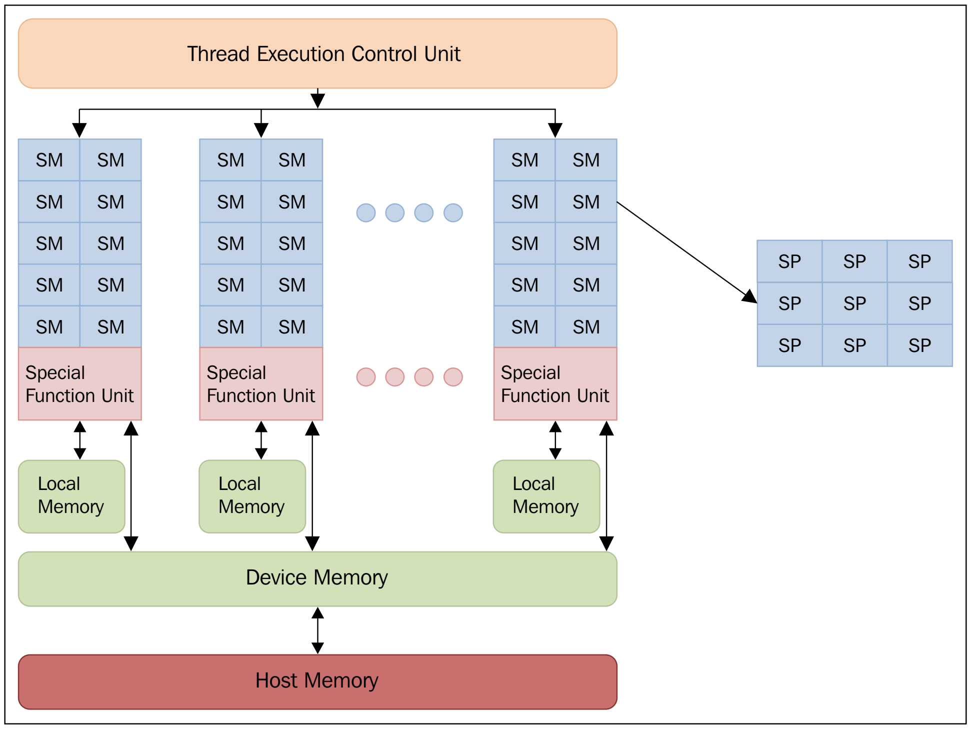
6
GPU Programming with Python

In this chapter, we will cover the following recipes:

- Using the PyCUDA module
- How to build a PyCUDA application
- Understanding the PyCUDA Memory Model with matrix manipulation
- Kernel invocations with GPUArray
- Evaluating element-wise expressions with PyCUDA
- The MapReduce operation with PyCUDA
- GPU programming with NumbaPro
- Using GPU-accelerated libraries with NumbaPro
- Using the PyOpenCL module
- How to build a PyOpenCL application
- Evaluating element-wise expressions with PyOpenCL
- Testing your GPU application with PyOpenCL

Introduction

The **graphics processing unit** (**GPU**) is an electronic circuit that specializes in processing data to render images from polygonal primitives. Although they were designed to carry out rendering images, the GPU has continued to evolve, becoming more complex and efficient in serving both the real-time and offline rendering community and in performing any scientific computations. GPUs are characterized by a highly parallel structure, which allows it to manipulate large datasets in an efficient manner. This feature combined with the rapid improvement in graphics hardware performance and the extent of programmability caught the attention of the scientific world with the possibility of using GPU for purposes other than just rendering images. Traditional GPUs are fixed function devices where the whole rendering pipeline is built on hardware. This restricts graphics programmers, leading them to use different, efficient and high-quality rendering algorithms. Hence, a new GPU was built with millions of lightweight parallel cores, which were programmable to render graphics using **shaders**. This is one of the biggest advancements in the field of computer graphics and the gaming industry. With lots of programmable cores available, the GPU vendors started developing models for parallel programming. Each GPU is indeed composed of several processing units called **Streaming Multiprocessor** (**SM**) that represent the first logic level of parallelism; and each SM infact works simultaneously and independently from the others.



The GPU architecture

Each SM is in turn divided into a group of **Stream Processors** (**SP**), each of which has a core of real execution and can sequentially run a thread. An SP represents the smallest unit of an execution logic and represents the level of finer parallelism. The division in SM and SP is structural in nature, but it is possible to outline a further logical organization of the SP of a GPU, which are grouped together in logical blocks characterized by a particular mode of execution. All cores that make up a group run the same instruction at the same time. This is just the **Single instruction, multiple data** (**SIMD**) model, which we described in the first chapter of this book.

Each SM also has a number of registers, which represent an area of memory for quick access that is temporary, local (not shared between the cores), and limited in size. This allows storage of frequently used values from a single core. The **general-purpose computing on graphics processing units** (**GP-GPU**) is the field devoted to the study of the techniques needed to exploit the computing power of the GPU to perform calculations quickly, thanks to the high level of parallelism inside. As seen before, GPUs are structured quite differently from conventional processors; for this, they have problems of a different nature and require specific programming techniques. The most outstanding feature that distinguishes a graphics processor is the high number of cores available, which allow us to carry out many threads of execution competitors, which are partially synchronized for the execution of the same operation. This feature is very useful and efficient in situations where you want to split your work in many parts to perform the same operations on different data. On the contrary, it is hard to make the best use of this architecture when there is a strong sequential and logical order to be respected in the operations to be carried out; otherwise, the work cannot be evenly divided into many small subparts. The programming paradigm that characterizes the GPU computing is called Stream Processing because the data can be viewed as a homogeneous flow of values to which the same operations are applied synchronously.

Currently, the most efficient solutions to exploit the computing power provided by GPU cards are the software libraries CUDA and OpenCL. In the following recipes, we will present the realization of these software libraries in the Python programming language.

Using the PyCUDA module

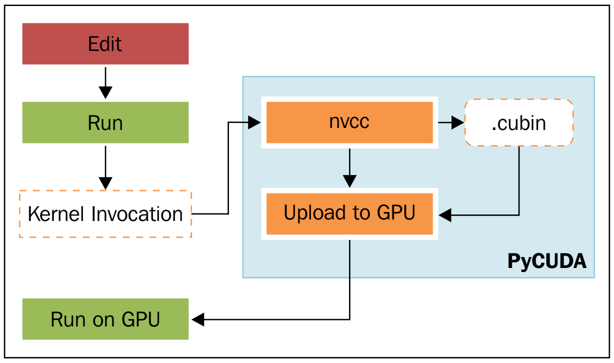
PyCUDA is a Python wrap for **Compute Unified Device Architecture** (**CUDA**), the software library developed by NVIDIA for GPU programming. The CUDA programming model is the starting point of understanding how to program the GPU properly with PyCUDA. There are concepts that must be understood and assimilated to be able to approach this tool correctly and to understand the more specific topics that are covered in the following recipes.

A hybrid programming model

The programming model "hybrid" of CUDA (and consequently of PyCUDA, which is a Python wrapper) is implemented through specific extensions to the standard library of the C language. These extensions have been created, whenever possible, syntactically like the function calls in the standard C library. This allows a relatively simple approach to a hybrid programming model that includes the host and device code. The management of the two logical parts is done by the NVCC compiler. Here is a brief description of how this compiler works:

1. It separates a device code from a host-code device.
2. It invokes a default compiler (for example, GCC) to compile the host code.
3. It builds the device code in the binary form (Cubin objects) or in the form assembly (code PTX).
4. It generates a host key "global" that also includes code PTX.

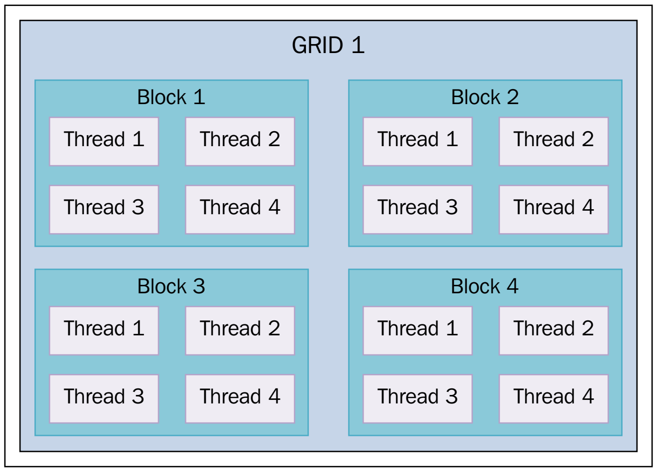
The compiled CUDA code is converted to a device-specific binary by the driver, during runtime. All the previously mentioned steps are executed by PyCUDA at runtime, which makes it a **Just-in-time** (**JIT**) compiler. The drawback of this approach is the increased load time of the application, which is the only way to maintain compatibility "forward", that is, you can perform operations on a device that does not exist at the time of the actual compilation. A JIT compilation therefore makes an application compatible with future devices that are built on architectures with higher computing power, so it is not yet possible to generate any binary code.



The PyCUDA execution model

The kernel and thread hierarchy

An important element of a CUDA program is a **kernel**. It represents the code that is executed parallelly on the basis of specifications that will be clarified later with the examples described here. Each kernel's execution is done by computing units that are called **threads**. Unlike threads in CPU, GPU threads are lighter in such a way that the change of context is not one of the factors to be taken into account in a code performance evaluation because it can be considered as instantaneous. To determine the number of threads that must perform a single kernel and their logical organization, CUDA defines a two-level hierarchy. In the highest level, it defines a so-called grid of blocks. This grid represents a bidimensional structure where the thread blocks are distributed, which are three-dimensional.



