CUDA Grid and Blocks tutorial

Contents

Goals of this tutorial

1. test connection to computer server for running this and future tutorials
2. Learn how to run CUDA kernels from python using PyCuda module
3. Understanding grid and blocks organization of CUDA kernel execution

Connect to tutorial compute server

if you don't have a INRIA Rennes account

- Go to this page : Connect to parallel server soroban

Else

- Go to this page for connect to igrida computer node connect to igrida gpu node

try to run a first Script

Here you can find a python code:

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy

from pycuda.compiler import SourceModule


# Kernel definition and on-line compilation
mod = SourceModule("""
__global__ void mykernel(float *destination, float *source, float b)
{
  const int index = threadIdx.x;
  destination[index] = source[index]* b;
}
""")

mykernel = mod.get_function("mykernel")

# size of data
num_data= 400
# Any arbitrary value for this exercise
my_constant = 123.0

# number of blocks of thread
num_blocks = 1
```

```
# number of threads in a block
block_size = num_data

# produce a data set for this exercise
source = numpy.random.randn(num_data).astype(numpy.float32)

destination = numpy.empty_like(source)

# kernel call with implicit data transfer to the GPU and return result
mykernel(
        drv.Out(destination), drv.In(source),numpy.float32(my_constant),
        block=(block_size,1,1), grid=(num_blocks,1))

# CPU compute same result and display the difference with GPU computing result
print (destination-source*my_constant)
```

Save this script in your account as name *tp_blockAndGrid.py* Them you can try to run on the machine:

```
andradgu@soroban:~/tp_cuda$ python tp_blockAndGrid.py
[ 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.  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.  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.  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.  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.  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.  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.  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.]
```

Running in other GPU

pycuda.autoinit module in script automatically initialize device #0 context for kernels executions. If you have another card on PC and you want to run your script in this card, you can set and environment variable **CUDA_DEVICE** with the number of device needed.

For instants, if you want to run your script in bash console in **second card**, you can run like this:

```
$ CUDA_DEVICE=1 python myCudaScript.py
```

Understanding What this script do

This script run a kernel named *mykernel* that takes a data array *source* of 400 float numbers in

input and multiply every elements by a scalar constant *b*. Them store the result in a *destination* array of the same size. The source data are generate in CPU using numpy API of random number. Implicit copy of data from source in Host memory to GPU memory are performed with call to **drv.In(source)**. Implicity copy of result from GPU memory to Host memory is done by **drv.Out(destination)**. So is possible to compare GPU result to the result of CPU computing using numpy API: **print (destination-source*my_constant)**

All the job is done by one single block of 400 threads. every thread execute a single load, multiplication and store from source to destination. This is possible because index is a bijective function of threadIdx.x that is unique index for all thread in linear block.

Data position	**threadIdx.x**	**value of *index***
0	0	0
1	1	1
...	...	...
398	398	398
399	399	399

You can try to change size of data by changing the value of **num_data**. But because there is a limit on block size (here 1024 threads per block max) you can not process problems more bigs that some floats.

Adapt the code to compute more data with more blocks

To use more blocks in a grid, you can change variable **num_blocks**. For instance, if you need to work with 8000 floats, you can need a grid of 20 blocks of 400 threads each.

Every block will work with a unique pack of data of *blockDim.x* elements (=400).

To do that, **you need to change kernel to set a global index of data elements**.

Next table help you to change the formula of index:

Data position	threadIdx.x	BlockIdx.x	value of *index* needed
0	0	0	0
1	1	0	1
...	...	...	...
399	399	0	399
400	0	1	400
401	1	1	401
...	...	...	...
798	398	1	798
799	399	1	799
800	0	2	800
801	1	2	801
...	...	...	...

Automatic define blocks number in function of data size

And easy way to automatic define the number of block in grid that you need is to set :

```
num_blocks = num_data / block_size
```

But if your data not fit exactly well on a integer number of blocks? For instance, if you have 11173 floats in data, you don't fit it with any size and number of blocks because 11173 is a prime number ! So to process all data, you need to set more blocks with a total of threads greater that the number of data to process. There is a "magic" formula to compute this number of blocks:

```
num_blocks = (num_data+block_size-1)/block_size
```

For a block of 400 threads we need 28 blocks in a grid to process all 11173 floats in data. So in the last block the last 27 threads are not needed. To switch off this threads:

1. Add a new parameter *int data_size* in kernel
2. Put **numpy.int32(num_data)** as value of this parameter in kernel call
3. Add a test using a global index of thread and *data_size* to detect last threads.

For finish, test different data size and block size.

Back to Parallel Computing tutorials