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.

How to do it...

In this example, we will again consider the task of adding two integer vectors of 100 elements. The achievement, of course, changes because we use the `ElementwiseKernel` class, as shown:

```
import pyopencl as cl
import pyopencl.array as cl_array
import numpy as np

context = cl.create_some_context()
queue = cl.CommandQueue(context)

vector_dimension = 100
vector_a = cl_array.to_device(queue,  np.random.randint(vector_
dimension, size=vector_dimension))
vector_b = cl_array.to_device(queue,  np.random.randint(vector_
dimension, size=vector_dimension))
result_vector = cl_array.empty_like(vector_a)

elementwiseSum = cl.elementwise.ElementwiseKernel(context, "int *a,
int *b, int *c", "c[i] = a[i] + b[i]", "sum")
elementwiseSum(vector_a, vector_b, result_vector)

print ("PyOpenCL ELEMENTWISE SUM OF TWO VECTORS")
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 result_vector
```

The output of this code is as follows:

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


Choose platform:
[0] <pyopencl.Platform 'NVIDIA CUDA' at 0x2cc6c40>
[1] <pyopencl.Platform 'Intel(R) OpenCL' at 0x3cf440>
Choice [0]:0
Set the environment variable PYOPENCL_CTX='0' to avoid being asked again.
```

```
PyOpenCL ELEMENTWISE SUM OF TWO VECTORS
VECTOR LENGTH = 100
INPUT VECTOR A
[70 95 47 53 71 52 15 10 95  5 76 40 55 87  7 18 44 72  2 42 47 86 58 87
64 79 44 94  5 54 92 21 60 67 43 92 38 49 97 14 17 35 87 94  3 17 87 24
50 43  39 71 84  7 64 60 29 74 65 82 42 35 96 80 94 57 21 56 94  8  3 94
30 64 44  34 79  5 88 80 98 88  5  2 77 57  7 93 49 42 56 19 81 36 19 24
27 18  1 40]
INPUT VECTOR B
[82 32 72  9 29 29 92  2 20 44 31 91 63 97 86 37 39 41 19 78 60 30 21 69
29  38 56 49 97 18 44 84 27 73 73 14 67 43 17 58 81 52 89 84 80 96 58 80
20 91  20 61 92 46 34 98 21 82 52 34 81 45 35 28 23 59 21 89 47 75 49 43
92 91 84  59 35 61 42 12 69 15 98 85 12 36 64 89 76 29  8 81 62  5 58 13
46 82 12 66]
OUTPUT VECTOR RESULT A + B
[152 127 119  62 100  81 107  12 115  49 107 131 118 184  93  55  83 113
  21 120 107 116  79 156  93 117 100 143 102  72 136 105  87 140 116 106
 105  92 114  72  98  87 176 178  83 113 145 104  70 134  59 132 176  53
  98 158  50 156 117 116 123  80 131 108 117 116  42 145 141  83  52 137
 122 155 128  93 114  66 130  92 167 103 103  87  89  93  71 182 125  71
  64 100 143  41  77  37  73 100  13 106]
```

How it works...

In the first line of the script, we import all the requested modules:

```
import pyopencl as cl
import pyopencl.array as cl_array
import numpy
```

To initialize the context, we use the `cl.create_some_context()` method. It asks the user which context must be used to perform the calculation:

```
Choose platform:
[0] <pyopencl.Platform 'NVIDIA CUDA' at 0x2cc6c40>
[1] <pyopencl.Platform 'Intel(R) OpenCL' at 0x3cf440>
```

Then, we instantiate the queue that will receive `ElementwiseKernel`:

```
queue = cl.CommandQueue(context)
```

The input vectors and the result vector are instantiated:

```
vector_dimension = 100
vector_a = cl_array.to_device(queue,  np.random.randint(vector_
dimension, size=vector_dimension))
vector_b = cl_array.to_device(queue,  np.random.randint(vector_
dimension, size=vector_dimension))
result_vector = cl_array.empty_like(vector_a)
```

The input vectors `vector_a` and `vector_b` are integer vectors of random values that are obtained using the NumPy's `random.radint` function. The inputs vectors are defined and copied into the device using the PyOpenCL statement:

```
cl.array_to_device(queue,array)
```

Finally, the `ElementwiseKernel` object is created:

```
elementwiseSum = cl.elementwise.ElementwiseKernel(context, "int *a,
int *b, int *c", "c[i] = a[i] + b[i]", "sum")
```

In this code:

- All the arguments are in the form of a string formatted as a C argument list (they are all integers)
- A snippet of C carries out the operation, which is the sum of the vector components
- The function's name is used to compile the kernel ^s

Then, we can call the `elementwiseSum` function with the arguments defined previously:

```
elementwiseSum(vector_a, vector_b, result_vector)
```

The example ends by printing the input vectors and the result is obtained:

```
print vector_a
print vector_b
print result_vector
```

Testing your GPU application with PyOpenCL

In this chapter, we comparatively tested the performance between a CPU and GPU. Before you begin the study of the performance of algorithms, it is important to keep in mind the platform of execution on which the tests were conducted. In fact, the specific characteristics of these systems interfere with the computational time and they represent an aspect of primary importance.

To perform the tests, we used the following machines

- **GPU**: GeForce GT 240
- **CPU**: Intel Core2 Duo 2.33 Ghz
- **RAM**: DDR2 4 Gb

How to do it...

In this test, the computation time of a simple mathematical operation, that is, the sum of two vectors with elements expressed in a floating point will be evaluated and compared. To make a comparison, the same operation was implemented in two separate functions.

The first one uses only the CPU, while the second is written using PyOpenCL and makes use of the GPU for calculation. The test is performed on vectors of a dimension equal to 10,000 elements.

The code for this is as follows:

```
from time import time  # Import time tools

import pyopencl as cl
import numpy as np
import PyOpeClDeviceInfo as device_info
import numpy.linalg as la

#input vectors
a = np.random.rand(10000).astype(np.float32)
b = np.random.rand(10000).astype(np.float32)

def test_cpu_vector_sum(a, b):
    c_cpu = np.empty_like(a)
    cpu_start_time = time()
    for i in range(10000):
            for j in range(10000):
                    c_cpu[i] = a[i] + b[i]
    cpu_end_time = time()
    print("CPU Time: {0} s".format(cpu_end_time - cpu_start_time))
    return c_cpu

def test_gpu_vector_sum(a, b):
    #define the PyOpenCL Context
    platform = cl.get_platforms()[0]
    device = platform.get_devices()[0]
    context = cl.Context([device])
```

```
    queue = cl.CommandQueue(context, \
                properties=cl.command_queue_properties.PROFILING_
ENABLE)

#prepare the data structure
    a_buffer = cl.Buffer\
               (context, \
                cl.mem_flags.READ_ONLY \
                | cl.mem_flags.COPY_HOST_PTR, hostbuf=a)
    b_buffer = cl.Buffer\
               (context, \
                cl.mem_flags.READ_ONLY \
                | cl.mem_flags.COPY_HOST_PTR, hostbuf=b)
    c_buffer = cl.Buffer\
               (context, \
                cl.mem_flags.WRITE_ONLY, b.nbytes)
    program = cl.Program(context, """
    __kernel void sum(__global const float *a,
                      __global const float *b,
                      __global float *c)
    {
        int i = get_global_id(0);
        int j;
        for(j = 0; j < 10000; j++)
        {
            c[i] = a[i] + b[i];
        }
    }""").build()
    #start the gpu test
    gpu_start_time = time()
    event = program.sum(queue, a.shape, None, \
                        a_buffer, b_buffer, c_buffer)
    event.wait()
    elapsed = 1e-9*(event.profile.end - event.profile.start)
    print("GPU Kernel evaluation Time: {0} s".format(elapsed))
    c_gpu = np.empty_like(a)
    cl.enqueue_read_buffer(queue, c_buffer, c_gpu).wait()
    gpu_end_time = time()
    print("GPU Time: {0} s".format(gpu_end_time - gpu_start_time))
    return c_gpu

#start the test
if __name__ == "__main__":
    #print the device info
```

```
        device_info.print_device_info()
        #call the test on the cpu
        cpu_result = test_cpu_vector_sum(a, b)
        #call the test on the gpu
        gpu_result = test_gpu_vector_sum(a, b)
        #
        assert (la.norm(cpu_result - gpu_result)) < 1e-5
```

The output of the test is as follows, where the device information with the execution time is printed out:

```
C:\Python Cook\Chapter 6 - GPU Programming with Python\Chapter 6 -
codes>python PyOpenCLTestApplication.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
    Device - Max Buffer/Image Size: 256 MB
    Device - Max Work Group Size: 512
============================================================
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
Device - Max Buffer/Image Size: 512 MB
Device - Max Work Group Size: 8192


CPU Time: 71.9769999981 s
GPU Kernel Time: 0.075756608 s
GPU Time: 0.0809998512268 s
```

Even if the test is not computationally expansive, it provides useful indications of the potential of a GPU card.

How it works...

As explained in the preceding section, the test consists of two parts. The code that runs on the CPU and the code that runs on the GPU. Both were taken to the execution time.

Regarding the test on the CPU, the `test_cpu_vector_sum` function has been implemented. It consists of two loops on 10,000 vectors elements:

```
cpu_start_time = time()
  for i in range(10000):
            for j in range(10000):
        c_cpu[i] = a[i] + b[i]
  cpu_end_time = time()
```

The sum operation of the *i*th vector components is executed 1,000,000,000 times, and it will be computationally expensive.

The total CPU time will have the following difference:

```
CPU Time = cpu_end_time - cpu_start_time
```

To test the GPU time, we implemented the regular definition schema of an application for PyOpenCL:

- We established the definition of the device and context
- We set up the queue for execution

- We created memory areas to perform the computation on the device (three buffers defined as `a_buffer`, `b_buffer`, `c_buffer`)
- We built the kernel
- We evaluated the kernel call and GPU time:

```
gpu_start_time = time()
             event = program.sum(queue, a.shape, None, \
                         a_buffer, b_buffer, c_buffer)

             cl.enqueue_read_buffer(queue, c_buffer, c_gpu).wait()
   gpu_end_time = time()
```

Here, `GPU Time =  gpu_end_time - gpu_start_time`.

Finally, in the main program we call the testing function and `print_device_info()` that we defined previously:

```
if __name__ == "__main__":
    device_info.print_device_info()
    cpu_result = test_cpu_vector_sum(a, b)
    gpu_result = test_gpu_vector_sum(a, b)
    assert (la.norm(cpu_result - gpu_result)) < 1e-5
```

To check the result, we used the `assert` statement that verifies the result and triggers an error if the condition is `false`.