This tutorial's code is on the source control.

OpenCL Tutorials 1 - Quickstart

Hi all,
this is the first OpenCL real tutorial. Here I will not approach technical concepts of the GPU architecture and performance guidelines. Instead, the focus will be OpenCL API basics, trying to cover the basic management part using a small kernel as example.

Ok, the first thing you should know about OpenCL is that OpenCL programs are divided in two parts: one that executes on the **device** (in our case, on the GPU) and other that executes on the **host** (in our case, the CPU). The device program is the one you may be concerned about. It's where the OpenCL magic happens. In order to execute code on the device, programmers can write special functions (called **kernels**), which are coded with the OpenCL Programming Language - a sort of C with some restrictions and special keywords and data types.
On the other hand, the host program offers an API so that you can manage your device execution. The host can be programmed in C or C++ and it controls the OpenCL environment (context, command-queue,...).

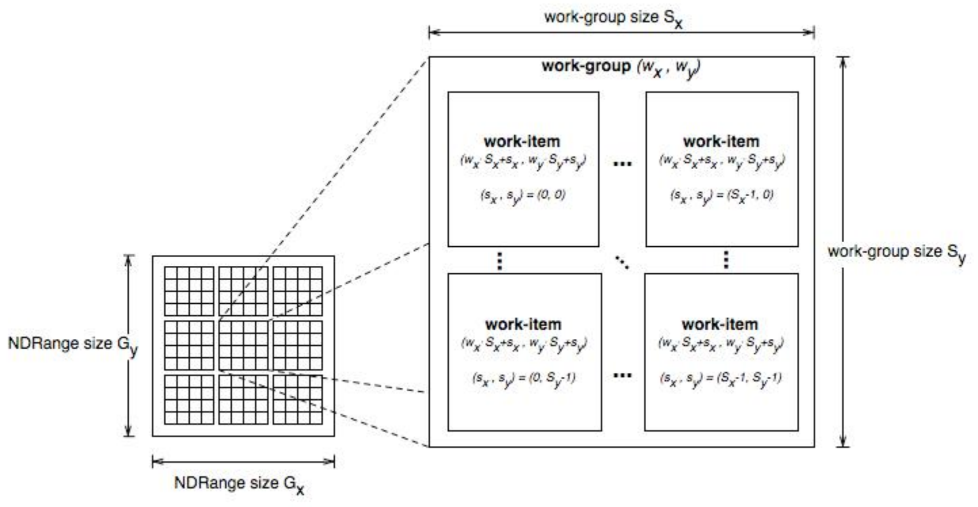
1. Device

Let's talk a bit about the device. The device, as mentioned above, is where most of the efforts on OpenCL programming will be made.
By now, we must know just a few basic concepts:

- **Kernel:** you can think on a kernel as a function that is executed on the device. There are other functions that can be executed on the device, as we'll see on the next tutorials, but there's a slight difference between these functions and kernels: kernels are **entry points** to the device program. In other words, kernel are the only functions that can be **called from the host**.

The question now is: how do I program a Kernel? How do I express the parallelism with kernels? What's the execution model? This leads us to the next concepts:

- **SIMT:** SIMT stands for SINGLE INSTRUCTION MULTIPLE THREAD and it reflects how instructions are executed in the host. As the name suggests, the same code is executed in parallel by a different thread, and each **thread** executes the code with different data.
- **Work-item:** the work-items are equivalent to the **CUDA threads**, and are the smallest execution entity. Every time a Kernel is launched, lots of *work-items* (a number specifyed by the programmer) are launched, each one executing the same code. Each work-item has an ID, which is accessible from the kernel, and which is used to distinguish the data to be processessed by each work-item.
- **Work-group:** work-groups exist to allow communication and cooperation between work-items. They reflect how work-items are organized (it's a N-dimensional grid of work-groups, with N = 1, 2 or 3). Work-groups are equivalent to **CUDA thread blocks**. As work-items, work-groups also have an unique ID that can be reffered from the kernel.
- **ND-Range:** the ND-Range is the next organization level, specifying how work-groups are organized (again, as a N-dimensional grid of work-groups, N = 1, 2 or 3);


work-group size S_x
work-group (w_x, w_y)
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, 0)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, 0)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, S_y-1)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, S_y-1)$
work-group size S_y
NDRange size G_y
NDRange size G_x

This is an example of the NDRange organization.

This is an example of the NDRange organization.

