Using the PyOpenCL module

Open Computing Language (**OpenCL**) is a framework used to develop programs that work across heterogeneous platforms, which can be made either by the CPU or GPU that are produced by different manufacturers. This platform was created by Apple, but has been developed and maintained by a non-profit consortium called the Khronos Group. This framework is the main alternative for the CUDA execution of software on a GPU, but has a point of view that is diametrically opposed. However, CUDA makes specialization its strong point (produced, developed, and compatible with NVIDIA), ensuring excellent performance at the expense of portability. OpenCL offers a solution compatible with nearly all devices on the market. Software written in OpenCL can run on processor products from all major industries, such as Intel, NVIDIA, IBM, and AMD. OpenCL includes a language to write kernels based on C99 (with some restrictions), allowing you to use the hardware available directly in the same way as with CUDA-C-Fortran or CUDA. OpenCL provides functions to run highly parallel and synchronization primitives, such as indicators for regions of memory and control mechanisms for the different platforms of execution. The portability of OpenCL programs, however, is limited to the ability to run the same code on different devices, and this ensures that the performance is equally reliable. To get the best performance possible, it is fundamental that you refer to the execution platform, optimizing the code based on the characteristics of the device. In the following recipes, we'll examine the Python implementation of OpenCL called PyOpenCL.

Getting ready

PyOpenCL is to OpenCL what PyCUDA is to CUDA: a Python wrapper to those GPGPU platforms (PyOpenCL can run alternatively on both NVIDIA and the AMD GPU card.) It is developed and maintained by Andreas Klöckner. Installing PyOpenCL on Windows is easy when using the binary package provided by Christoph Gohlke. His webpage contains Windows binary installers for the most recent versions of hundreds of Python packages. It is of invaluable help for those Python users that use Windows.

With these instructions, you will build a 32-bit PyOpenCL library for a Python 2.7 distro on a Windows 7 machine with a NVIDIA GPU card:

1. Go to `http://www.lfd.uci.edu/~gohlke/pythonlibs/#pyopencl` and download the file from `pyopencl-2015.1-cp27-none-win32.whl` (and the relative dependencies if required).
2. Download and install the Win32 OpenCL driver (from Intel) from `http://registrationcenter.intel.com/irc_nas/5198/opencl_runtime_15.1_x86_setup.msi`.
3. Finally, install the `pyOpenCL` file from Command Prompt with the command:

   ```
   pip install pyopencl-2015.1-cp27-none-win32.whl
   ```

How to do it...

In this first example, we verify that the PyOpenCL environment is correctly installed.

So, a simple script that can enumerate all major hardware features using the OpenCL library is presented as:

```
import pyopencl as cl


def print_device_info() :
    print('\n' + '=' * 60 + '\nOpenCL Platforms and Devices')
    for platform in cl.get_platforms():
        print('=' * 60)
        print('Platform - Name:  ' + platform.name)
        print('Platform - Vendor:  ' + platform.vendor)
        print('Platform - Version:  ' + platform.version)
        print('Platform - Profile:  ' + platform.profile)

        for device in platform.get_devices():
            print('    ' + '-' * 56)
            print('    Device - Name:  ' \
                  + device.name)
            print('    Device - Type:  ' \
                  + cl.device_type.to_string(device.type))
            print('    Device - Max Clock Speed:  {0} Mhz'\
                  .format(device.max_clock_frequency))
            print('    Device - Compute Units:  {0}'\
                  .format(device.max_compute_units))
            print('    Device - Local Memory:  {0:.0f} KB'\
                  .format(device.local_mem_size/1024.0))
            print('    Device - Constant Memory:  {0:.0f} KB'\
                  .format(device.max_constant_buffer_size/1024.0))
            print('    Device - Global Memory: {0:.0f} GB'\
                  .format(device.global_mem_size/1073741824.0))
            print('    Device - Max Buffer/Image Size: {0:.0f} MB'\
                  .format(device.max_mem_alloc_size/1048576.0))
            print('    Device - Max Work Group Size: {0:.0f}'\
                  .format(device.max_work_group_size))
    print('\n')

if __name__ == "__main__":
    print_device_info()
```

