handson_gpu_2022

October 31, 2022

1 Setup Iniziale

1. Attivare il supporto GPU in Runtime->Change Runtime Type->Hardware Accelerator
2. Check if pyCUDA è installato

[ ]:
```
import pycuda
```

[ ]:
```
!pip install pycuda
```

[2]:
```
import pycuda
```

4. Controlla la versione di CUDA installata

[3]:
```
!nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2021 NVIDIA Corporation
Built on Sun_Feb_14_21:12:58_PST_2021
Cuda compilation tools, release 11.2, V11.2.152
Build cuda_11.2.r11.2/compiler.29618528_0
```

2 Esplorare la Bash

[5]:
```
!ls
```

```
drive  handson_gpu_2022_files  handson_gpu_2022.ipynb  sample_data
```

[6]:
```
mkdir test_dir
```

[7]:
```
cd test_dir
```

```
/content/test_dir
```

[ ]:
```
ls
```

[11]:
```
!touch ciao
```

[12]:
```
ls
```

```
ciao
```

[13]:
```
rm ciao
```

[16]:
```
ls
```

[17]:
```
pwd
```

[17]:
```
'/content/test_dir'
```

[18]:
```
cd ..
```

```
/content
```

[19]:
```
!gcc --version
```

```
gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

3 Caratteristiche della GPU in uso

Proviamo a capire le caratteristiche della GPU che abbiamo a disposizione. E' importante sapere la struttura della GPU per capire che operazioni possiamo fare.

[20]:
```
!nvidia-smi
```

```
Mon Oct 31 10:59:57 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   40C    P8     9W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
```

```
|        ID   ID                                                            Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