The distribution of (3-dimensional) threads in a two-level hierarchy of PyCUDA

Based on this structure, a kernel function must be launched with additional parameters that specify precisely the size of the grid and block.

Getting ready

On the Wiki page `http://wiki.tiker.net/PyCuda/Installation`, the basic instructions to install PyCuda on the main operative systems (Linux, Mac, and Windows) are explained.

With these instructions, you will build a 32-bit PyCUDA library for a Python 2.7 distro:

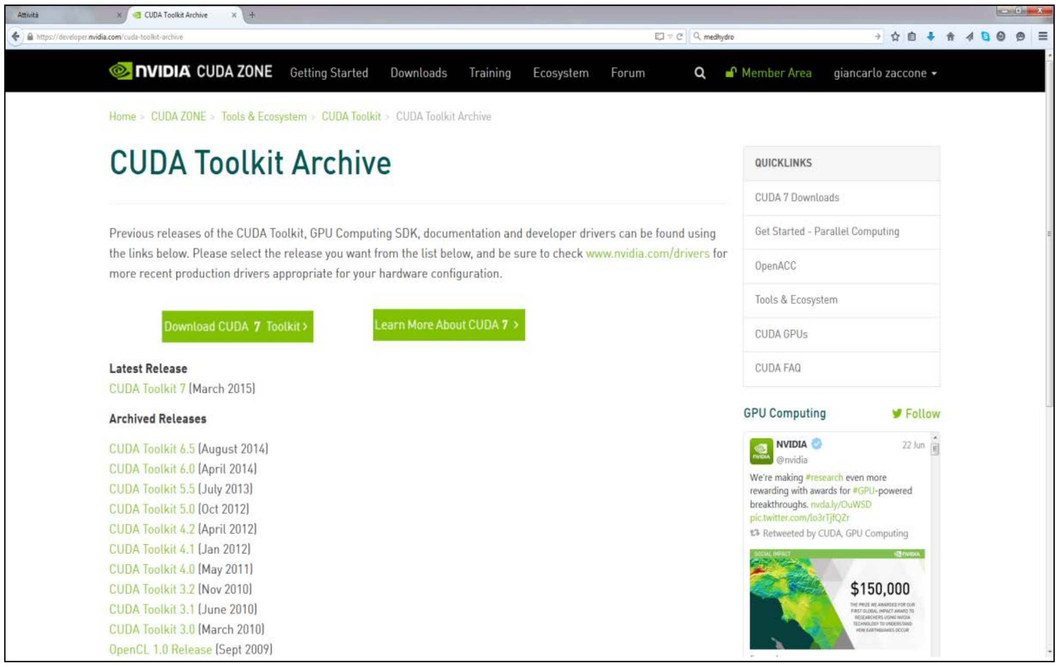
1. The first step is to download and install all the components provided by NVDIA to develop with CUDA (refer to `https://developer.nvidia.com/cuda-toolkit-archive`) for all the available versions. These components are:
   - The CUDA toolkit is available at `http://developer.download.nvidia.com/compute/cuda/4_2/rel/toolkit/cudatoolkit_4.2.9_win_32.msi`.
   - The NVIDIA GPU Computing SDK is available at `http://developer.download.nvidia.com/compute/cuda/4_2/rel/sdk/gpucomputingsdk_4.2.9_win_32.exe`.
   - The NVIDIA CUDA Development Driver is available at `http://developer.download.nvidia.com/compute/cuda/4_2/rel/drivers/devdriver_4.2_winvista-win7_32_301.32_general.exe`.
2. Download and install NumPy (for 32-bit Python 2.7) and Visual Studio C++ 2008 Express (be sure to set all the system variables).
3. Open the file `msvc9compiler.py` located at `/Python27/lib/distutils/`. After the line 641: `ld_args.append ('/IMPLIB:' + implib_file)`, add the new line `ld_args.append('/MANIFEST')`.
4. Download PyCUDA from `https://pypi.python.org/pypi/pycuda`.
5. Open Visual Studio 2008 Command Prompt, click on Start, go to **All Programs** | **Microsoft Visual Studio 2008** | **Visual Studio Tools** | **Visual Studio Command Prompt (2008)**, and follow the given steps:
   1. Go in the `PyCuda` directory.
   2. Execute `python configure.py`.
   3. Edit the created file `siteconf.py`:

```
BOOST_INC_DIR = []
BOOST_LIB_DIR = []
BOOST_COMPILER = 'gcc43'
USE_SHIPPED_BOOST = True
BOOST_PYTHON_LIBNAME = ['boost_python']
BOOST_THREAD_LIBNAME = ['boost_thread']
CUDA_TRACE = False
CUDA_ROOT = 'C:\\Program Files\\NVIDIA GPU Computing
Toolkit\\CUDA\\v4.2'
CUDA_ENABLE_GL = False
CUDA_ENABLE_CURAND = True
CUDADRV_LIB_DIR = ['${CUDA_ROOT}/lib/Win32']
CUDADRV_LIBNAME = ['cuda']
```

```
CUDART_LIB_DIR = ['${CUDA_ROOT}/lib/Win32']
CUDART_LIBNAME = ['cudart']
CURAND_LIB_DIR = ['${CUDA_ROOT}/lib/Win32']
CURAND_LIBNAME = ['curand']
CXXFLAGS = ['/EHsc']
LDFLAGS = ['/FORCE']
```

6. Finally, install PyCUDA with the following commands in VS2008 Command Prompt:

```
python setup.py build
python setup.py install
```



Home > CUDA ZONE > Tools & Ecosystem > CUDA Toolkit > CUDA Toolkit Archive

CUDA Toolkit Archive

Previous releases of the CUDA Toolkit, GPU Computing SDK, documentation and developer drivers can be found using the links below. Please select the release you want from the list below, and be sure to check www.nvidia.com/drivers for more recent production drivers appropriate for your hardware configuration.

Download CUDA 7 Toolkit >   Learn More About CUDA 7 >

Latest Release
CUDA Toolkit 7 (March 2015)

Archived Releases
CUDA Toolkit 6.5 (August 2014)
CUDA Toolkit 6.0 (April 2014)
CUDA Toolkit 5.5 (July 2013)
CUDA Toolkit 5.0 (Oct 2012)
CUDA Toolkit 4.2 (April 2012)
CUDA Toolkit 4.1 (Jan 2012)
CUDA Toolkit 4.0 (May 2011)
CUDA Toolkit 3.2 (Nov 2010)
CUDA Toolkit 3.1 (June 2010)
CUDA Toolkit 3.0 (March 2010)
OpenCL 1.0 Release (Sept 2009)

QUICKLINKS
CUDA 7 Downloads
Get Started - Parallel Computing
OpenACC
Tools & Ecosystem
CUDA GPUs
CUDA FAQ

GPU Computing   Follow

NVIDIA @nvidia   22 Jun
We're making #research even more rewarding with awards for #GPU-powered breakthroughs. nvda.ly/OuWSD pic.twitter.com/Io3rTjfQZr
Retweeted by CUDA, GPU Computing

SOCIAL IMPACT
$150,000
THE PRIZE WE AWARDED FOR OUR FIRST GLOBAL IMPACT AWARD TO RESEARCHERS USING NVIDIA TECHNOLOGY TO UNDERSTAND HOW EARTHQUAKES OCCUR

The CUDA toolkit download page

How to do it...

The present example has a dual function. The first is to verify that PyCUDA is properly installed and the second is to read and print the characteristics of the GPU cards:

```
import pycuda.driver as drv
drv.init()
print "%d device(s) found." % drv.Device.count()
for ordinal in range(drv.Device.count()):
       dev = drv.Device(ordinal)
       print "Device #%d: %s" % (ordinal, dev.name())
       print " Compute Capability: %d.%d" % dev.compute_capability()
       print " Total Memory: %s KB" % (dev.total_memory()//(1024))
```

After running the code, we should have an output like this:

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


1 device(s) found.
Device #0: GeForce GT 240
 Compute Capability: 1.2
 Total Memory: 1048576 KB
```

How it works...

The execution is pretty simple. In the first line of code, `pycuda.driver` is imported and then initialized:

```
import pycuda.driver as drv
drv.init()
```

The `pycuda.driver` module exposes the driver level to the programming interface of CUDA, which is more flexible than the CUDA C "runtime-level" programming interface, and it has a few features that are not present at runtime.

Then, it cycles into `drv.Device.count()`, and for each GPU card found, the name of the cards and main characteristics (computing capability and total memory) are printed:

```
print "Device #%d: %s" % (ordinal, dev.name())
print " Compute Capability: %d.%d" % dev.compute_capability()
print " Total Memory: %s KB" % (dev.total_memory()//(1024))
```

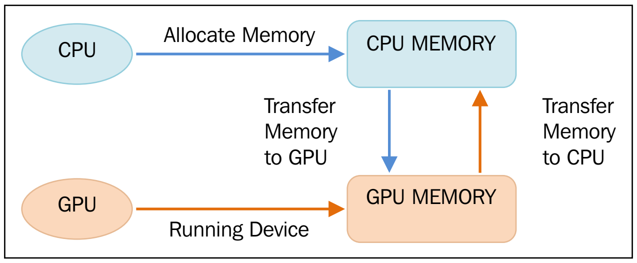
See also

- PyCUDA is developed by Andreas Klöckner (`http://mathema.tician.de/aboutme/`). For any other information concerning PyCUDA, you can refer to `http://documen.tician.de/pycuda/`.

How to build a PyCUDA application

The PyCUDA programming model is designed for the common execution of a program on a CPU and GPU, so as to allow you to perform the sequential parts on the CPU and the numeric parts, which are more intensive on the GPU. The phases to be performed in the sequential mode are implemented and executed on the CPU (host), while the steps to be performed in parallel are implemented and executed on the GPU (device). The functions to be performed in parallel on the device are called kernels. The steps to execute a generic function kernel on the device are as follows:

1. The first step is to allocate the memory on the device.
2. Then we need to transfer data from the host memory to that allocated on the device.
3. Next, we need to run the device:
    1. Run the configuration.
    2. Invoke the kernel function.
4. Then, we need to transfer the results from the memory on the device to the host memory.
5. Finally, release the memory allocated on the device.



The PyCUDA programming model

How to do it...

To show the PyCUDA workflow, let's consider a 5×5 random array and the following procedure:

1. Create the 5×5 array on the CPU.
2. Transfer the array to the GPU.
3. Perform a task on the array in the GPU (double all the items in the array).
4. Transfer the array from the GPU to the CPU.
5. Print the results.

The code for this is as follows:

```
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule

import numpy

a = numpy.random.randn(5,5)
a = a.astype(numpy.float32)

a_gpu = cuda.mem_alloc(a.nbytes)
cuda.memcpy_htod(a_gpu, a)

mod = SourceModule("""
  __global__ void doubleMatrix(float *a)
  {
    int idx = threadIdx.x + threadIdx.y*4;
    a[idx] *= 2;
  }
  """)

func = mod.get_function("doubleMatrix")
func(a_gpu, block=(5,5,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print ("ORIGINAL MATRIX")
print a
print ("DOUBLED MATRIX AFTER PyCUDA EXECUTION")
print a_doubled
```

The example output should be like this:

```
C:\Python CookBook\Chapter 6 - GPU Programming with Python\ >python
PyCudaWorkflow.py
ORIGINAL MATRIX
[[-0.59975582  1.93627465  0.65337795  0.13205571 -0.46468592]
 [ 0.01441949  1.40946579  0.5343408  -0.46614054 -0.31727529]
 [-0.06868593  1.21149373 -0.6035406  -1.29117763  0.47762445]
 [ 0.36176383 -1.443097    1.21592784 -1.04906416 -1.18935871]
 [-0.06960868 -1.44647694 -1.22041082  1.17092752  0.3686313 ]]
```

```
DOUBLED MATRIX AFTER PyCUDA EXECUTION
[[-1.19951165  3.8725493   1.3067559   0.26411143 -0.92937183]
 [ 0.02883899  2.81893158  1.0686816  -0.93228108 -0.63455057]
 [-0.13737187  2.42298746 -1.2070812  -2.58235526  0.95524889]
 [ 0.72352767 -1.443097    1.21592784 -1.04906416 -1.18935871]
 [-0.06960868 -1.44647694 -1.22041082  1.17092752  0.3686313 ]]
```

How it works...

The preceding code starts with the following imports:

```
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule
```

The `import pycuda.autoinit` statement automatically picks a GPU to run based on its availability and number. It also creates a GPU context for the subsequent code to run. If needed, both the chosen device and the created context are available from `pycuda.autoinit` as importable symbols, whereas the `SourceModule` component is the object where a C-like code for the GPU must be written.

The first step is to generate the input 5×5 matrix. Since most GPU computations involve large arrays of data, the `numpy` module must be imported:

```
import numpy
a = numpy.random.randn(5,5)
```

Then, the items in the matrix are converted into a single precision mode, many NVIDIA cards support only a single precision:

```
a = a.astype(numpy.float32)
```

The first operation that needs to be done in order to implement a GPU is to load the input array from the host memory (CPU) to the device (GPU). This is done at the beginning of the operation and consists of two steps that are performed by invoking the following two functions provided PyCUDA:

- The memory allocation on the device is performed via the function `cuda.mem_alloc`. The device and host memory may *not ever* communicate while performing a function kernel. This means that, to run a function parallelly on the device, the data related to it *must* be present in the memory of the device itself. Before you copy data from the host memory to the device memory, you must allocate the memory required on the device: `a_gpu = cuda.mem_alloc(a.nbytes)`.

- Copy the matrix from the host memory to that of the device with the following function:

  ```
  call cuda.memcpy_htod : cuda.memcpy_htod(a_gpu, a).
  ```

Also note that `a_gpu` is one-dimensional and on the device, we need to handle it as such. All these operations do not require the invocation of a kernel and are made directly by the main processor. The `SourceModule` entity serves to define the (C-like) kernel function `doubleMatrix` that multiplies each array entry by `2`:

```
mod = SourceModule("""
  __global__ void doubleMatrix(float *a)
  {
    int idx = threadIdx.x + threadIdx.y*4;
    a[idx] *= 2;
  }
  """)
```

The `__global__ qualifier` directive indicates that the function `doubleMatrix` will be processed on the device. Only the CUDA nvcc compiler will perform this task.

Let's take a look at the function's body:

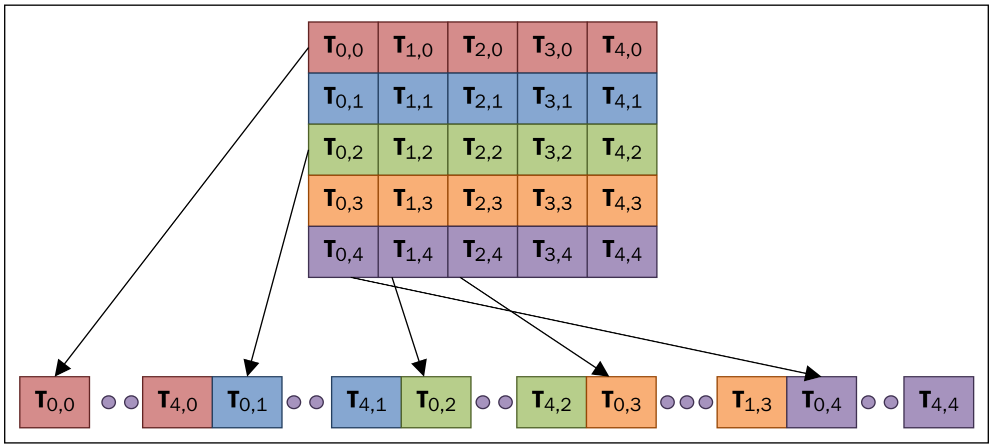
```
int idx = threadIdx.x + threadIdx.y*4;
```

The `idx` parameter is the matrix index identified by the thread coordinates `threadIdx.x` and `threadIdx.y`. Then, the element matrix with the index `idx` is multiplied by `2`:

```
                 a[idx] *= 2;
```

Note that this kernel function will be executed once in 16 different threads. Both the variables `threadIdx.x` and `threadIdx.y` contain indices between `0` and `3` and the pair is different for each thread. Threads scheduling is directly linked to the GPU architecture and its intrinsic parallelism. A block of threads is assigned to a single **Streaming Multiprocessor** (**SM**), and the threads are further divided into groups called **warps**. The size of a warp depends on the architecture under consideration. The threads of the same warp are managed by the control unit called the **warp scheduler**. To take full advantage of the inherent parallelism of SM, the threads of the same warp must execute the same instruction. If this condition does not occur, we speak of the divergence of threads. If the same warp threads execute different instructions, the control unit cannot handle all the warps. It must follow the sequences of instructions for every single thread (or for homogeneous subsets of threads) in a serial mode. Let's observe how the thread block is divided into various warps, threads are divided by the value of `threadIdx`.

The `threadIdx` structure consists of three fields: `threadIdx.x`, `threadIdx.y`, and `threadIdx.z`.


$T_{0,0}$ $T_{1,0}$ $T_{2,0}$ $T_{3,0}$ $T_{4,0}$
$T_{0,1}$ $T_{1,1}$ $T_{2,1}$ $T_{3,1}$ $T_{4,1}$
$T_{0,2}$ $T_{1,2}$ $T_{2,2}$ $T_{3,2}$ $T_{4,2}$
$T_{0,3}$ $T_{1,3}$ $T_{2,3}$ $T_{3,3}$ $T_{4,3}$
$T_{0,4}$ $T_{1,4}$ $T_{2,4}$ $T_{3,4}$ $T_{4,4}$
$T_{0,0}$ $T_{4,0}$ $T_{0,1}$ $T_{4,1}$ $T_{0,2}$ $T_{4,2}$ $T_{0,3}$ $T_{1,3}$ $T_{0,4}$ $T_{4,4}$

Thread blocks subdivision: T(x,y) where x = threadIdx.x and y = threadIdx.y

We can see that the code in the kernel function will be automatically compiled by the nvcc CUDA compiler. If there are no errors, the pointer of this compiled function will be created. In fact, `mod.get_function("doubleMatrix")` returns an identifier to the `func` function that we created:

```
func = mod.get_function("doubleMatrix ")
```

To perform a function on the device, you must first configure the execution appropriately. This means that you need to determine the size of the coordinates to identify and distinguish the thread belonging to different blocks. This will be done using the block parameter inside the `func` call:

```
func(a_gpu, block = (5, 5, 1))
```

The `block = (5, 5, 1)` function tells us that we are calling a kernel function with the `a_gpu` linearized input matrix and a single thread block of the size `5` threads in the *x* direction, `5` threads in the *y* direction, and `1` thread in the *z* direction, `16` threads in total. This structure is designed with the parallel implementation of the algorithm in mind. The division of the workload results in an early form of parallelism that is sufficient and necessary to make use of the computing resources provided by the GPU. Once you've configured the kernel's invocation, you can invoke the kernel function that executes instructions parallelly on the device. Each thread executes the same code kernel.

After the computation in the GPU device, we use an array to store the results:

```
a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
```

This will be printed as follows:

```
print a
print a_doubled
```

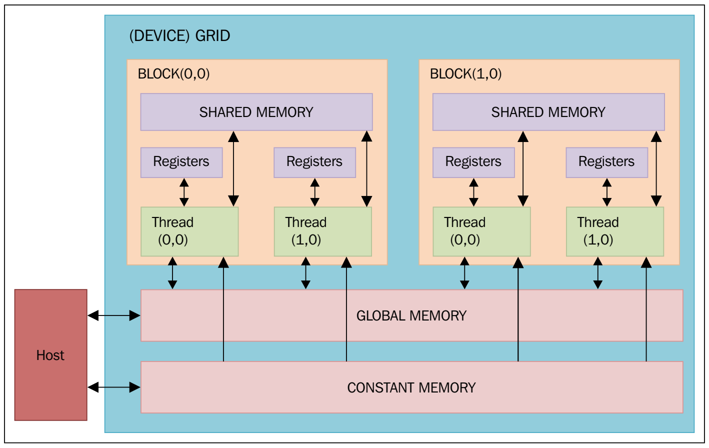
There's more...

A warp executes one common instruction at a time. So, to maximize the efficiency of the structure all must agree with the same thread's path of execution. When more than one thread block is assigned to a multiprocessor to run, they are partitioned into warps that are scheduled by a component called the warp scheduler.

Understanding the PyCUDA memory model with matrix manipulation

A PyCUDA program, to make the most of available resources, should respect the rules dictated by the structure and the internal organization of the SM that imposes constraints on the performance of the thread. In particular, the knowledge and correct use of the various types of memory that the GPU makes available is fundamental in order to achieve maximum efficiency in the programs. In the CUDA-capable GPU card, there are four types of memories, which are defined, as follows:

- **Registers**: In this, a register is allocated for each thread. This can only access its register but not the registers of other threads, even if they belong to the same block.
- **The shared memory**: Here, each block has its own shared memory between the threads that belong to it. Even this memory is extremely fast.
- **The constant memory**: All threads in a grid have constant access to the memory, but can be accessed only while reading. The data present in it persists for the entire duration of the application.
- **The global memory**: All threads of all the grids (so all kernels) have access to the global memory. The constant memory data present in it persists for the entire duration of the application.



The GPU memory model

One of the key points to understand how to make the PyCUDA programs with satisfactory performance is that not all memory is the same, but you have to try to make the best of each type of memory. The basic idea is to minimize the global memory access via the use of the shared memory. The technique is usually used to divide the domain/codomain of the problem in such a way so that we enable a block of threads to perform its elaborations in a closed subset of data. In this way, the threads adhering to the concerned block will work together to load the shared global memory area that is to be processed in the memory, to then proceed to exploiting the higher speed of this memory zone.

The basic steps to be performed for each thread will then be as follows:

1. Load data from the global memory to the shared memory.
2. Synchronize all the threads of the block so that everyone can read safety positions shared memory filled by other threads.
3. Process the data of the shared memory.
4. Make a new synchronization as necessary to ensure that the shared memory has been updated with the results.
5. Write the results in the global memory.

How to do it…

To better understand this technique, we'll present an example which will clarify this approach. This example is based on the product of two matrices. The previous figure shows the product of matrices in the standard way and the correspondent sequential code to calculate where each element must be loaded from a row and a column of the matrix input:

```
void SequentialMatrixMultiplication(float*M,float *N,float *P, int
width)
{
  for (int i=0; i< width; ++i)
      for(int j=0;j < width; ++j) {
          float sum = 0;
          for (int k = 0 ; k < width; ++k) {
              float a = M[I * width + k];
              float b = N[k * width + j];
              sum += a * b;
                     }
         P[I * width + j] = sum;
    }
}
```

If each thread was entrusted with the task of calculating an element of the matrix, the memory accesses would dominate the execution time of the algorithm. What we can do is rely on a block of threads for the task of calculating a submatrix of output so that it is possible to reuse the data loaded from the global memory and to collaborate threads in order to minimize the memory accesses for each of them.

The following example shows this technique:

```
import numpy as np
from pycuda import driver, compiler, gpuarray, tools

# -- initialize the device
import pycuda.autoinit

kernel_code_template = """
__global__ void MatrixMulKernel(float *a, float *b, float *c)
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    float Pvalue = 0;
    for (int k = 0; k < %(MATRIX_SIZE)s; ++k) {
        float Aelement = a[ty * %(MATRIX_SIZE)s + k];
        float Belement = b[k * %(MATRIX_SIZE)s + tx];
        Pvalue += Aelement * Belement;
    }

    c[ty * %(MATRIX_SIZE)s + tx] = Pvalue;
}
"""
MATRIX_SIZE = 5

a_cpu = np.random.randn(MATRIX_SIZE, MATRIX_SIZE).astype(np.float32)
b_cpu = np.random.randn(MATRIX_SIZE, MATRIX_SIZE).astype(np.float32)
c_cpu = np.dot(a_cpu, b_cpu)
a_gpu = gpuarray.to_gpu(a_cpu)
b_gpu = gpuarray.to_gpu(b_cpu)

c_gpu = gpuarray.empty((MATRIX_SIZE, MATRIX_SIZE), np.float32)

kernel_code = kernel_code_template % {
    'MATRIX_SIZE': MATRIX_SIZE
    }
mod = compiler.SourceModule(kernel_code)

matrixmul = mod.get_function("MatrixMulKernel")

matrixmul(
    a_gpu, b_gpu,
    c_gpu,
    block = (MATRIX_SIZE, MATRIX_SIZE, 1),
    )

# print the results
print "-" * 80
print "Matrix A (GPU):"
print a_gpu.get()

print "-" * 80
print "Matrix B (GPU):"
print b_gpu.get()

print "-" * 80
print "Matrix C (GPU):"
print c_gpu.get()

print "-" * 80
print "CPU-GPU difference:"
print c_cpu - c_gpu.get()

np.allclose(c_cpu, c_gpu.get())
```

The example output will be as follows:

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


-------------------------------------------------------------------------
Matrix A (GPU):
[[ 0.90780383 -0.4782407   0.23222363 -0.63184392  1.05509627]
 [-1.27266967 -1.02834761 -0.15528528 -0.09468858  1.037099  ]
 [-0.18135822 -0.69884419  0.29881889 -1.15969539  1.21021318]
 [ 0.20939326 -0.27155793 -0.57454145  0.1466181   1.84723163]
 [ 1.33780348 -0.42343542 -0.50257754 -0.73388749 -1.883829  ]]
-------------------------------------------------------------------------
```

```
Matrix B (GPU):
[[ 0.04523897  0.99969769 -1.04473436  1.28909719  1.10332143]
 [-0.08900332 -1.3893919   0.06948703 -0.25977209 -0.49602833]
 [-0.6463753  -1.4424541  -0.81715286  0.67685211 -0.94934392]
 [ 0.4485206  -0.77086055 -0.16582981  0.08478995  1.26223004]
 [-0.79841441 -0.16199949 -0.35969591 -0.46809086  0.20455229]]
------------------------------------------------------------------------
Matrix C (GPU):
[[-1.19226956  1.55315971 -1.44614291  0.90420711  0.43665022]
 [-0.73617989  0.28546685  1.02769876 -1.97204924 -0.65403283]
 [-1.62555301  1.05654192 -0.34626681 -0.51481217 -1.35338223]
 [-1.0040834   1.00310731 -0.4568972  -0.90064859  1.47408712]
 [ 1.59797418  3.52156591 -0.21708387  2.31396151  0.85150564]]
------------------------------------------------------------------------


CPU-GPU difference:
[[  0.00000000e+00   0.00000000e+00   0.00000000e+00  -5.96046448e-08
    0.00000000e+00]
 [  0.00000000e+00   5.96046448e-08   0.00000000e+00   0.00000000e+00
    5.96046448e-08]
 [ -1.19209290e-07   2.38418579e-07   0.00000000e+00  -5.96046448e-08
    0.00000000e+00]
 [  0.00000000e+00   0.00000000e+00  -2.98023224e-08  -5.96046448e-08
    0.00000000e+00]
 [  1.19209290e-07   0.00000000e+00   0.00000000e+00   0.00000000e+00
    0.00000000e+00]]
```

How it works...