The output that shows the main characteristics of the CPU and GPU card that is installed should be like this:

```
C:\Python CookBook\Chapter 6 - GPU Programming with Python\>python
PyOpenCLDeviceInfo.py

============================================================
OpenCL Platforms and Devices
============================================================
Platform - Name:  NVIDIA CUDA
Platform - Vendor:  NVIDIA Corporation
Platform - Version:  OpenCL 1.1 CUDA 6.0.1
Platform - Profile:  FULL_PROFILE
    --------------------------------------------------------
    Device - Name:  GeForce GT 240
    Device - Type:  GPU
    Device - Max Clock Speed:  1340 Mhz
    Device - Compute Units:  12
    Device - Local Memory:  16 KB
    Device - Constant Memory:  64 KB
    Device - Global Memory: 1 GB



============================================================
Platform - Name:  Intel(R) OpenCL
Platform - Vendor:  Intel(R) Corporation
Platform - Version:  OpenCL 1.2
Platform - Profile:  FULL_PROFILE
    --------------------------------------------------------
    Device - Name:  Intel(R) Core(TM)2 Duo CPU     E6550  @ 2.33GHz
    Device - Type:  CPU
    Device - Max Clock Speed:  2330 Mhz
    Device - Compute Units:  2
    Device - Local Memory:  32 KB
    Device - Constant Memory:  128 KB
    Device - Global Memory: 2 GB
```

How it works...

The code is very simple. In the first line, we import the `pyopencl` module:

```
import pyopencl as cl
```

Then, the `platform.get_devices()` method is used to get a list of devices. For each device, the set of its main features are printed on the screen:

- The name and device type
- Max clock speed
- Compute units
- Local/constant/global memory

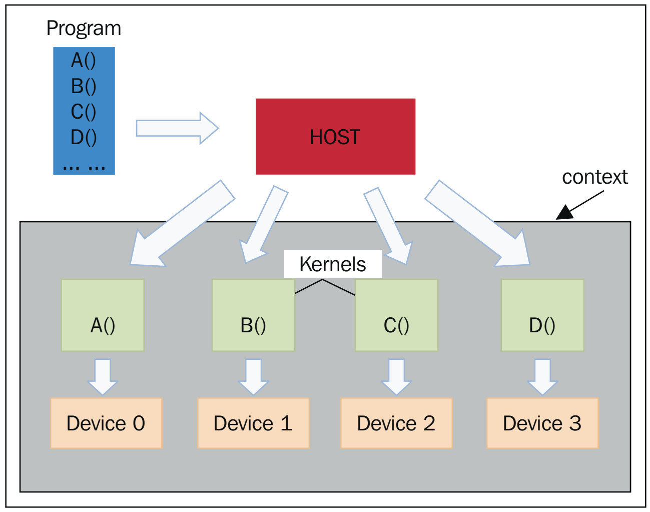
How to build a PyOpenCL application

As for programming with PyCUDA, the first step to build a program for PyOpenCL is the encoding of the host application. In fact, it is performed on the host computer (typically, the user's PC) and then it dispatches the kernel application on the connected devices (GPU cards).

The host application must contain five data structures:

- **Device**: This identifies the hardware where the kernel code must be executed. A PyOpenCL application can be executed on CPU and GPU cards but also in embedded devices, such as **Field Programmable Gate Array** (**FPGA**).
- **Program**: This is a group of kernels. A program selects the kernel that must be executed on the device.
- **Kernel**: This is the code to be executed on the device. A kernel is essentially a C-like function that enables it to be compiled for execution on any device that supports OpenCL drivers. A kernel is the only way the host can call a function that will run on a device. When the host invokes a kernel, many work items start running on the device. Each work item runs the code of the kernel, but works on a different part of the dataset.
- **Command queue**: Here, each device receives kernels through this data structure. A command queue orders the execution of kernels on the device.