oppure si può usare il modulo pycuda, interrogando le funzioni del driver

[21]:
```
import pycuda.driver as drv
drv.init()
drv.get_version()
devn=drv.Device.count()
print("N GPU "+str(devn))
devices = []
for i in range(devn):
  devices.append(drv.Device(i))
for sp in devices:
  print("GPU name: "+str(sp.name))
  print("Compute Capability = "+str(sp.compute_capability()))
  print("Total Memory = "+str(sp.total_memory()/(2.**20))+' MBytes')
  attr = sp.get_attributes()
  print(attr)
```

```
N GPU 1
GPU name: <bound method name of <pycuda._driver.Device object at
0x7fcc247393b0>>
Compute Capability = (7, 5)
Total Memory = 15109.75 MBytes
{pycuda._driver.device_attribute.ASYNC_ENGINE_COUNT: 3,
pycuda._driver.device_attribute.CAN_MAP_HOST_MEMORY: 1,
pycuda._driver.device_attribute.CAN_USE_HOST_POINTER_FOR_REGISTERED_MEM: 1,
pycuda._driver.device_attribute.CLOCK_RATE: 1590000,
pycuda._driver.device_attribute.COMPUTE_CAPABILITY_MAJOR: 7,
pycuda._driver.device_attribute.COMPUTE_CAPABILITY_MINOR: 5,
pycuda._driver.device_attribute.COMPUTE_MODE:
pycuda._driver.compute_mode.DEFAULT,
pycuda._driver.device_attribute.COMPUTE_PREEMPTION_SUPPORTED: 1,
pycuda._driver.device_attribute.CONCURRENT_KERNELS: 1,
pycuda._driver.device_attribute.CONCURRENT_MANAGED_ACCESS: 1,
pycuda._driver.device_attribute.DIRECT_MANAGED_MEM_ACCESS_FROM_HOST: 0,
pycuda._driver.device_attribute.ECC_ENABLED: 1,
pycuda._driver.device_attribute.GENERIC_COMPRESSION_SUPPORTED: 0,
pycuda._driver.device_attribute.GLOBAL_L1_CACHE_SUPPORTED: 1,
pycuda._driver.device_attribute.GLOBAL_MEMORY_BUS_WIDTH: 256,
pycuda._driver.device_attribute.GPU_OVERLAP: 1,
pycuda._driver.device_attribute.HANDLE_TYPE_POSIX_FILE_DESCRIPTOR_SUPPORTED: 1,
pycuda._driver.device_attribute.HANDLE_TYPE_WIN32_HANDLE_SUPPORTED: 0,
pycuda._driver.device_attribute.HANDLE_TYPE_WIN32_KMT_HANDLE_SUPPORTED: 0,
pycuda._driver.device_attribute.HOST_NATIVE_ATOMIC_SUPPORTED: 0,
```

```
pycuda._driver.device_attribute.INTEGRATED: 0,
pycuda._driver.device_attribute.KERNEL_EXEC_TIMEOUT: 0,
pycuda._driver.device_attribute.L2_CACHE_SIZE: 4194304,
pycuda._driver.device_attribute.LOCAL_L1_CACHE_SUPPORTED: 1,
pycuda._driver.device_attribute.MANAGED_MEMORY: 1,
pycuda._driver.device_attribute.MAXIMUM_SURFACE1D_LAYERED_LAYERS: 2048,
pycuda._driver.device_attribute.MAXIMUM_SURFACE1D_LAYERED_WIDTH: 32768,
pycuda._driver.device_attribute.MAXIMUM_SURFACE1D_WIDTH: 32768,
pycuda._driver.device_attribute.MAXIMUM_SURFACE2D_HEIGHT: 65536,
pycuda._driver.device_attribute.MAXIMUM_SURFACE2D_LAYERED_HEIGHT: 32768,
pycuda._driver.device_attribute.MAXIMUM_SURFACE2D_LAYERED_LAYERS: 2048,
pycuda._driver.device_attribute.MAXIMUM_SURFACE2D_LAYERED_WIDTH: 32768,
pycuda._driver.device_attribute.MAXIMUM_SURFACE2D_WIDTH: 131072,
pycuda._driver.device_attribute.MAXIMUM_SURFACE3D_DEPTH: 16384,
pycuda._driver.device_attribute.MAXIMUM_SURFACE3D_HEIGHT: 16384,
pycuda._driver.device_attribute.MAXIMUM_SURFACE3D_WIDTH: 16384,
pycuda._driver.device_attribute.MAXIMUM_SURFACECUBEMAP_LAYERED_LAYERS: 2046,
pycuda._driver.device_attribute.MAXIMUM_SURFACECUBEMAP_LAYERED_WIDTH: 32768,
pycuda._driver.device_attribute.MAXIMUM_SURFACECUBEMAP_WIDTH: 32768,
pycuda._driver.device_attribute.MAXIMUM_TEXTURE1D_LAYERED_LAYERS: 2048,
pycuda._driver.device_attribute.MAXIMUM_TEXTURE1D_LAYERED_WIDTH: 32768,
pycuda._driver.device_attribute.MAXIMUM_TEXTURE1D_LINEAR_WIDTH: 268435456,
pycuda._driver.device_attribute.MAXIMUM_TEXTURE1D_MIPMAPPED_WIDTH: 32768,
pycuda._driver.device_attribute.MAXIMUM_TEXTURE1D_WIDTH: 131072,
pycuda._driver.device_attribute.MAXIMUM_TEXTURE2D_ARRAY_HEIGHT: 32768,
pycuda._driver.device_attribute.MAXIMUM_TEXTURE2D_ARRAY_NUMSLICES: 2048,