1.1 Kernel

Ok, time to program our first kernel. Let's start with a small one, a kernel that **adds two vectors**. This kernel should take four parameters: two vectors to be added, another to store the result, and the vectors size.If you write a program that solves this problem on the **CPU** it will be something like this:

```
void vector_add_cpu (const float* src_a,
                     const float* src_b,
                     float*  res,
                     const int num)
{
   for (int i = 0; i < num; i++)
      res[i] = src_a[i] + src_b[i];
}
```

However, on the GPU the logic would be slightly different. Instead of having one thread iterating through all elements, we could have each thread computing one element, which index is the same of the thread. Let's see the code:

```
__kernel void vector_add_gpu (__global const float* src_a,
                     __global const float* src_b,
                     __global float* res,
                   const int num)
{
   /* get_global_id(0) returns the ID of the thread in execution.
   As many threads are launched at the same time, executing the same kernel,
   each one will receive a different ID, and consequently perform a different
computation.*/
   const int idx = get_global_id(0);

   /* Now each work-item asks itself: "is my ID inside the vector's range?"
   If the answer is YES, the work-item performs the corresponding computation*/
   if (idx < num)
      res[idx] = src_a[idx] + src_b[idx];
}
```

A few things you should have noticed about this code are:

- The **kernel** reserved word, which specifies that the function is a kernel. Kernel functions must **always** return void;
- The **global** reserved word in front of the parameters. This specifies where the argument's memory is and will be explored in the next tutorials.

Additionally, all kernels must be in ".cl" files, and ".cl" files must contain only OpenCL code.

2. Host

Our kernel is already coded, let's program the host now.

2.1. Creating the basic OpenCL run-time environment

There are some elements we must get familiar with:

- **Platform**: "The host plus a collection of devices managed by the OpenCL framework that allow an application to share resources and execute kernels on devices in the platform." Platforms are represented by a *cl_platform* object, which can be initialized using the following function:

```
// Returns the error code
cl_int oclGetPlatformID (cl_platform_id *platforms)   // Pointer to the
platform object
```

- **Device:** are represented by *cl_device* objects, initialized with the following function.

```
// Returns the error code
cl_int clGetDeviceIDs  (cl_platform_id platform,
           cl_device_type device_type,   // Bitfield identifying the type. For
the GPU we use CL_DEVICE_TYPE_GPU
           cl_uint num_entries,   // Number of devices, typically 1
           cl_device_id *devices,   // Pointer to the device object
           cl_uint *num_devices)   // Puts here the number of devices matching
the device_type
```

- **Context:** defines the entire OpenCL environment, including OpenCL kernels, devices, memory management, command-queues, etc. Contexts in OpenCL are referenced by an *cl_context* object, which must be initialized using the following function:

```
// Returns the context
// Returns the context
cl_context clCreateContext (const cl_context_properties *properties,   //
Bitwise with the properties (see specification)
                cl_uint num_devices,   // Number of devices
                const cl_device_id *devices,   // Pointer to the devices object
                void (*pfn_notify)(const char *errinfo, const void
*private_info, size_t cb, void *user_data),   // (don't worry about this)
                void *user_data,   // (don't worry about this)
                cl_int *errcode_ret)   // error code result
```

- **Command-Queue:** the OpenCL command-queue, as the name may suggest, is an object where OpenCL commands are enqueued to be executed by the device. "The command-queue is created on a specific device in a context [...] Having multiple command-queues allows applications to queue multiple independent commands without requiring synchronization." (OpenCL Specification).

```
cl_command_queue clCreateCommandQueue (cl_context context,
                cl_device_id device,
                cl_command_queue_properties properties,   // Bitwise with the
properties
                cl_int *errcode_ret)   // error code result
```

The following examples shows a practical usage of these elements:

```
cl_int error = 0;   // Used to handle error codes
cl_platform_id platform;
cl_context context;
cl_command_queue queue;
cl_device_id device;

// Platform
error = oclGetPlatformID(&platform);
if (error != CL_SUCCESS) {
   cout << "Error getting platform id: " << errorMessage(error) << endl;
   exit(error);
}
// Device
error = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
if (err != CL_SUCCESS) {
   cout << "Error getting device ids: " << errorMessage(error) << endl;
   exit(error);
}
// Context
context = clCreateContext(0, 1, &device, NULL, NULL, &error);
if (error != CL_SUCCESS) {
   cout << "Error creating context: " << errorMessage(error) << endl;
   exit(error);
}
// Command-queue
queue = clCreateCommandQueue(context, device, 0, &error);
if (error != CL_SUCCESS) {
   cout << "Error creating command queue: " << errorMessage(error) << endl;
   exit(error);
}
```