Let's consider the PyCUDA programming workflow. First of all, we must prepare the input matrix and the output matrix to store the results:

```
MATRIX_SIZE = 2
a_cpu = np.random.randn(MATRIX_SIZE, MATRIX_SIZE).astype(np.float32)
b_cpu = np.random.randn(MATRIX_SIZE, MATRIX_SIZE).astype(np.float32)
c_cpu = np.dot(a_cpu, b_cpu)
```

Then, we transfer these matrixes in the GPU device with the PyCUDA function `gpuarray.to_gpu()`:

```
a_gpu = gpuarray.to_gpu(a_cpu)
b_gpu = gpuarray.to_gpu(b_cpu)
c_gpu = gpuarray.empty((MATRIX_SIZE, MATRIX_SIZE), np.float32)
```

The core of the algorithm is the kernel function:

```
__global__ void MatrixMulKernel(float *a, float *b, float *c)
{
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    float Pvalue = 0;

    for (int k = 0; k < %(MATRIX_SIZE)s; ++k) {
        float Aelement = a[ty * %(MATRIX_SIZE)s + k];
        float Belement = b[k * %(MATRIX_SIZE)s + tx];
        Pvalue += Aelement * Belement;
    }

    c[ty * %(MATRIX_SIZE)s + tx] = Pvalue;
}
```

Note that the `__global__` keyword specifies that this function is a kernel function, and it must be called from a host to generate the thread hierarchy on the device.

The `threadIdx.x` and `threadIdy.y` are the threads indexes in the grid. We also note again that all these threads execute the same kernel code, so different threads will have different values with different thread coordinates. In this parallel version, the loop variables *i* and *j* of the sequential version (refer to the code in the *How to do it* section) are now replaced with `threadIdx.x` and `threadIdx.y`. The loop iteration through these indexes is simply replaced by these thread indexes, so in the parallel version, we have only one loop iteration. When the kernel `MatrixMulKernel` is invoked, it is executed as a grid of the size 2×2 of parallel threads:

```
mod = compiler.SourceModule(kernel_code)
matrixmul = mod.get_function("MatrixMulKernel")
matrixmul(
    a_gpu, b_gpu,
    c_gpu,
    block = (MATRIX_SIZE, MATRIX_SIZE, 1),
    )
```

Each CUDA thread grid typically comprises of thousands to millions of lightweight GPU threads per kernel invocation. Creating enough threads to fully utilize the hardware often requires a large amount of data parallelism; for example, each element of a large array might be computed in a separate thread.

Finally, we print out the results to verify that the computation is ok and report the differences between the `c_cpu` and `c_gpu` matrix products:

```
print "-" * 80
print "CPU-GPU difference:"
print c_cpu - c_gpu.get()

np.allclose(c_cpu, c_gpu.get())
```

Kernel invocations with GPUArray

In the previous recipe, we saw how to invoke a kernel function using the class:

```
pycuda.compiler.SourceModule(kernel_source, nvcc="nvcc", options=None,
other_options)
```

It creates a module from the CUDA source code called `kernel_source`. Then, the NVIDIA nvcc compiler is invoked with options to compile the code.

However, PyCUDA introduces the class `pycuda.gpuarray.GPUArray` that provides a high-level interface to perform calculations with CUDA:

```
class pycuda.gpuarray.GPUArray(shape, dtype, *, allocator=None,
order="C")
```

This works in a similar way to `numpy.ndarray`, which stores its data and performs its computations on the compute device. The `shape` and `dtype` arguments work exactly as in NumPy.

All the arithmetic methods in GPUArray support the broadcasting of scalars. The creation of `gpuarray` is quite easy. One way is to create a NumPy array and convert it, as shown in the following code:

```
>>> import pycuda.gpuarray as gpuarray
>>> from numpy.random import randn
>>> from numpy import float32, int32, array
>>> x = randn(5).astype(float32)
>>> x_gpu = gpuarray.to_gpu(x)
```

You can print `gpuarray` as you do normally:

```
>>> xarray([-0.24655211,  0.00344609,  1.45805557,  0.22002029,
1.28438667])
>>> x_gpuarray([-0.24655211,  0.00344609,  1.45805557,  0.22002029,
1.28438667])
```

How to do it...

The following example represents not only an easy introduction, but also a common use case of GPU computations, perhaps in the form of an auxiliary step between other calculations. The script for this is as follows:

```
import pycuda.gpuarray as gpuarray
import pycuda.driver as cuda
import pycuda.autoinit
import numpy

a_gpu = gpuarray.to_gpu(numpy.random.randn(4,4).astype(numpy.float32))
a_doubled = (2*a_gpu).get()
print a_doubled
print a_gpu
```

The output is (running the function from Python IDLE) as follows:

```
C \Python Parallel Programming INDEX\Chapter 6 - GPU Programming wit
h Python\python PyCudaGPUArray.py
ORIGINAL MATRIX
[[-0.60254627  1.16694951  1.48510635 -1.46718287  2.11878467]
 [ 2.63159704 -3.6541729   2.44197178 -1.12101364  0.22178674]
 [-0.87713826 -1.9803952   0.98741448 -2.83859134 -1.55612338]
 [ 0.79552311 -0.25934356 -1.12207913 -0.21778747 -4.0459609 ]
 [-1.74858582  1.34928024 -2.55908132  2.22259712  0.82242775]]

DOUBLED MATRIX AFTER PyCUDA EXECUTION USING GPUARRAY CALL
[[-0.30127314  0.58347476  0.74255317 -0.73359144  1.05939233]
 [ 1.31579852 -1.82708645  1.22098589 -0.56050682  0.11089337]
 [-0.43856913 -0.9901976   0.49370724 -1.41929567 -0.77806169]
 [ 0.39776155 -0.12967178 -0.56103957 -0.10889374 -2.02298045]
 [-0.87429291  0.67464012 -1.27954066  1.11129856  0.41121387]]
```

How it works...

Of course, we have to import all the required modules:

```
import pycuda.gpuarray as gpuarray
import pycuda.driver as cuda
import pycuda.autoinit
import numpy
```

The `a_gpu` input matrix contains all the items that are generated randomly. To perform the computation in the GPU, (double all the items in the matrix) we have only one statement:

```
a_doubled = (2*a_gpu).get()
```

The result is put in the `a_doubled` matrix (using the `get()` method). Finally, the result is printed as follows:

```
print a_doubled
```

There's more...

The `pycuda.gpuarray.GPUArray` supports all arithmetic operators and a number of methods and functions, all patterned after the corresponding functionality in NumPy. In addition to this, many special functions are available in `pycuda.cumath`. The arrays of approximately uniformly distributed random numbers may be generated using the functionality in `pycuda.curandom`.

Evaluating element-wise expressions with PyCUDA

The `PyCuda.elementwise.ElementwiseKernel` function allows us to execute the kernel on complex expressions that are made of one or more operands into a single computational step, which is as follows:

```
ElementwiseKernel(arguments,operation,name,optional_parameters)
```

Here, we note that:

- `arguments`: This is a C argument list of all the parameters that are involved in the kernel's execution.
- `operation`: This is the operation that is to be executed on the specified arguments. If the argument is a vector, each operation will be performed for each entry.
- `name`: This is the kernel's name.
- `optional_parameters`: These are the compilation directives that are not used in the following example.

How to do it...

In this example, we'll show you the typical use of the `ElementwiseKernel` call. We have two vectors of 50 elements, `input_vector_a` and `input_vector_b`, that are built in a random way. The task here is to evaluate their linear combination.

The code for this is as follows:

```
import pycuda.autoinit
import numpy
from pycuda.curandom import rand as curand
from pycuda.elementwise import ElementwiseKernel
import numpy.linalg as la


input_vector_a = curand((50,))
input_vector_b = curand((50,))
mult_coefficient_a = 2
mult_coefficient_b = 5


linear_combination = ElementwiseKernel(
        "float a, float *x, float b, float *y, float *c",
        "c[i] = a*x[i] + b*y[i]",
        "linear_combination")

linear_combination_result = gpuarray.empty_like(input_vector_a)
linear_combination(mult_coefficient_a, input_vector_a,\
                   mult_coefficient_b, input_vector_b,\
                   linear_combination_result)


print ("INPUT VECTOR A =")
print (input_vector_a)

print ("INPUT VECTOR B = ")
print (input_vector_b)

print ("RESULTING VECTOR C = ")
print linear_combination_result

print ("CHECKING THE RESULT EVALUATING THE DIFFERENCE VECTOR BETWEEN C
AND THE LINEAR COMBINATION OF A AND B")
print ("C - (%sA + %sB) = "%(mult_coefficient_a,mult_coefficient_b))
print (linear_combination_result - (mult_coefficient_a*input_vector_a\
                                    + mult_coefficient_b*input_
vector_b))
assert la.norm((linear_combination_result - \
                (mult_coefficient_a*input_vector_a +\
                 mult_coefficient_b*input_vector_b)).get()) < 1e-5
```