pycuda._driver.device_attribute.MAXIMUM_TEXTURE2D_ARRAY_WIDTH: 32768,
pycuda._driver.device_attribute.MAXIMUM_TEXTURE2D_GATHER_HEIGHT: 32768,
pycuda._driver.device_attribute.MAXIMUM_TEXTURE2D_GATHER_WIDTH: 32768,
pycuda._driver.device_attribute.MAXIMUM_TEXTURE2D_HEIGHT: 65536,
pycuda._driver.device_attribute.MAXIMUM_TEXTURE2D_LINEAR_HEIGHT: 65000,
pycuda._driver.device_attribute.MAXIMUM_TEXTURE2D_LINEAR_PITCH: 2097120,
pycuda._driver.device_attribute.MAXIMUM_TEXTURE2D_LINEAR_WIDTH: 131072,
pycuda._driver.device_attribute.MAXIMUM_TEXTURE2D_MIPMAPPED_HEIGHT: 32768,
pycuda._driver.device_attribute.MAXIMUM_TEXTURE2D_MIPMAPPED_WIDTH: 32768,
pycuda._driver.device_attribute.MAXIMUM_TEXTURE2D_WIDTH: 131072,
pycuda._driver.device_attribute.MAXIMUM_TEXTURE3D_DEPTH: 16384,
pycuda._driver.device_attribute.MAXIMUM_TEXTURE3D_DEPTH_ALTERNATE: 32768,
pycuda._driver.device_attribute.MAXIMUM_TEXTURE3D_HEIGHT: 16384,
pycuda._driver.device_attribute.MAXIMUM_TEXTURE3D_HEIGHT_ALTERNATE: 8192,
pycuda._driver.device_attribute.MAXIMUM_TEXTURE3D_WIDTH: 16384,
pycuda._driver.device_attribute.MAXIMUM_TEXTURE3D_WIDTH_ALTERNATE: 8192,
pycuda._driver.device_attribute.MAXIMUM_TEXTURECUBEMAP_LAYERED_LAYERS: 2046,
pycuda._driver.device_attribute.MAXIMUM_TEXTURECUBEMAP_LAYERED_WIDTH: 32768,
pycuda._driver.device_attribute.MAXIMUM_TEXTURECUBEMAP_WIDTH: 32768,
pycuda._driver.device_attribute.MAX_BLOCKS_PER_MULTIPROCESSOR: 16,
pycuda._driver.device_attribute.MAX_BLOCK_DIM_X: 1024,
pycuda._driver.device_attribute.MAX_BLOCK_DIM_Y: 1024,
```

```
pycuda._driver.device_attribute.MAX_BLOCK_DIM_Z: 64,
pycuda._driver.device_attribute.MAX_GRID_DIM_X: 2147483647,
pycuda._driver.device_attribute.MAX_GRID_DIM_Y: 65535,
pycuda._driver.device_attribute.MAX_GRID_DIM_Z: 65535,
pycuda._driver.device_attribute.MAX_PERSISTING_L2_CACHE_SIZE: 0,
pycuda._driver.device_attribute.MAX_PITCH: 2147483647,
pycuda._driver.device_attribute.MAX_REGISTERS_PER_BLOCK: 65536,
pycuda._driver.device_attribute.MAX_REGISTERS_PER_MULTIPROCESSOR: 65536,
pycuda._driver.device_attribute.MAX_SHARED_MEMORY_PER_BLOCK: 49152,
pycuda._driver.device_attribute.MAX_SHARED_MEMORY_PER_BLOCK_OPTIN: 65536,
pycuda._driver.device_attribute.MAX_SHARED_MEMORY_PER_MULTIPROCESSOR: 65536,
pycuda._driver.device_attribute.MAX_THREADS_PER_BLOCK: 1024,
pycuda._driver.device_attribute.MAX_THREADS_PER_MULTIPROCESSOR: 1024,
pycuda._driver.device_attribute.MEMORY_CLOCK_RATE: 5001000,
pycuda._driver.device_attribute.MEMORY_POOLS_SUPPORTED: 1,
pycuda._driver.device_attribute.MULTIPROCESSOR_COUNT: 40,
pycuda._driver.device_attribute.MULTI_GPU_BOARD: 0,
pycuda._driver.device_attribute.MULTI_GPU_BOARD_GROUP_ID: 0,
pycuda._driver.device_attribute.PAGEABLE_MEMORY_ACCESS: 0,
pycuda._driver.device_attribute.PAGEABLE_MEMORY_ACCESS_USES_HOST_PAGE_TABLES: 0,
pycuda._driver.device_attribute.PCI_BUS_ID: 0,
pycuda._driver.device_attribute.PCI_DEVICE_ID: 4,
pycuda._driver.device_attribute.PCI_DOMAIN_ID: 0,
pycuda._driver.device_attribute.READ_ONLY_HOST_REGISTER_SUPPORTED: 1,
pycuda._driver.device_attribute.RESERVED_SHARED_MEMORY_PER_BLOCK: 0,
pycuda._driver.device_attribute.SINGLE_TO_DOUBLE_PRECISION_PERF_RATIO: 32,
pycuda._driver.device_attribute.STREAM_PRIORITIES_SUPPORTED: 1,
pycuda._driver.device_attribute.SURFACE_ALIGNMENT: 512,
pycuda._driver.device_attribute.TCC_DRIVER: 0,
pycuda._driver.device_attribute.TEXTURE_ALIGNMENT: 512,
pycuda._driver.device_attribute.TEXTURE_PITCH_ALIGNMENT: 32,
pycuda._driver.device_attribute.TOTAL_CONSTANT_MEMORY: 65536,
pycuda._driver.device_attribute.UNIFIED_ADDRESSING: 1,
pycuda._driver.device_attribute.WARP_SIZE: 32}
```

