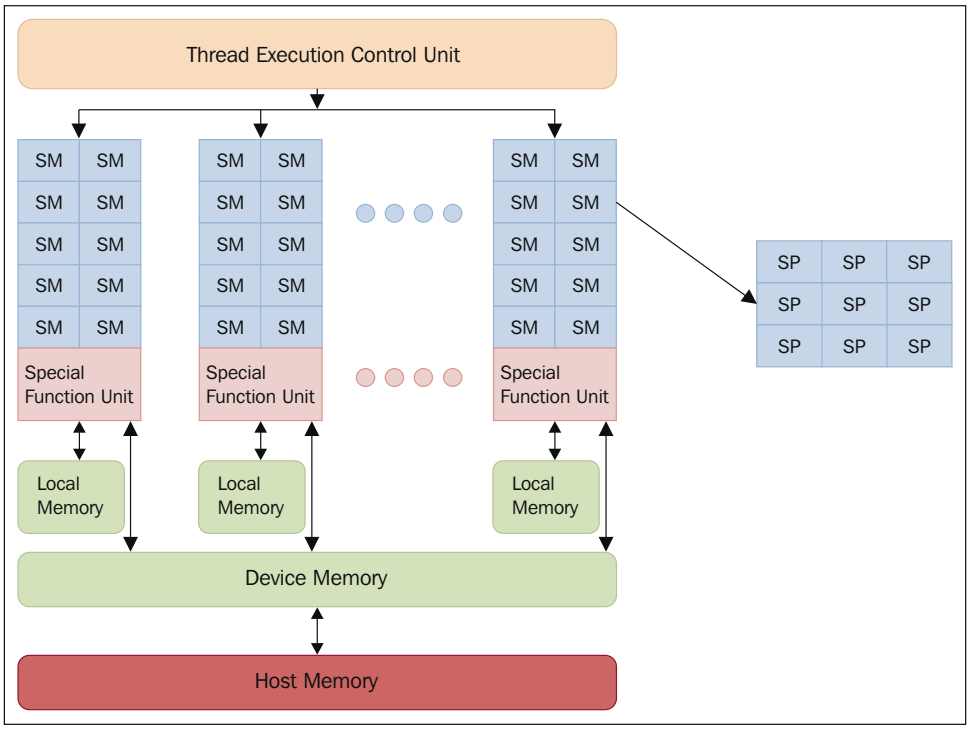
6
GPU Programming with Python

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

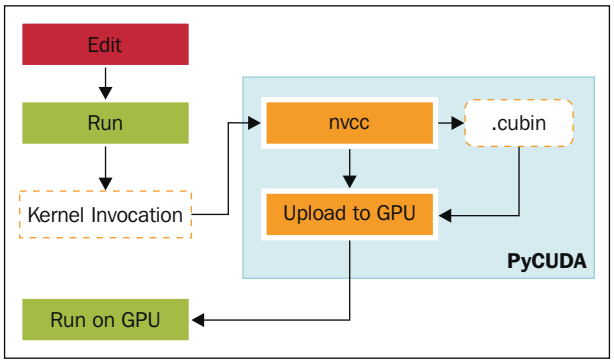
PyCUDA is a Python wrap for **Compute Unified Device Architecture** (**CUDA**), the software library developed by NVIDIA for GPU programming. The CUDA programming model is the starting point of understanding how to program the GPU properly with PyCUDA. There are concepts that must be understood and assimilated 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.

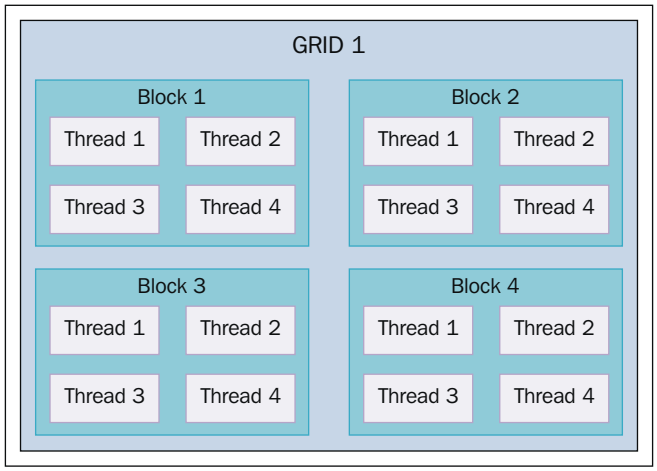
The compiled CUDA code is converted to a device-specific binary by the driver, during runtime. All the previously mentioned steps are executed by PyCUDA at runtime, which makes it a **Just-in-time** (**JIT**) compiler. The drawback of this approach is the increased load time of the application, which 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:

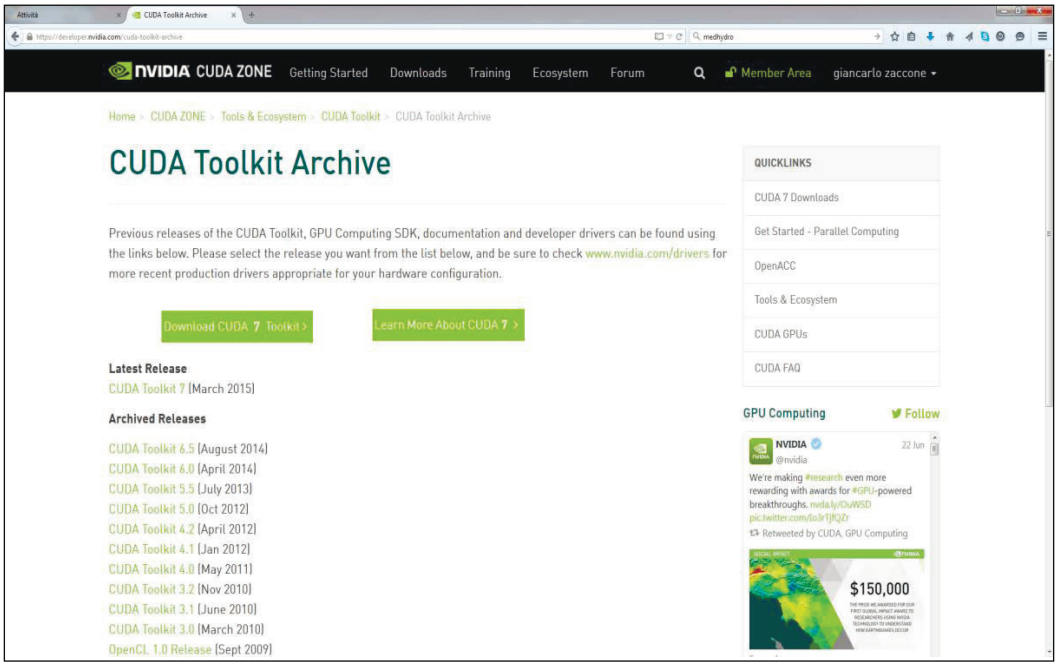
1. The first step is to download and install all the components provided by NVDIA to develop with CUDA (refer to `https://developer.nvidia.com/cuda-toolkit-archive`) for all the available versions. These components are:
   - The CUDA toolkit is available at `http://developer.download.nvidia.com/compute/cuda/4_2/rel/toolkit/cudatoolkit_4.2.9_win_32.msi`.
   - 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
```



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))
```

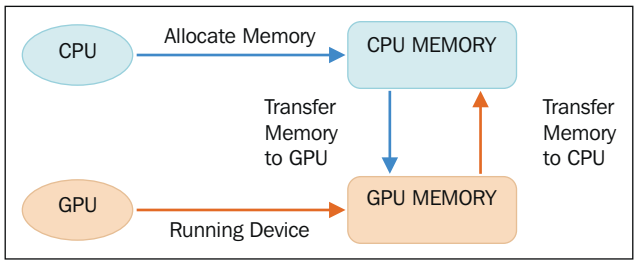
See also

- 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:

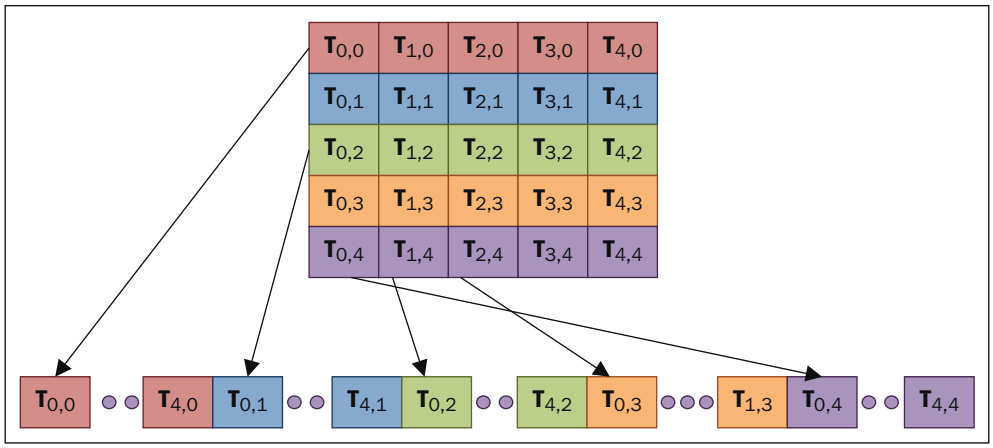
```
int idx = threadIdx.x + threadIdx.y*4;
```

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`.



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
```