The output for this from Command Prompt is as follows:

```
C:\Python CookBook\Chapter 6 - GPU Programming with Python\ >python
PyCudaElementWise.py
INPUT VECTOR A =
[ 0.73191601  0.7004351   0.87159222  0.49621502  0.19640177
0.75579387
  0.35208538  0.97497243  0.36948711  0.34328628  0.06811771
0.04270195
  0.15690483  0.39899695  0.2927697   0.36201504  0.09503061
0.45646626
  0.35608584  0.01598917  0.75943208  0.49343511  0.79146844
0.33111155
  0.18454118  0.83971804  0.01466237  0.77959627  0.54659295
0.4575595
  0.55539894  0.23285247  0.14676388  0.72028935  0.87861985
0.13928016
  0.18071586  0.8029055   0.05551658  0.49400434  0.40941685
0.55373788
  0.07541087  0.55443048  0.19723719  0.72457349  0.46491891
0.65380263
  0.93845034  0.27472526]
INPUT VECTOR B =
[ 0.29464501  0.21645674  0.93407696  0.48678038  0.71135205
0.0588627
  0.99216938  0.879906    0.07517455  0.84360296  0.57358545
0.73907417
  0.06841258  0.1816148   0.53327322  0.30980903  0.96774238
0.90884209
  0.39139062  0.97678316  0.41284555  0.17893282  0.47421032
0.13706622
  0.62038481  0.22524452  0.67131585  0.06617502  0.02492006
0.99894243
  0.28288943  0.55505407  0.14323047  0.54854101  0.2742492
0.01146096
  0.45902726  0.03561942  0.78358203  0.32014725  0.13187674
0.42909116
  0.2633251   0.07679776  0.80823648  0.57373965  0.40740359
0.26024994
  0.61452144  0.46388686]
```

```
RESULTING VECTOR C =
[ 2.93705702  2.48315382  6.41356945  3.42633176  3.94956398
1.80590129
  5.6650176   6.34947491  1.11484694  4.90458727  3.00416279
3.78077483
  0.65587258  1.70606792  3.25190544  2.2730751   5.02877283
5.45714283
  2.6691246   4.91589403  3.58309197  1.88153434  3.95398855
1.34755421
  3.47100639  2.80565882  3.38590407  1.89006758  1.21778619
5.90983152
  2.52524495  3.24097538  1.00968003  4.18328381  3.12848568
0.33586511
  2.65656805  1.78390813  4.02894306  2.58874488  1.47821736
3.25293159
  1.46744728  1.49284983  4.43565702  4.31784534  2.96685553
2.60885501
  4.94950771  2.86888456]
CHECKING THE RESULT EVALUATING THE DIFFERENCE VECTOR BETWEEN C AND THE
LINEAR COMBINATION OF A AND B
C - (2A + 5B) =
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

How it works...

After the usual import, we note:

```
from pycuda.elementwise import ElementwiseKernel
```

We must build all the elements that are to be manipulated. Let's remember that the task to be done is to evaluate a linear combination of two vectors `input_vector_a` and `input_vector_b`. These two vectors are initialized using the PyCUDA `curandom` library, which is used for the generation of pseudorandom numbers:

To import the library, use the following code:

```
from pycuda.curandom import rand as curand
```

To define the random vector (50 elements), use:

```
input_vector_a = curand((50,))
input_vector_b = curand((50,))
```

We defined the two coefficients of multiplication that are to be used in the calculation of the linear combination of these two vectors:

```
mult_coefficient_a = 2
mult_coefficient_b = 5
```

The core example is the kernel invocation for which we use the PyCUDA `ElementwiseKernel` construct, shown as follows:

```
linear_combination = ElementwiseKernel(
        "float a, float *x, float b, float *y, float *c",
        "c[i] = a*x[i] + b*y[i]",
        "linear_combination")
```

The first line of the argument list (in a C-style definition) defines all the parameters to be inserted for the calculation:

```
        "float a, float *x, float b, float *y, float *c",
```

The second line defines how to manipulate the arguments list. For each value of the index `i`, a sum of these components must be evaluated:

```
  "c[i] = a*x[i] + b*y[i]",
```

The last line gives the `linear_combination` name to `ElementwiseKernel`.

After the kernel, the resulting vector is defined. It is an empty vector of the same dimension as of the input vector:

```
linear_combination_result = gpuarray.empty_like(input_vector_a)
Finally evaluate the kernel:
linear_combination(mult_coefficient_a, input_vector_a,\
                   mult_coefficient_b, input_vector_b,\
                   linear_combination_result)
```

You can check the results using the following code:

```
assert la.norm((linear_combination_result - \
                (mult_coefficient_a*input_vector_a +\
                 mult_coefficient_b*input_vector_b)).get()) < 1e-5
```

The `assert` function tests the result and triggers an error if the condition is `false`.

There's more...

In addition to the `curand` library, derived from the CUDA library, PyCUDA provides other math libraries, so you can take a look at the libraries listed at `http://documen.tician.de/pycuda`.

The MapReduce operation with PyCUDA

PyCUDA provides a functionality to perform reduction operations on the GPU. This is possible with the `pycuda.reduction.ReductionKernel` method:

```
ReductionKernel(dtype_out, arguments, map_expr ,reduce_expr,
                name,optional_parameters)
```

Here, we note that:

- `dtype_out`: This is the output's data type. It must be specified by the `numpy.dtype` data type.
- `arguments`: This is a C argument list of all the parameters involved in the reduction's operation.
- `map_expr`: This is a string that represents the mapping operation. Each vector in this expression must be referenced with the variable `i`.
- `reduce_expr`: This is a string that represents the reduction operation. The operands in this expression are indicated by lowercase letters, such as `a, b, c, ..., z`.
- `name`: This is the name associated with `ReductionKernel`, with which the kernel is compiled.
- `optional_parameters`: These are not important in this recipe as they are the compiler's directives.

The method executes a kernel on vector arguments (at least one), performs `map_expr` on each entry of the vector argument, and then performs `reduce_expr` on its outcome.

How to do it…

This example shows the implementation of a dot product of two vectors (500 elements) through an instantiation of the `ReductionKernel` class. The dot product, or scalar product, is an algebraic operation that takes two equal length sequences of numbers (usually coordinate vectors) and returns a single number that is the sum of the products of the corresponding entries of the two sequences of numbers. This is a typical MapReduce operation, where the Map operation is an index-by-index product and the reduction operation is the sum of all the products.

The PyCUDA code for this task is very short:

```
import pycuda.gpuarray as gpuarray
import pycuda.autoinit
import numpy
from pycuda.reduction import ReductionKernel

vector_length = 400
```

```
input_vector_a = gpuarray.arange(vector_length, dtype=numpy.int)
input_vector_b = gpuarray.arange(vector_length, dtype=numpy.int)
dot_product = ReductionKernel(numpy.int,
                       arguments="int *x, int *y",
                       map_expr="x[i]*y[i]",
                       reduce_expr="a+b", neutral="0")

dot_product = dot_product (input_vector_a, input_vector_b).get()

print("INPUT VECTOR A")
print input_vector_a

print("INPUT VECTOR B")
print input_vector_b

print("RESULT DOT PRODUCT OF A * B")
print dot_product
```

Running the code from Command Prompt, you will have an output like this:

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



INPUT VECTOR A
[  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53
  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71
  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89
  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107
 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125
 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161
 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179
 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197
 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215
 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233
 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251
 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269
 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287
 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305
 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323
 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341
 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359
 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377
 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395
 396 397 398 399]

INPUT VECTOR B
[  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53
  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71
  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89
  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107
 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125
 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161
 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179
 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197
 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215
 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233
 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251
 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269
 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287
 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305
 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323
 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341
 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359
 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377
 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395
 396 397 398 399]

RESULT DOT PRODUCT OF A * B
21253400
```

How it works...

In this script, the input vectors `input_vector_a` and `input_vector_b` are integer vectors. Each of them, as you can see from the preceding output result, ranges from `0` to `399` elements (400 elements in total):

```
vector_length = 400

input_vector_a = gpuarray.arange(vector_length, dtype=numpy.int)
input_vector_b = gpuarray.arange(vector_length, dtype=numpy.int)
```

After the definition of the inputs, we can define the MapReduce operation by calling the `ReductionKernel` PyCUDA function:

```
dot_product = ReductionKernel(numpy.int,
                       arguments="int *x, int *y",
                       map_expr="x[i]*y[i]",
                       reduce_expr="a+b", neutral="0")
```

This kernel operation is defined as follows:

- The first entry in the argument list tells us that the output will be an integer
- The second entry defines the data types for the inputs (array of integers) in a C-like notation
- The third entry is the map operation, which is the product of the *i*th element of the two vectors
- The fourth operation is the reduction operation, which is the sum of all the products

Observe that the end result of calling the `ReductionKernel` instance is a `GPUArray` scalar that still resides in the GPU. It can be brought to the CPU by a call to its `get` method or can be used in place of the GPU.

Then, the kernel function is invocated, as shown:

```
dot_product = dot_product (input_vector_a, input_vector_b).get()
```

The input vectors and the resulting dot product are printed out:

```
print input_vector_a
print input_vector_b
print dot_product
```

GPU programming with NumbaPro

NumbaPro is a Python compiler that provides a CUDA-based API to write CUDA programs. It is designed for array-oriented computing tasks, much like the widely used NumPy library. The data parallelism in array-oriented computing tasks is a natural fit for accelerators such as GPUs. NumbaPro understands NumPy array types and uses them to generate efficient compiled code for execution on GPUs or multicore CPUs.

The compiler works by allowing you to specify type signatures for Python functions, which enable compilation at runtime (called the JIT compilation).

The most important decorators are:

- `numbapro.jit`: This allows a developer to write CUDA-like functions. When encountered, the compiler translates the code under the decorator into the pseudo assembly PTX language to be executed in the GPU.
- `numbapro.autojit`: This annotates a function for a deferred compilation procedure. This means that each function with this signature is compiled exactly once.
- `numbapro.vectorize`: This creates a so-called `ufunc` object (the Numpy universal function) that takes a function and executes it parallelly in vector arguments.
- `guvectorize`: This creates a so-called `gufunc` object (the NumPy generalized universal function). A `gufunc` object may operate on entire subarrays (refer to `http://docs.continuum.io/numbapro/generalizedufuncs.html` for more references.)