oppure anche con il metodo DeviceData()

[22]:
```
from pycuda import autoinit
from pycuda.tools import DeviceData
specs = DeviceData()
print ('Max threads per block = '+str(specs.max_threads))
print ('Warp size            ='+str(specs.warp_size))
print ('Warps per MP         ='+str(specs.warps_per_mp))
print ('Thread Blocks per MP ='+str(specs.thread_blocks_per_mp))
print ('Registers            ='+str(specs.registers))
print ('Shared memory        ='+str(specs.shared_memory))
```

```
Max threads per block = 1024
```

```
Warp size            =32
Warps per MP         =64
Thread Blocks per MP =8
Registers            =65536
Shared memory        =49152
```

4 Esempio GPU in C

(comunque ci servirà dopo) Proviamo a scrivere e compulare un programma GPU in C. Notare il comando (magic) all'inizio che serve per salvare nel workspace il contenuto della cella in un file

[23]:
```
%%writefile VecAdd.cu
# include <stdio.h>
# include <cuda_runtime.h>
// CUDA Kernel
__global__ void vectorAdd(const float *A, const float *B, float *C, int␣
 ↪numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < numElements)
    {
        C[i] = A[i] + B[i];
    }
}

/**
 * Host main routine
 */
int main(void)
{
    int numElements = 15;
    size_t size = numElements * sizeof(float);
    printf("[Vector addition of %d elements]\n", numElements);

    float a[numElements],b[numElements],c[numElements];
    float *a_gpu,*b_gpu,*c_gpu;

    cudaMalloc((void **)&a_gpu, size);
    cudaMalloc((void **)&b_gpu, size);
    cudaMalloc((void **)&c_gpu, size);

    for (int i=0;i<numElements;++i ){

        a[i] = i*i;
        b[i] = i;
```

```
    }
    // Copy the host input vectors A and B in host memory to the device input␣
↪vectors in
    // device memory
    printf("Copy input data from the host memory to the CUDA device\n");
    cudaMemcpy(a_gpu, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(b_gpu, b, size, cudaMemcpyHostToDevice);

    // Launch the Vector Add CUDA Kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =(numElements + threadsPerBlock - 1) / threadsPerBlock;
    printf("CUDA kernel launch with %d blocks of %d threads\n", blocksPerGrid,␣
↪threadsPerBlock);
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(a_gpu, b_gpu, c_gpu,␣
↪numElements);

    // Copy the device result vector in device memory to the host result vector
    // in host memory.
    printf("Copy output data from the CUDA device to the host memory\n");
    cudaMemcpy(c, c_gpu, size, cudaMemcpyDeviceToHost);

    for (int i=0;i<numElements;++i ){
        printf("%f \n",c[i]);
    }

    // Free device global memory
    cudaFree(a_gpu);
    cudaFree(b_gpu);
    cudaFree(c_gpu);

    printf("Done\n");
    return 0;
}
```

```
Writing VecAdd.cu
```

[24]:
```
ls
```

```
drive/                    handson_gpu_2022.ipynb
test_dir/
handson_gpu_2022_files/   sample_data/
VecAdd.cu
```

[26]:
```
!nvcc -o VecAdd VecAdd.cu -arch=compute_70 -code=sm_70
```

[27]:
```
!./VecAdd
```

```
[Vector addition of 15 elements]
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 1 blocks of 256 threads
Copy output data from the CUDA device to the host memory
0.000000
2.000000
6.000000
12.000000
20.000000
30.000000
42.000000
56.000000
72.000000
90.000000
110.000000
132.000000
156.000000
182.000000
210.000000
Done
```

5 Implementazione con pycuda

Facciamo un primo esempio con pycuda

importiamo i moduli ch eci servono

[28]:
```
from pycuda import autoinit
from pycuda import gpuarray
import numpy as np
```

definiamo i vettori a, b e c sull'host. Tutti di lunghezza 15, a con i numeri da 0..14 e b con i quadrati. c è inizializzato a 0

[29]:
```
aux = range(15)
a = np.array(aux).astype(np.float32)
b = (a*a).astype(np.float32)
c = np.zeros(len(aux)).astype(np.float32)
```

