

`errcode_ret` The error code.

Kernel Source

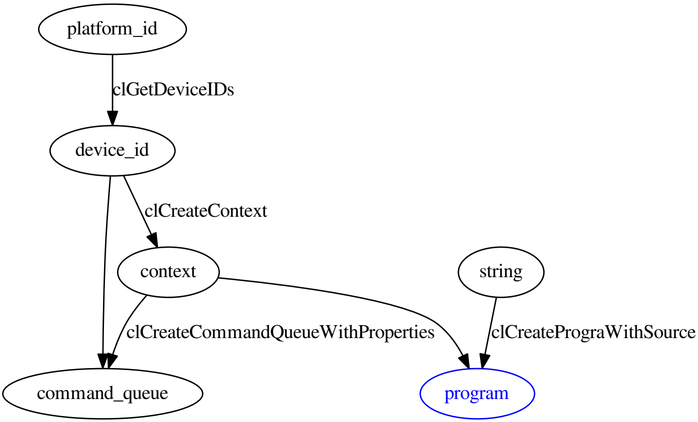
Example 17: (vectorAdd.c)

```
FILE *kernelfp = fopen(argv[1], "r");
assert(kernelfp != NULL);
char kernelBuffer[MAXK];
const char *constKernelSource = kernelBuffer;
size_t kernelLength =
  fread(kernelBuffer, 1, MAXK, kernelfp);
printf("The size of kernel source is %zu\n", kernelLength);
cl_program program =
  clCreateProgramWithSource(context, 1, &constKernelSource,
                            &kernelLength, &status);
assert(status == CL_SUCCESS);
```

From File to String

- We store our kernel in a file "kernel.cl".
- We read the contents of "kernel.cl" into a buffer `kernelBuffer` with fread. We also know the length of the file by the return value of fread.
- The we call `clCreateProgramWithSource` with the `kernelBuffer`. Note that we only have one string.

Program


platform_id
clGetDeviceIDs
device_id
clCreateContext
context
string
clCreateCommandQueueWithProperties
clCreatePrograWithSource
command_queue
program

Discussion

- Identify the part of the code that determines the length of the kernel string.

Compile and Build

- After building your program object from a string, you need to compile and link it.
- OpenCL provides a `clBuildProgram` to do this.

Prototype 18: clBuildProgram.h

```
l_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);
```

Parameters

- `program` The program to be build.
- `num_devices` The number of devices to be used in `device_list`.
- `device_list` The device list.
- `options` The build option to be used.
- `pfn_notify` The callback function for error.
- `user_data` The parameter supplied to `pfn_notify`.

Build Program

Example 19: (vectorAdd.c)

```
status =
  clBuildProgram(program, GPU_id_got, GPU, NULL, NULL,
                 NULL);
assert(status == CL_SUCCESS);
printf("Build program completes\n");
```

Discussion

- Identify the GPUs for which we have compiled and built the program.

Kernel

- After building the program object, we are ready to build kernel.
- We only need the program object to do this.

Prototype 20: clCreateKernel.h

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

Parameters

`program` The kernel program we built.

`kernel_name` The name of the kernel function.

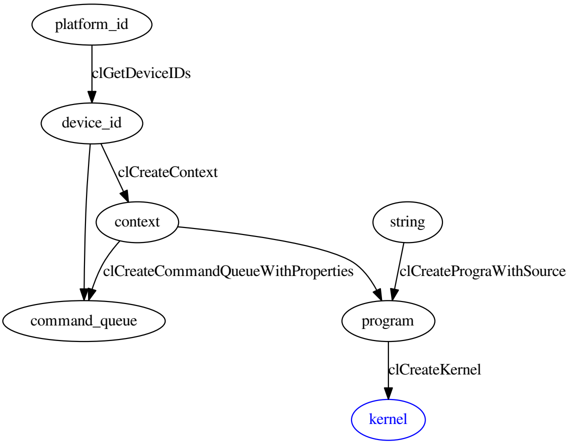
`errcode_ret` Error code.