- **Context**: This is a group of devices. A context allows devices to receive kernels and transfer data.



PyOpenCL programming

The preceding figure shows how these data structures can work in a host application. Note that a program can contain multiple functions to be executed on the device, and each kernel encapsulates only a single function from the program.

How to do it...

In this example, we show you the basic steps to build a PyOpenCL program. The task here is to execute the parallel sum of two vectors. In order to maintain a readable output, let's consider two vectors each from the 100 elements. The resulting vector will be for each *i*th element, which is the sum of the *i*th element `vector_a` and `vector_b`.

Of course, to be able to appreciate the parallel execution of this code, you can also increase some orders whose magnitude is of the size of the `vector_dimension` input:

```
import numpy as np
import pyopencl as cl
import numpy.linalg as la

vector_dimension = 100

vector_a = np.random.randint(vector_dimension, size=vector_dimension)
vector_b = np.random.randint(vector_dimension, size=vector_dimension)
```

```
platform = cl.get_platforms()[0]
device = platform.get_devices()[0]

context = cl.Context([device])
queue = cl.CommandQueue(context)

mf = cl.mem_flags
a_g = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR,
hostbuf=vector_a)
b_g = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR,
hostbuf=vector_b)

program = cl.Program(context, """
__kernel void vectorSum(__global const int *a_g, __global const int
*b_g, __global int *res_g) {
  int gid = get_global_id(0);
  res_g[gid] = a_g[gid] + b_g[gid];
}
""").build()

res_g = cl.Buffer(context, mf.WRITE_ONLY, vector_a.nbytes)
program.vectorSum(queue, vector_a.shape, None, a_g, b_g, res_g)

res_np = np.empty_like(vector_a)
cl.enqueue_copy(queue, res_np, res_g)

print ("PyOPENCL SUM OF TWO VECTORS")
print ("Platform Selected = %s" %platform.name )
print ("Device Selected = %s" %device.name)
print ("VECTOR LENGTH = %s" %vector_dimension)
print ("INPUT VECTOR A")
print vector_a
print ("INPUT VECTOR B")
print vector_b
print ("OUTPUT VECTOR RESULT A + B ")
print res_np

assert(la.norm(res_np - (vector_a + vector_b))) < 1e-5
```

The output from Command Prompt should be like this:

```
C:\Python CookBook\ Chapter 6 - GPU Programming with Python\Chapter 6 -
codes>python PyOpenCLParallellSum.py
```

```
Platform Selected = NVIDIA CUDA
Device Selected = GeForce GT 240

VECTOR LENGTH = 100
INPUT VECTOR A
[ 0 29 88 46 68 93 81  3 58 44 95 20 81 69 85 25 89 39 47 29 47 48 20 86
59 99  3 26 68 62 16 13 63 28 77 57 59 45 52 89 16  6 18 95 30 66 19 29
31 18 42 34 70 21 28  0 42 96 23 86 64 88 20 26 96 45 28 53 75 53 39 83
85 99 49 93 23 39  1 89 39 87 62 29 51 66  5 66 48 53 66  8 51  3 29 96
67 38 22 88]

INPUT VECTOR B
[98 43 16 28 63  1 83 18  6 58 47 86 59 29 60 68 19 51 37 46 99 27  4 94
5 22 3 96 18 84 29 34 27 31 37 94 13 89  3 90 57 85 66 63  8 74 21 18 34
93 17 26  9 88 38 28 14 68 88 90 18  6 40 30 70 93 75  0 45 86 15 10 29
84 47 74 22 72 69 33 81 31 45 62 81 66 69 14 71 96 91 51 35  4 63 36 28
65 10 41]

OUTPUT VECTOR RESULT A + B
[ 98  72 104  74 131  94 164  21  64 102 142 106 140  98 145  93 108  90
  84  75 146  75  24 180  64 121   6 122  86 146  45  47  90  59 114 151
  72 134  55 179  73  91  84 158  38 140  40  47  65 111  59  60  79 109
  66  28  56 164 111 176  82  94  60  56 166 138 103  53 120 139  54  93
 114 183  96 167  45 111  70 122 120 118 107  91 132 132  74  80 119 149
 157  59  86   7  92 132  95 103  32 129]
```

How it works...