Definiamo i vettori sulla GPU e copiamo dentro il contenuto dei vettori a,b e c definiti sull'host

[30]:
```
a_gpu = gpuarray.to_gpu(a)
b_gpu = gpuarray.to_gpu(b)
c_gpu = gpuarray.to_gpu(c)
```

un primo modo semplice per sommare i vettori e semplicemente usare il +

[31]:
```
c_gpu=a_gpu+b_gpu
```

stampiamo i risultati

[32]:
```
print(c_gpu)
```

```
[  0.   2.   6.  12.  20.  30.  42.  56.  72.  90. 110. 132. 156. 182.
 210.]
```

[33]:
```
c_gpu
```

[33]:
```
array([  0.,   2.,   6.,  12.,  20.,  30.,  42.,  56.,  72.,  90., 110.,
       132., 156., 182., 210.], dtype=float32)
```

Un secondo modo è quello di utilizzre il metodo elementwise, che applica la stessa "Operation" a tutti gli elementi dei vettori

[34]:
```
from pycuda.elementwise import ElementwiseKernel
myCudaFunc = ElementwiseKernel(arguments = "float *a, float *b, float *c",
                               operation = "c[i] = a[i]+b[i]",
                               name = "mySumK")
```

[35]:
```
myCudaFunc(a_gpu,b_gpu,c_gpu)
```

[36]:
```
c_gpu
```

[36]:
```
array([  0.,   2.,   6.,  12.,  20.,  30.,  42.,  56.,  72.,  90., 110.,
       132., 156., 182., 210.], dtype=float32)
```

Il vantaggio è che si possono definire anche operazioni piu' complesse della semplice somma, ad esempio

[37]:
```
from pycuda.elementwise import ElementwiseKernel
lin_comb = ElementwiseKernel(
        "float a, float *x, float b, float *y, float *z",
        "z[i] = a*x[i] + b*y[i]",
        "linear_combination")
```

in ogni caso l'operazione che vogliamo fare deve stare su una sola riga, quindi non può essere toppo complicata

[38]:
```
lin_comb(3.,a_gpu,5.,b_gpu,c_gpu)
```

[39]:
```
c_gpu
```

[39]:
```
array([   0.,    8.,   26.,   54.,   92.,  140.,  198.,  266.,  344.,
        432.,  530.,  638.,  756.,  884., 1022.], dtype=float32)
```

Il terzo metodo è il piu' "generico". SI utilizza il mtodo SourceModule che permette di definire anche kernel piu' complessi. L'idea è che questi kernel siano comunque scritti in Cuda/C

[40]:
```
from pycuda.compiler import SourceModule
```

carichiamo il file contenente il codice in c che avevamo scritto prima (fare !ls se avete dubbi sul nome che gli avete dato)

[41]:
```
!ls
```

```
drive                   handson_gpu_2022.ipynb  test_dir  VecAdd.cu
handson_gpu_2022_files  sample_data             VecAdd
```

[42]:
```
cudaCode = open("VecAdd.cu","r") #apro il file
myCUDACode = cudaCode.read() #copio il testo del file nella lista myCUDACode
```

dentro la lista myCUDACode ci sta il nostro programma:

[43]:
```
myCUDACode
```

[43]:
```
'# include <stdio.h>\n# include <cuda_runtime.h>\n// CUDA Kernel\n__global__
void vectorAdd(const float *A, const float *B, float *C, int numElements)\n{\n
int i = blockDim.x * blockIdx.x + threadIdx.x;\n    if (i < numElements)\n
{\n        C[i] = A[i] + B[i];\n    }\n}\n \n/**\n * Host main routine\n */\nint
main(void)\n{\n    int numElements = 15;\n    size_t size = numElements *
sizeof(float);\n    printf("[Vector addition of %d elements]\\n",
numElements);\n \n    float a[numElements],b[numElements],c[numElements];\n
float *a_gpu,*b_gpu,*c_gpu;\n \n    cudaMalloc((void **)&a_gpu, size);\n
cudaMalloc((void **)&b_gpu, size);\n    cudaMalloc((void **)&c_gpu, size);\n \n
for (int i=0;i<numElements;++i ){\n \n        a[i] = i*i;\n        b[i] = i;\n
\n    }\n    // Copy the host input vectors A and B in host memory to the device
input vectors in\n    // device memory\n    printf("Copy input data from the
host memory to the CUDA device\\n");\n    cudaMemcpy(a_gpu, a, size,
cudaMemcpyHostToDevice);\n    cudaMemcpy(b_gpu, b, size,
cudaMemcpyHostToDevice);\n \n    // Launch the Vector Add CUDA Kernel\n    int
threadsPerBlock = 256;\n    int blocksPerGrid =(numElements + threadsPerBlock -
1) / threadsPerBlock;\n    printf("CUDA kernel launch with %d blocks of %d
threads\\n", blocksPerGrid, threadsPerBlock);\n    vectorAdd<<<blocksPerGrid,
threadsPerBlock>>>(a_gpu, b_gpu, c_gpu, numElements);\n \n    // Copy the device
result vector in device memory to the host result vector\n    // in host
memory.\n    printf("Copy output data from the CUDA device to the host
memory\\n");\n    cudaMemcpy(c, c_gpu, size, cudaMemcpyDeviceToHost);\n \n
for (int i=0;i<numElements;++i ){\n        printf("%f \\n",c[i]);\n    }\n \n
// Free device global memory\n    cudaFree(a_gpu);\n    cudaFree(b_gpu);\n
cudaFree(c_gpu);\n \n    printf("Done\\n");\n    return 0;\n}\n'
```

compiliamo il codice just-in-time con il metodo SourceModule()

SourceModule() chiama il coompilatore e genera dentro myCode una funzione per avere a disposizione il kernel.

[44]:
```
myCode = SourceModule(myCUDACode)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: UserWarning: The
CUDA compiler succeeded, but said the following:
kernel.cu(17): warning: function "main" cannot be declared in a linkage-
specification


  """Entry point for launching an IPython kernel.
```

ora il kernel (e l'host) è compilato. Importiamolo nel programma in python

[45]:
```
importedKernel = myCode.get_function("vectorAdd")
```

Fino ad ora la "geometria" della GPU l'ha gestita automaticamente pyCuda. Ora siamo noi a definire la "geometria" della GPU che vogliamo usare:

[46]:
```
nThreadsPerBlock = 256
nBlockPerGrid = 1
nGridsPerBlock = 1
```

resettiamo il vettore c_gpu (per essere sicuri sia vuoto)

[47]:
```
c_gpu.set(c)
c_gpu
```

[47]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
      dtype=float32)
```

Il puntatore nella memoria gpu è dato dall'attributo gpudata

[48]:
```
a_gpu.gpudata
```

[48]:
```
<pycuda._driver.DeviceAllocation at 0x7fcc161cc080>
```

[49]:
```
b_gpu.gpudata
```

[49]:
```
<pycuda._driver.DeviceAllocation at 0x7fcc161cc2b0>
```

lanciamo il kernel importato passandogli i **puntatori** dei vettori e la geometria della GPU

[50]:
```
importedKernel(a_gpu.gpudata, b_gpu.gpudata, c_gpu.gpudata,␣
↪block=(nThreadsPerBlock,nBlockPerGrid,nGridsPerBlock))
```

[51]:
```
c_gpu
```

[51]:
```
array([  0.,   2.,   6.,  12.,  20.,  30.,  42.,  56.,  72.,  90., 110.,
       132., 156., 182., 210.], dtype=float32)
```

ovviamente questo ultimo metodo è eccessivo se ho

6 Somma di Matrici

Puliamo la memoria

[52]:
```
%reset
```

