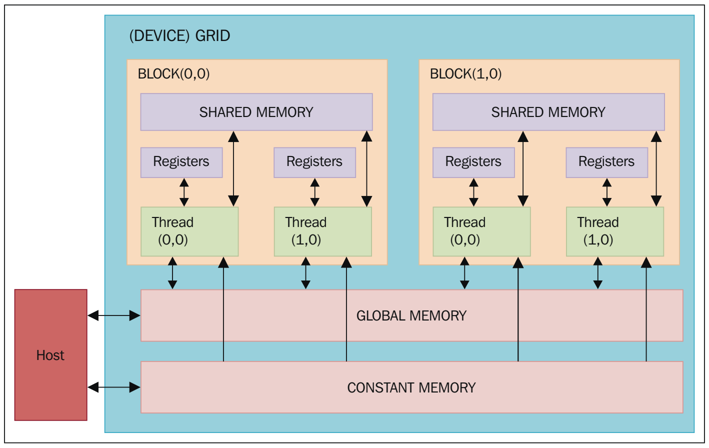
There's more...

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`.