Create Kernel

Example 21: (vectorAdd.c)

```
    cl_kernel kernel = clCreateKernel(program, "add", &status);
    assert(status == CL_SUCCESS);
    printf("Build kernel completes\n");
```

Kernel


platform_id
clGetDeviceIDs
device_id
clCreateContext
context
string
clCreateCommandQueueWithProperties
clCreatePrograWithSource
command_queue
program
clCreateKernel
kernel

Discussion

- Look into file kernel.cl and make sure that the kernel function name there matches the name we provided while creating the kernel.

Vectors

- We now prepare the input vectors `A` and `B`, and the output vector `C` in host memory.
- We place them on heap and properly initialize them.

Vector

Example 22: (vectorAdd.c)

```
cl_uint* A = (cl_uint*)malloc(N * sizeof(cl_uint));
cl_uint* B = (cl_uint*)malloc(N * sizeof(cl_uint));
cl_uint* C = (cl_uint*)malloc(N * sizeof(cl_uint));
assert(A != NULL && B != NULL && C != NULL);

for (int i = 0; i < N; i++) {
  A[i] = i;
  B[i] = N - i;
}
```

Discussion

- Observe the code and determine the correct value of `C`.

Buffer

- Devices cannot access the memory of the host directly.
- After the host receives the input from I/O, it needs to create a buffer object and link it with the host memory the data is in, so the device can access the data.
- OpenCL provide this linkage by a pointer to `cl_mem`, i.e., a buffer.

Prototype 23: clCreateBuffer.h

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

Parameters

`context` The context this buffer belongs to.

`flags` Properties for this buffer.

`size` The size of the buffer.

`host_ptr` The memory on the host this buffer refers to.

`errcode_ret` Error code.

Buffer Property

We can set the following property for a buffer.

CL_MEM_READ_ONLY The buffer is read only.

CL_MEM_WRITE_ONLY The buffer is write only.

CL_MEM_READ_WRITE We can read and write the buffer

CL_MEM_USE_HOST_PTR OpenCL uses the host memory as the buffer.

CL_MEM_COPY_HOST_PTR OpenCL copies the contents of the host memory into a buffer accessible from GPU.

Create Buffer

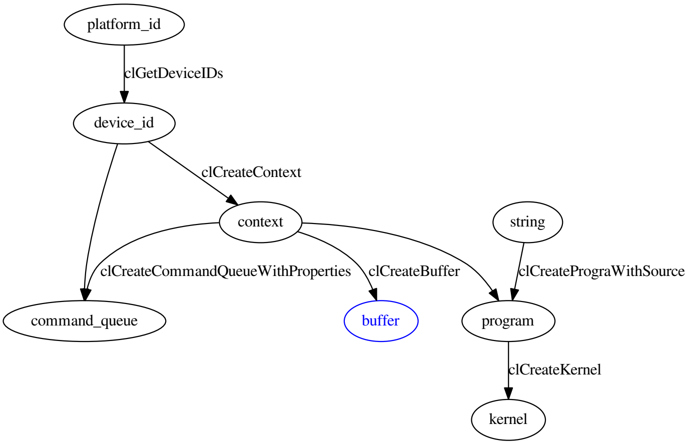
- We set the buffer for `A` and `B` to be read only, and the buffer for `C` to be write only.
- Note that we need to check the status for errors.