```
Once deleted, variables cannot be recovered. Proceed (y/[n])? y
```

importiamo le cose che ci servono

[53]:
```
import numpy as np
from pycuda import gpuarray, autoinit
import pycuda.driver as cuda
from pycuda.tools import DeviceData
from pycuda.tools import OccupancyRecord as occupancy
```

inizializziamo gli array con le dimensioni appropriate

[54]:
```
presCPU, presGPU = np.float32, 'float'
#presCPU, presGPU = np.float64, 'double'
a_cpu = np.random.random((512,512)).astype(presCPU)
b_cpu = np.random.random((512,512)).astype(presCPU)
c_cpu = np.zeros((512,512), dtype=presCPU)
```

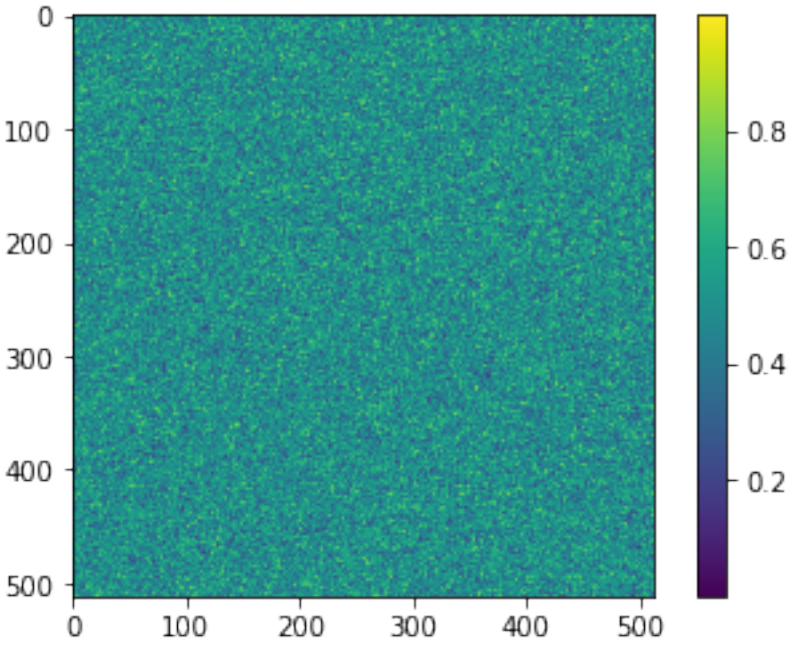
carichiamo matplotlib per poterlo usare nella Ipython

[55]:
```
%matplotlib inline
```

[56]:
```
from matplotlib import pyplot as plt
```

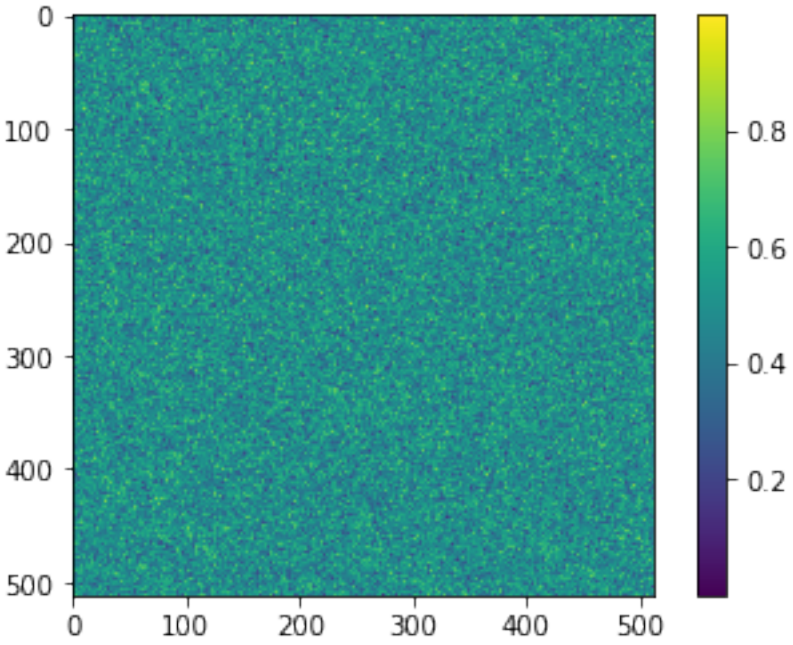
[57]:
```
plt.imshow(a_cpu)
plt.colorbar()
```

[57]:
```
<matplotlib.colorbar.Colorbar at 0x7fcc160380d0>
```



[58]:
```
plt.imshow(b_cpu)
plt.colorbar()
```

[58]: <matplotlib.colorbar.Colorbar at 0x7fcc100b6750>



copiamo gli array sulla gpu

[59]:
```
a_gpu = gpuarray.to_gpu(a_cpu)
b_gpu = gpuarray.to_gpu(b_cpu)
c_gpu = gpuarray.to_gpu(c_cpu)
```

[60]:
```
c_gpu
```

[60]:
```
array([[0., 0., 0., …, 0., 0., 0.],
       [0., 0., 0., …, 0., 0., 0.],
       [0., 0., 0., …, 0., 0., 0.],
       …,
       [0., 0., 0., …, 0., 0., 0.],
       [0., 0., 0., …, 0., 0., 0.],
       [0., 0., 0., …, 0., 0., 0.]], dtype=float32)
```

facciamo la somma prima sull'host

[61]:
```
c_cpu=a_cpu+b_cpu
```

[62]:
```
c_cpu
```

[62]:
```
array([[1.2318479 , 0.7115891 , 1.0627784 , …, 1.0367434 , 1.2572979 ,
        1.435501  ],
       [1.1625738 , 0.69545865, 1.1233734 , …, 1.7289808 , 0.98316765,
        1.472131  ],
       [0.31298584, 0.7491113 , 0.5359408 , …, 0.9131348 , 0.6572113 ,
        0.75546235],
       …,
       [1.0218427 , 1.3309641 , 0.73231196, …, 1.7113471 , 1.4630361 ,
        0.8734757 ],
       [1.1179438 , 1.7768974 , 1.6918807 , …, 1.1337817 , 0.73312706,
        0.67441964],
       [0.6375501 , 0.8411865 , 0.85855544, …, 0.9857786 , 0.13109833,
        0.6627484 ]], dtype=float32)
```

misuriamo il tempo che ci vuole sull'host per fare la somma

[63]:
```
t_cpu = %timeit -o c_cpu = a_cpu+b_cpu
```