2.2. Allocating Memory

Ok, the basic host is configured. What about the memory management? To execute our small kernel we need to allocate 3 vectors and initialize at least 2 of them. Let's see how to do this.
To perform the same operation on the host, we would do something like this:

```
const int size = 1234567
float* src_a_h = new float[size];
float* src_b_h = new float[size];
float* res_h = new float[size];
// Initialize both vectors
for (int i = 0; i < size; i++) {
   src_a_h = src_b_h = (float) i;
}
```

To represent memory allocated on the device we use the *cl_mem* type. To allocate memory we use:

```
// Returns the cl_mem object referencing the memory allocated on the device
cl_mem clCreateBuffer (cl_context context,   // The context where the memory
will be allocated
          cl_mem_flags flags,
          size_t size,   // The size in bytes
          void *host_ptr,
          cl_int *errcode_ret)
```

Where *flags* is a bitwise and the options are:

- CL_MEM_READ_WRITE
- CL_MEM_WRITE_ONLY
- CL_MEM_READ_ONLY
- CL_MEM_USE_HOST_PTR
- CL_MEM_ALLOC_HOST_PTR
- CL_MEM_COPY_HOST_PTR - copies the memory pointed by *host_ptr*

Using this function we get the following code:

```
const int mem_size = sizeof(float)*size;
// Allocates a buffer of size mem_size and copies mem_size bytes from src_a_h
cl_mem src_a_d = clCreateBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, mem_size, src_a_h, &error);
cl_mem src_b_d = clCreateBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_COPY_HOST_PTR, mem_size, src_b_h, &error);
cl_mem res_d = clCreateBuffer(context, CL_MEM_WRITE_ONLY, mem_size, NULL,
&error);
```

2.3. The Program and the Kernels

By now you may be asking yourself questions like these: *How do we call kernels? How does the compiler knows how to put the code on the device? Moreover, how do we compile kernels?*.
Here comes a confusing concept when we are beginning: **OpenCL Programs** vs **OpenCL Kernels**.

- **Kernel:** you should already know, as it's explained up in the tutorial, that kernels are essentially functions that we can call from the host and that will run on the device. What you maybe don't know is that **kernels are compiled at run-time**! More generally, every code that runs on the device, what includes kernels and non-kernel functions called by kernels, are compiled at run-time. This introduces the next concept, the concept of program:
- **Program:** an OpenCL Program is formed by a set of kernels, functions and declarations, and it's represented by an cl_program object. When creating a program, you must specify which files compose the program, and then compile it.

To create a program you may use the following function:

```
// Returns the OpenCL program
cl_program clCreateProgramWithSource (cl_context context,
                  cl_uint count,   // number of files
                  const char **strings,   // array of strings, each one is a
file
                  const size_t *lengths,   // array specifying the file lengths
                  cl_int *errcode_ret)   // error code to be returned
```

After we create the program, we can compile it using:

```
cl_int clBuildProgram (cl_program program,
                  cl_uint num_devices,
                  const cl_device_id *device_list,
                  const char *options,   // Compiler options, see the
specifications for more details
                  void (*pfn_notify)(cl_program, void *user_data),
                  void *user_data)
```

To view the compile log (warnings, errors, ...) we must use the following:

```
cl_int clGetProgramBuildInfo (cl_program program,
                cl_device_id device,
                cl_program_build_info param_name,   // The parameter we want to
know
                size_t param_value_size,
                void *param_value,    // The answer
                size_t *param_value_size_ret)
```

Finally we can "extract" the entry points to the program. This is done creating cl*kernel* objects:

```
cl_kernel clCreateKernel (cl_program program,   // The program where the kernel
is
         const char *kernel_name,   // The name of the kernel, i.e. the name of
the kernel function as it's declared in the code
         cl_int *errcode_ret)
```

Notice that we can have several OpenCL programs and several kernels related to each one.

The code related to this small chapter is the following:

```
// Creates the program
// Uses NVIDIA helper functions to get the code string and it's size (in bytes)
size_t src_size = 0;
const char* path = shrFindFilePath("vector_add_gpu.cl", NULL);
const char* source = oclLoadProgSource(path, "", &src_size);
cl_program program = clCreateProgramWithSource(context, 1, &source, &src_size,
&error);
assert(error == CL_SUCCESS);

// Builds the program
error = clBuildProgram(program, 1, &device, NULL, NULL, NULL);
assert(error == CL_SUCCESS);

// Shows the log
char* build_log;
size_t log_size;
// First call to know the proper size
clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, 0, NULL,
&log_size);
build_log = new char[log_size+1];
// Second call to get the log
clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG, log_size,
build_log, NULL);
build_log[log_size] = '\0';
cout << build_log << endl;
delete[] build_log;

// Extracting the kernel
cl_kernel vector_add_kernel = clCreateKernel(program, "vector_add_gpu",
&error);
assert(error == CL_SUCCESS);
```

2.4. Launching the Kernel

Once our kernel object exists, we can now launch our kernel.
First of all, we must **set up the arguments**. This is done using:

```
cl_int  clSetKernelArg (cl_kernel kernel,   // Which kernel
          cl_uint arg_index,   // Which argument
          size_t arg_size,   // Size of the next argument (not of the value
pointed by it!)
          const void *arg_value)    // Value
```

This function should be called for each argument.