Vector

Example 24: (vectorAdd.c)

```
  cl_mem bufferA =
    clCreateBuffer(context,
                   CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                   N * sizeof(cl_uint), A, &status);
  assert(status == CL_SUCCESS);
  cl_mem bufferB =
    clCreateBuffer(context,
                   CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                   N * sizeof(cl_uint), B, &status);
  assert(status == CL_SUCCESS);
  cl_mem bufferC =
    clCreateBuffer(context,
                   CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
                   N * sizeof(cl_uint), C, &status);
  assert(status == CL_SUCCESS);
  printf("Build buffers completes\n");
```

Buffer


platform_id
clGetDeviceIDs
device_id
clCreateContext
context
string
clCreateCommandQueueWithProperties
clCreateBuffer
clCreatePrograWithSource
command_queue
buffer
program
clCreateKernel
kernel

Discussion

- Observe the flag we give to each buffer. Why some of them are read only and some of them are write only?

Parameter Linking

- Now we want to connect the parameters kernel will see with the buffers we provide.
- This is set by the order the parameters appearing in the prototype. That is, we need to make sure the order is A, B, then C. Please refer to the following kernel source.

Example 25: (kernel.cl)

```
__kernel void add(__global int* matrixA,
                  __global int* matrixB,
                  __global int* matrixC)
{
        int idx = get_global_id(0);
        matrixC[idx] = matrixA[idx] + matrixB[idx];
}
```

Prototype 26: clSetKernelArg.h

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

Parameters

`kernel`	The kernel to set the argument.
`arg_index`	the index of the argument (staring from 0).
`arg_size`	The location of the argument.
`arg_value`	The size of the argument.

Mapping

Example 27: (vectorAdd.c)

```
  status = clSetKernelArg(kernel, 0, sizeof(cl_mem),
                          (void*)&bufferA);
  assert(status == CL_SUCCESS);
  status = clSetKernelArg(kernel, 1, sizeof(cl_mem),
                          (void*)&bufferB);
  assert(status == CL_SUCCESS);
  status = clSetKernelArg(kernel, 2, sizeof(cl_mem),
                          (void*)&bufferC);
  assert(status == CL_SUCCESS);
  printf("Set kernel arguments completes\n");
```

Discussion

- Observe the code and make sure that the order of parameters is correct.

Shape of Data

- Now we need to define the shape of the data domain we will work on.
- First we determine the dimension of the data domain.
- Then we determine the size of each dimension of the global domain, and the size of each dimension of the a work group.
- These two information also determine the number of work groups.

Prototype 28: clEnqueueNDRangeKernel.h

```
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);
```

Parameters

`command_queue` The commandqueue to send the kernel code.

`kernel` the kernel to run.

`work_dim` The dimension of work items.

`global_work_offset` NULL for now.

`global_work_size` An array of `work_dim` integers that specify the dimension of the global data domain.

Parameters

`local_work_size` An array of `work_dim` integers that specify dimensions of a work group.

`num_events_in_wait_list` The number of events this computation must wait for completion.

`event_wait_list` The list of events to wait for.

`event` Returns an event the describes the result of this computation.

Data Shape

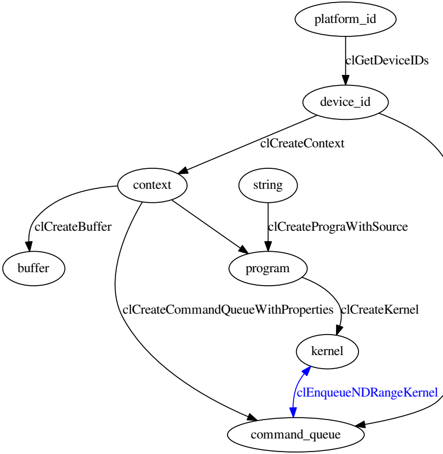
Example 29: (vectorAdd.c)

```
size_t globalThreads[] = {(size_t)N};
size_t localThreads[] = {1};
status =
  clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL,
                         globalThreads, localThreads,
                         0, NULL, NULL);
assert(status == CL_SUCCESS);
printf("Specify the shape of the domain completes.\n");
```

Domain Shape

- We set the domain as one dimensional. The size is `N`.
- The size of a work group is set to 1, so there will be N work groups. Each work group has one work item.