In the first line of the code after the required module import, we defined the input vectors:

```
vector_dimension = 100
vector_a = np.random.randint(vector_dimension, size= vector_dimension)
vector_b = np.random.randint(vector_dimension, size= vector_dimension)
```

Each vector contains 100 integers items that are randomly selected thought the NumPy function `np.random.randint(max integer , size of the vector)`.

Then, we must select the device to run the kernel code. To do this, we must first select the platform using the PyOpenCL's `get_platform()` statement:

```
platform = cl.get_platforms()[0]
```

This platform, as you can see from the output, corresponds to the NVIDIA CUDA platform. Then, we must select the device using the platform's `get_device()` method:

```
device = platform.get_devices()[0]
```

In the following code, the context and queue are defined. PyOpenCL provides the method context (device selected) and queue (context selected):

```
context = cl.Context([device])
queue = cl.CommandQueue(context)
```

To perform the computation in the device, the input vector must be transferred to the device's memory. So, two input buffers in the device memory must be created:

```
mf = cl.mem_flags
a_g = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR,
hostbuf=vector_a)
b_g = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR,
hostbuf=vector_b)
```

Also, we prepare the buffer for the resulting vector:

```
res_g = cl.Buffer(context, mf.WRITE_ONLY, vector_a.nbytes)
```

Finally, the core of the script, that is, the kernel code is defined inside `program`:

```
program = cl.Program(context, """
__kernel void vectorSum(__global const int *a_g, __global const int
*b_g, __global int *res_g) {
  int gid = get_global_id(0);
  res_g[gid] = a_g[gid] + b_g[gid];
}
""").build()
```

The kernel's name is `vectorSum`, while the parameter list defines the data types of the input arguments (vectors of integers) and output data type (a vector of the integer).

In the body of the kernel function, the sum of two vectors is defined as follows:

- **Initialize the vector index**: `int gid = get_global_id(0)`
- **Sum up the vector's components**: `res_g[gid] = a_g[gid] + b_g[gid];`

In OpenCL and PyOpenCL, buffers are attached to a context and are only moved to a device once the buffer is used on that device. Finally, we execute `vectorSum` in the device:

```
program.vectorSum(queue, vector_a.shape, None, a_g, b_g, res_g)
```

To visualize the results, an empty vector is built:

```
res_np = np.empty_like(vector_a)
```

Then, the result is copied into this vector:

```
cl.enqueue_copy(queue, res_np, res_g)
```

Finally, the results are displayed:

```
print ("VECTOR LENGTH = %s" %vector_dimension)
print ("INPUT VECTOR A")
print vector_a
print ("INPUT VECTOR B")
print vector_b
print ("OUTPUT VECTOR RESULT A + B ")
print res_np
```

To check the result, we use the `assert` statement. It tests the result and triggers an error if the condition is `false`:

```
assert(la.norm(res_np - (vector_a + vector_b))) < 1e-5
```

Evaluating element-wise expressions with PyOpenCl

Similar to PyCUDA, PyOpenCL provides the functionality in the `pyopencl.elementwise` class that allows us to evaluate the complicated expressions in a single computational pass. The method that realized this is:

```
ElementwiseKernel(context, argument, operation, name,",",",
                       optional_parameters)
```

Here:

- `context`: This is the device or the group of devices on which the element-wise operation will be executed
- `argument`: This is a C-like argument list of all the parameters involved in the computation
- `operation`: This is a string that represents the operation that is to be performed on the argument list
- `name`: This is the kernel name associated with `ElementwiseKernel`
- `optional_parameters`: These are not important for this recipe.