After all arguments are enqueued, we can call the kernel using:

```
cl_int  clEnqueueNDRangeKernel (cl_command_queue command_queue,
                            cl_kernel kernel,
                            cl_uint  work_dim,    // Choose if we are using 1D,
2D or 3D work-items and work-groups
                            const size_t *global_work_offset,
                            const size_t *global_work_size,   // The total
number of work-items (must have work_dim dimensions)
                            const size_t *local_work_size,     // The number of
work-items per work-group (must have work_dim dimensions)
                            cl_uint num_events_in_wait_list,
                            const cl_event *event_wait_list,
                            cl_event *event)
```

This chapter's code is the following:

```
// Enqueuing parameters
// Note that we inform the size of the cl_mem object, not the size of the
memory pointed by it
error = clSetKernelArg(vector_add_k, 0, sizeof(cl_mem), &src_a_d);
error |= clSetKernelArg(vector_add_k, 1, sizeof(cl_mem), &src_b_d);
error |= clSetKernelArg(vector_add_k, 2, sizeof(cl_mem), &res_d);
error |= clSetKernelArg(vector_add_k, 3, sizeof(size_t), &size);
assert(error == CL_SUCCESS);

// Launching kernel
const size_t local_ws = 512;    // Number of work-items per work-group
// shrRoundUp returns the smallest multiple of local_ws bigger than size
const size_t global_ws = shrRoundUp(local_ws, size);    // Total number of
work-items
error = clEnqueueNDRangeKernel(queue, vector_add_k, 1, NULL, &global_ws,
&local_ws, 0, NULL, NULL);
assert(error == CL_SUCCESS);
```

2.5. Reading back

Read back is easy. Analogously to what we've done to write to memory, now we must enqueue a read buffer operation, which is done using:

```
cl_int  clEnqueueReadBuffer (cl_command_queue command_queue,
                      cl_mem buffer,   // from which buffer
                      cl_bool blocking_read,   // whether is a blocking or non-
blocking read
                      size_t offset,   // offset from the beginning
                      size_t cb,   // size to be read (in bytes)
                      void *ptr,   // pointer to the host memory
                      cl_uint num_events_in_wait_list,
                      const cl_event *event_wait_list,
                      cl_event *event)
```

The usage of this function is this:

```
// Reading back
float* check = new float[size];
clEnqueueReadBuffer(queue, res_d, CL_TRUE, 0, mem_size, check, 0, NULL, NULL);
```

2.5. Cleaning up

Being good programmers we could never think about not cleaning up memory, right!
The basic thing you must know is: everything allocated with clCreate... (buffers, kernels, queues, ...) must be destroyed with clRelease...

The code is the following:

```
// Cleaning up
delete[] src_a_h;
delete[] src_b_h;
delete[] res_h;
delete[] check;
clReleaseKernel(vector_add_k);
clReleaseCommandQueue(queue);
clReleaseContext(context);
clReleaseMemObject(src_a_d);
clReleaseMemObject(src_b_d);
clReleaseMemObject(res_d);
```

That's all folks, please do not hesitate to contact me if you have any doubt.

Last edited Jul 19 2010 at 5:04 PM by bjurkovski, version 28

Comments

guoshuai002011 Oct 30 2011 at 4:16 AM
Hi,Please continue! I need this, deadly! Thanks!

eLRuLL Jun 21 2011 at 11:33 PM
one question, does the host part of this program go in the .cpp file???

cqq Apr 1 2011 at 10:52 AM
Their's a error:
cl_int clSetKernelArg (cl_kernel kernel, // Which kernel
cl_uint arg_index, // Which argument
size_t arg_size, // Size of the next argument (not of the value pointed by it!)
const void *arg_value) // Value

the arg_size should be the size of the current argument, pointed by arg_value.

MJLHThomassen Oct 27 2010 at 2:36 PM

Is this series going to continue?

alien_kill Apr 14 2010 at 4:12 PM
Muito bom:) Por favor continue!

Sign in to add a comment