All these decorators have a compiler directive called a target that selects the code generation target. The NumbaPro compiler supports the parallel and GPU targets. The parallel target is available to vectorize the operations, while the GPU directive offloads the computation to a NVIDIA CUDA GPU.

Getting ready

NumbaPro is part of Anaconda Accelerate, which is a commercially licensed product (NumbaPro is also available under a free license for academic users) from Continuum Analytics. It is built on top of the BSD-licensed, open source Numba project, which itself relies heavily on the capabilities of the LLVM compiler. The GPU backend of NumbaPro utilizes the LLVM-based NVIDIA Compiler SDK.

To get started with NumbaPro, the first step is to download and install the Anaconda Python distribution (`http://continuum.io/downloads`), which is a completely free, enterprise-ready Python distribution for large-scale data processing, predictive analytics, and scientific computing. It includes many popular packages (Numpy, Scipy, Matplotlib, iPython, and so on) and `conda`, which is a powerful package manager.

Once you have Anaconda installed, you must type the following instructions from Anaconda's Command Prompt:

```
> conda update conda
> conda install accelerate
> conda install numbapro
```

NumbaPro does not ship the CUDA driver. It is the user's responsibility to ensure that their systems are using the latest drivers. After the installation, it's possible to perform the detection of the CUDA library and GPU, so let's open Python from the Anaconda console and type:

```
import numbapro
numbapro.check_cuda()
```

The output of these two lines of code should be as follows (we used a 64-bit Anaconda distro):

```
C:\Users\Giancarlo\Anaconda>python
Python 2.7.10 |Anaconda 2.3.0 (64-bit)| (default, May 28 2015, 16:44:52)
[MSC v.1500 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://binstar.org
>>> import numbapro
Vendor:  Continuum Analytics, Inc.
Package: mkl
Message: trial mode expires in 30 days
Vendor:  Continuum Analytics, Inc.
Package: mkl
Message: trial mode expires in 30 days
Vendor:  Continuum Analytics, Inc.
Package: numbapro
Message: trial mode expires in 30 days
>>> numbapro.check_cuda()
------------------------------libraries detection------------------------
```

```
Finding cublas
        located at C:\Users\Giancarlo\Anaconda\DLLs\cublas64_60.dll
        trying to open library...       ok
Finding cusparse
        located at C:\Users\Giancarlo\Anaconda\DLLs\cusparse64_60.dll
        trying to open library...       ok
Finding cufft
        located at C:\Users\Giancarlo\Anaconda\DLLs\cufft64_60.dll
        trying to open library...       ok
Finding curand
        located at C:\Users\Giancarlo\Anaconda\DLLs\curand64_60.dll
        trying to open library...       ok
Finding nvvm
        located at C:\Users\Giancarlo\Anaconda\DLLs\nvvm64_20_0.dll
        trying to open library...       ok
        finding libdevice for compute_20...     ok
        finding libdevice for compute_30...     ok
        finding libdevice for compute_35...     ok
-------------------------------hardware detection------------------------
Found 1 CUDA devices
id 0            GeForce 840M                            [SUPPORTED]
                      compute capability: 5.0
                           pci device id: 0
                              pci bus id: 8
Summary:
        1/1 devices are supported
PASSED
True
>>>
```

How to do it...

In this example, we give a demonstration of the NumbaPro compiler using the annotation `@guvectorize`. In the following task, we try to execute a matrix multiplication using the Numbapro module:

```
from numbapro import guvectorize
import numpy as np

@guvectorize(['void(int64[:,:], int64[:,:], int64[:,:])'],
             '(m,n),(n,p)->(m,p)')
def matmul(A, B, C):
    m, n = A.shape
    n, p = B.shape
    for i in range(m):
        for j in range(p):
            C[i, j] = 0
            for k in range(n):
                C[i, j] += A[i, k] * B[k, j]

dim = 10
A = np.random.randint(dim,size=(dim, dim))
B = np.random.randint(dim,size=(dim, dim))


C = matmul(A, B)
print("INPUT MATRIX A")
print(":\n%s" % A)
print("INPUT MATRIX B")
print(":\n%s" % B)
print("RESULT MATRIX C = A*B")
print(":\n%s" % C)
```

After running the code (using the Anaconda console), we should have an output like this:

```
INPUT MATRIX A
:
[[7 7 8 5 8 5 1 9 5 9]
 [3 5 5 4 6 7 6 5 3 1]
 [7 1 6 8 7 9 0 3 3 3]
 [7 4 4 3 7 8 1 2 1 2]
 [4 7 7 1 3 5 5 6 7 6]
 [5 0 1 5 8 4 4 4 4 9]
 [1 3 2 0 7 3 7 2 3 4]
 [0 2 9 0 7 5 9 7 4 7]
 [7 3 7 6 5 6 4 2 2 7]
 [2 1 9 7 1 0 3 5 7 3]]
INPUT MATRIX B
:
[[2 9 8 4 2 3 9 7 3 1]
 [9 1 3 3 8 0 7 6 3 5]
 [7 4 9 6 6 5 9 7 6 6]
 [6 8 3 1 5 4 4 7 7 5]
 [6 2 5 1 2 8 6 0 5 8]
 [4 4 5 7 6 0 1 1 3 8]
 [2 7 8 6 1 9 8 4 1 6]
 [2 2 9 8 3 6 1 4 7 4]
 [9 9 6 9 3 3 3 2 4 9]
 [8 4 6 7 8 8 8 6 7 8]]

RESULT MATRIX C = A*B
:
[[368 284 402 331 304 295 361 291 327 378]
 [231 207 278 226 188 199 236 177 193 273]
 [248 247 280 217 208 190 243 198 232 279]
 [201 181 232 175 173 149 218 156 170 225]
 [297 239 331 301 239 225 290 225 229 315]
 [235 229 270 222 181 248 246 175 219 280]
 [174 142 201 166 124 185 192 108 129 217]
 [267 213 348 297 212 292 289 194 233 334]
 [266 254 305 239 228 230 303 234 232 288]
 [227 219 255 215 166 189 214 196 204 229]]
```

How it works...

The `@guvectorize` annotation works on array arguments. This decorator takes an extra argument to specify the `gufunc` signature. The arguments are explained, as follows:

- The first three arguments specify the types of data to be managed, which are the array of integers: `'void(int64[:,:], int64[:,:], int64[:,:])'`
- The last argument of `@guvectorize` specifies how to manipulate the matrix dimensions: `'(m,n),(n,p)->(m,p)'`

```
@guvectorize(['void(int64[:,:], int64[:,:], int64[:,:])'],
        '(m,n),(n,p)->(m,p)')
```

In the subsequent code, we define the `matmul(A, B, C)` operation. It accepts the two input matrix `A` and `B` and produces a `C` output matrix. According to the `gufunc` signature, we should have:

```
A(m,n)* B(n,p) = C(m,p) where m,n,p are the matrix dimensions.
```

The matrix product is simply performed via three `for` loops along with the matrix indices:

```
          for i in range(m):
                 for j in range(p):
                    C[i, j] = 0
                     for k in range(n):
                         C[i, j] += A[i, k] * B[k, j]
```

The Numpy's function `randint` is used to build integers from random matrices:

```
dim = 10
A = np.random.randint(dim,size=(dim, dim))
B = np.random.randint(dim,size=(dim, dim))
```

Finally, the `matmul` function is called with these matrices with arguments, and the resultant matrix is printed out:

```
   C = matmul(A, B)
   print("RESULT MATRIX C = A*B")
        print(":\n%s" % C)
```

Using GPU-accelerated libraries with NumbaPro

NumbaPro provides a Python wrap for CUDA libraries for numerical computing. Each code using these libraries will get a significant speedup without writing any GPU-specific code. The libraries are explained as follows:

- **cuBLAS**: This is a library developed by NVIDIA that provides the main functions of linear algebra to run on a GPU. Like the **Basic Linear Algebra Subprograms** (**BLAS**) library that implements the functions of linear algebra on the CPU, the cuBLAS library classifies its functions into three levels:
    - **Level 1**: Vector operations
    - **Level 2**: Transactions between a matrix and vector
    - **Level 3**: Operations between matrices

The division of these functions in the three levels is based on the number of nested loops that are needed to perform the selected operation. More precisely, the operations of the level are essential cycles that are geared to complete the execution of the selected function.

- **cuFFT**: This provides a simple interface to calculate the **Fast Fourier Transform** (**FFT**) in a distributed manner on an NVIDIA GPU, enabling you to exploit the parallelism of the GPU without having to develop your own implementation of the FFT.
- **cuRAND**: This library provides the creation of quasirandom numbers. A quasirandom number is a random number generated by a deterministic algorithm.
- **cuSPArse**: This provides a set of functions for the management of sparse matrices. Unlike the previous case, its functions are classified into four levels:
    - **Level 1**: These are operations between a vector that is stored in a shed and a vector that is stored in a dense format.
    - **Level 2**: These are the transactions between a matrix format stored in a shed and a vector stored in the dense format.
    - **Level 3**: These are the operations in a matrix format that are stored in a shed and set of vectors that are stored in a dense format (this set can be considered as one large dense matrix.)
    - **Conversion**: These are operations that allow the conversion between different storage formats.

How to do it…

In this example, we present an implementation of **GEneral Matrix Multiply** (**GEMM**), which is a routine to perform matrix-matrix multiplication on NVIDIA GPUs. The sequential version using the NumPy Python module and the parallel version using the cuBLAS library will be reported. Also, a comparison of the execution time will be made between the two algorithms.

