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.