NDRange


platform_id
clGetDeviceIDs
device_id
clCreateContext
context
string
clCreateBuffer
clCreatePrograWithSource
buffer
program
clCreateCommandQueueWithProperties
clCreateKernel
kernel
clEnqueueNDRangeKernel
command_queue

Discussion

- How many work groups do we have?
- How many work items do we have?

Get Result

- Finally we need to get the result from buffer back to host array `C`
- We accomplish this by placing a read buffer command into the command queue.
- When the GPU runs this command the buffer will be copied into `C`.

Prototype 30: clEnqueueReadBuffer.h

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

Parameters

- `command_queue` The command queue to place the read buffer command.
- `buffer` The buffer to read.
- `blocking_read` If the read is blocking.
- `offset` The offset in buffer to read.
- `cb` The amount of data to read.
- `ptr` The location of host memory the data will be read into.

Parameters

`num_events_in_wait_list` The number of events this computation must wait for completion.

`event_wait_list` The list of events to wait for.

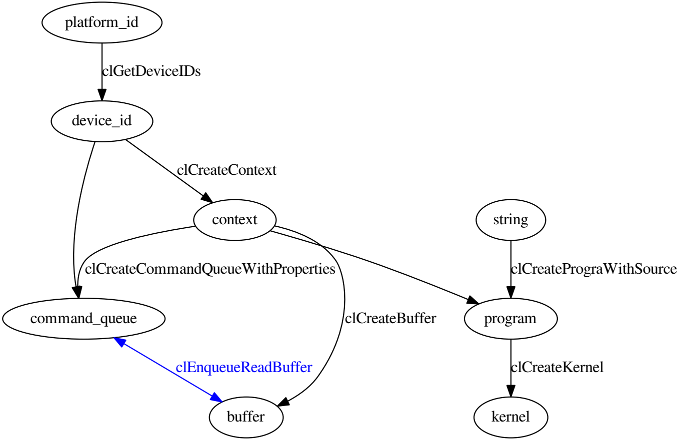
`event` Returns an event the decibels the result of this computation.

Get Results

Example 31: (vectorAdd.c)

```
clEnqueueReadBuffer(commandQueue, bufferC, CL_TRUE,
                    0, N * sizeof(cl_uint), C,
                    0, NULL, NULL);
printf("Kernel execution completes.\n");
```

Read Buffer


platform_id
clGetDeviceIDs
device_id
clCreateContext
context
string
clCreateCommandQueueWithProperties
clCreateProgramWithSource
command_queue
clCreateBuffer
program
clEnqueueReadBuffer
clCreateKernel
buffer
kernel

Receive Results

- We set the mode to be blocking because we can only check the result when all data are ready.
- We read the data from the *beginning* so we set the offset to 0.

Discussion

- Which GPU does this command queue connect to?

Check

- Finally the host checks the correctness of the computation.
- The host also releases the objects it created.
  - The buffers `A`, `B`, and `C`.
  - Other objects related to OpenCL.
    - context
    - program
    - kernel
    - command queue
    - buffers

Release Object

The object releasing API is as follow. The only parameter is the object you want to release.

- `clReleaseContext`
- `clReleaseCommandQueue`
- `clReleaseProgram`
- `clReleaseKernel`
- `clReleaseMemObject`

Get Results

Example 32: (vectorAdd.c)

```
  for (int i = 0; i < N; i++)
    assert(A[i] + B[i] == C[i]);

  free(A);                        /* host memory */
  free(B);
  free(C);
  clReleaseContext(context);      /* context etcmake */
  clReleaseCommandQueue(commandQueue);
  clReleaseProgram(program);
  clReleaseKernel(kernel);
  clReleaseMemObject(bufferA);    /* buffers */
  clReleaseMemObject(bufferB);
  clReleaseMemObject(bufferC);
  return 0;
}
```

Demonstration

- Run the `vectorAdd-cl` program.

Discussion

- Have we released all objects we created?