The code for this is as follows:

```
import numbapro.cudalib.cublas as cublas
import numpy as np
from timeit import default_timer as timer

dim = 10

def gemm():
    print("Version 2".center(80, '='))

    A = np.random.rand(dim,dim)
    B = np.random.rand(dim, dim)
```

```
        D = np.zeros_like(A, order='F')

        print("MATRIX A :")
        print A
        print("VECTOR B :")
        print B

        # NumPy
        start = timer()
        E = np.dot(A, B)
        numpy_time = timer() - start
        print("Numpy took %f seconds" % numpy_time)

        # cuBLAS
        blas = cublas.Blas()

        start = timer()
        blas.gemm('T', 'T', dim, dim, dim, 1.0, A, B, 1.0, D)
        cuda_time = timer() - start
        print ("RESULT MATRIX EVALUATED WITH CUBLAS")
        print D
        print("CUBLAS took %f seconds" % cuda_time)
        diff = np.abs(D - E)
        print("Maximum error %f" % np.max(diff))


    def main():

        gemm()

    if __name__ == '__main__':
       main()
```

The output obtained for this will be as follows:

```
MATRIX A :
[[ 0.79582178  0.95671563  0.69251157  0.85600979  0.32826726  0.72861569
   0.20724061  0.55065641  0.2257875   0.90146437]
 [ 0.6742022   0.43449657  0.04862685  0.9023226   0.87598306  0.20774405
   0.15774015  0.2847742   0.81601615  0.34114773]
 [ 0.61500219  0.65982283  0.73493152  0.21913261  0.80862566  0.73982082
   0.84005388  0.38745489  0.676947    0.31530397]
 [ 0.60694411  0.65138528  0.63773284  0.06589098  0.49177294  0.02029247
   0.9064746   0.93419845  0.14609622  0.28317855]
 [ 0.60166404  0.41423776  0.09938464  0.19315303  0.07374789  0.45335697
   0.2912572   0.81481984  0.65222424  0.0670377 ]
 [ 0.32192297  0.30244072  0.86595209  0.37701833  0.79095644  0.11518194
   0.88491826  0.98290063  0.62965353  0.38323725]
 [ 0.21512101  0.64731098  0.4079146   0.8371392   0.01398673  0.85945652
   0.0586854   0.48812094  0.3625991   0.58142603]
 [ 0.77378663  0.43994483  0.5620805   0.70350504  0.60589009  0.09605428
   0.25423268  0.06869655  0.13642323  0.00221422]
 [ 0.77808301  0.47386303  0.54323866  0.42010733  0.80652762  0.05903843
   0.63316824  0.58479485  0.45141828  0.46231481]
 [ 0.97122802  0.53723365  0.68688748  0.54315409  0.00883411  0.9855186
   0.53542786  0.83478941  0.27459888  0.21024639]]
VECTOR B :
[[ 0.17084153  0.44546677  0.21551063  0.39731923  0.00102686  0.81069924
   0.00681474  0.01126972  0.13769525  0.63437229]
 [ 0.81913609  0.97583768  0.52579565  0.20179695  0.24066758  0.18154282
   0.75033104  0.41878918  0.96892428  0.54358419]
 [ 0.10071768  0.3090773   0.94185921  0.70550442  0.10651627  0.62659408
   0.23255164  0.96166165  0.65615938  0.16991118]
 [ 0.84163163  0.59296382  0.12281989  0.32851275  0.78716318  0.02568872
   0.02367708  0.65485736  0.79834789  0.76747705]
 [ 0.90406949  0.03424157  0.01519989  0.5011444   0.63175281  0.17705116
   0.16257016  0.81357471  0.58567631  0.24503327]
 [ 0.62989968  0.47944669  0.86860435  0.94086568  0.24312278  0.13450463
   0.16352136  0.42323191  0.46907905  0.97772097]
 [ 0.44608094  0.19969488  0.01035155  0.69528549  0.07219375  0.91454669
   0.18330497  0.76095336  0.12880003  0.24301603]
 [ 0.37860881  0.33079438  0.19275564  0.58316669  0.35753971  0.63697732
   0.72063491  0.42698316  0.53811423  0.83682958]
 [ 0.42135462  0.89413827  0.00620849  0.63770542  0.29376823  0.68415057
   0.71826696  0.9748898   0.9086774   0.7084634 ]
 [ 0.08020851  0.47789158  0.45538401  0.26468263  0.84960276  0.1108932
   0.0407631   0.41811299  0.2539022   0.73346706]]
```

```
Numpy took 1.167435 seconds
RESULT MATRIX EVALUATED WITH CUBLAS
[[ 2.93393517  3.22653293  2.58999843  2.97688025  2.40723642  2.22561846
   1.71083261  3.20145366  3.4654546   3.9246803 ]
 [ 2.70759988  2.42236864  0.94108333  2.20715685  2.06739391  1.78390442
   1.37381915  2.80760808  2.87826551  2.88739456]
 [ 2.93301949  2.70921232  2.08465713  3.39447429  1.76684939  2.84034554
   1.8600905   3.70096673  3.21368161  3.20257798]
 [ 2.05665894  1.92477247  1.42646422  2.45288009  1.27576149  2.65682509
   1.68187918  2.6942483   2.30742661  2.35163885]
 [ 1.68553937  1.98030198  1.05436088  2.03107385  0.98066787  1.94328559
   1.54050405  1.8876191   2.04514196  2.49719893]
 [ 2.55782414  2.2600454   1.57942935  3.11991574  1.91570669  2.93236718
   1.92525406  3.76932667  3.03618471  2.87628333]
 [ 2.27705425  2.53777179  1.98218876  2.30511984  1.85547257  1.36423334
   1.39131705  2.43879465  2.75148098  3.14994564]
 [ 1.94662205  1.62822264  1.12425671  1.72230283  1.21131853  1.56748417
   0.79113948  2.08449619  2.05742732  1.82536594]
 [ 2.42686338  2.22641127  1.3762425   2.57727754  1.80747335  2.53040609
   1.51847658  3.05078902  2.68199133  2.72340269]
 [ 2.44854528  2.69315101  2.3255071   3.17886105  1.47260987  2.69597578
   1.65043895  2.79595207  2.82714486  3.58489296]]
CUBLAS took 0.004226 seconds
Maximum error 0.000000
```

The result obtained confirms the effectiveness of the cuBLAS library.

How it works...

In order to make a comparison between a NumPy and cuBLAS implementation of a matrix product, we import all the required libraries:

```
import numbapro.cudalib.cublas as cublas
import numpy as np
```

Also, we define the matrix dimension:

```
dim = 10
```

The core algorithm is the `gemm()` function. First, we define the input matrices:

```
A = np.random.rand(dim,dim)
B = np.random.rand(dim,dim)
```

Here, `D` will contain the output of the cuBLAS implementation:

```
    D = np.zeros_like(A, order='F')
```

In this example, we compare the calculation done with NumPy and cuBLAS. The NumPy evaluation is: `E = np.dot(A,B)`, where the matrix `E` will contain the dot product.

Finally, the cuBLAS implementation is as follows:

```
 blas = cublas.Blas()
     start = timer()
     blas.gemm('T', 'T', dim, dim, dim, 1.0, A, B, 1.0, D)
     cuda_time = timer() - start
```

The `gemm()` function is a cuBLAS level 3 function:

```
numbapro.cudalib.cublas.Blas.gemm(transa, transb, m, n, k, alpha,
                                  A, B,beta, C)
```

It realizes a matrix-matrix multiplication in the following form:

```
C = alpha * op(A) * op(B) + beta * C where op is transpose or not.
```

At the end of the function, we compare the two results and report the execution time (`cuda_time`):

```
    print("CUBLAS took %f seconds" % cuda_time)
        diff = np.abs(D - E)
        print("Maximum error %f" % np.max(diff))
```

There's more...

In this example, we saw an application of the cuBLAS library. For more complete references, refer to `http://docs.nvidia.com/cuda/cublas/index.html` and `http://docs.continuum.io/numbapro/cudalib` for a complete list of CUDA function libraries wrapped with NumbaPro.

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

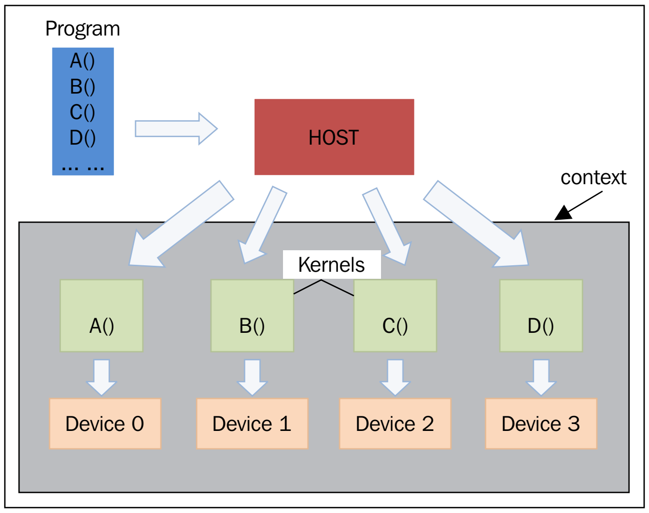
How to build a PyOpenCL application

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.

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`.