```
172 µs ± 9.2 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

definiamo il kernel gpu per fare la somma

[64]:
```
cudaKernel = '''
__global__ void matrixAdd(float *A, float *B, float *C)
{
    int tid_x = blockDim.x * blockIdx.x + threadIdx.x;
    int tid_y = blockDim.y * blockIdx.y + threadIdx.y;
```

```
    int tid   = gridDim.x * blockDim.x * tid_y + tid_x;
    C[tid] = A[tid] + B[tid];
}
'''
```

ora dobbiamo compilare questo kernel e generare la funzione da usare in python

[65]:
```
from pycuda.compiler import SourceModule
myCode = SourceModule(cudaKernel)
```

[66]:
```
addMatrix = myCode.get_function("matrixAdd") # The output of get_function is␣
↪the GPU-compiled function.
```

[67]:
```
type(addMatrix)
```

[67]:
```
pycuda._driver.Function
```

dobbiamo decidere la geoemtria della GPU. Ad esempio si possono cercare di sfruttare tutt i threads a disposizione in un blocco. Quati thread ci sono in un blocco?

[68]:
```
dev = cuda.Device(0)
devdata = DeviceData(dev)
print ("Using device : "+dev.name() )
print("Max threads per block: "+str(dev.max_threads_per_multiprocessor))
```

```
Using device : Tesla T4
Max threads per block: 1024
```

Quindi possiamo usare blocchi 32x32. Le nostre matrici sono 512x512, per cui dobbiamo usare 16x16 blocchi

[69]:
```
cuBlock = (32,32,1)
cuGrid = (16,16,1)
```

abbiamo già compilato il kernel con SourceModule. Ora abbiamo due modi per lanciarlo. O chiamiamo direttamente la funzione (come abbiamo fatto sopra per la somam di vettori)

```
kernelFunction(arg1,arg2, ... ,block=(n,m,l),grid=(r,s,t)
```

oppure usiamo la ”preparation”

```
kernelFunction.prepare('ABC..') # Each letter corresponds to an input data type of the functio
kernelFunction.prepared_call(grid,block,arg1.gpudata,arg2,...) # When using GPU arrays, they s
```

il primo metodo è, per noi

[70]:
```
addMatrix(a_gpu,b_gpu,c_gpu,block=cuBlock,grid=cuGrid)
```

con la preparation è possibile midurare il tempo di esecuzione

[71]:
```
addMatrix.prepare('PPP')
addMatrix.prepared_call(cuGrid,cuBlock,a_gpu.gpudata,b_gpu.gpudata,c_gpu.
 ↪gpudata)
```

[72]:
```
time2 = addMatrix.prepared_timed_call(cuGrid,cuBlock,a_gpu.gpudata,b_gpu.
 ↪gpudata,c_gpu.gpudata)
```

[73]:
```
time2()
```

[73]:
```
2.7008000761270522e-05
```

per controllare il risultato dobbiamo copiare il risultato dalla gpu alla cpu

[74]:
```
c = c_gpu.get()
```

controlliamo il risultato per cpu e gpu

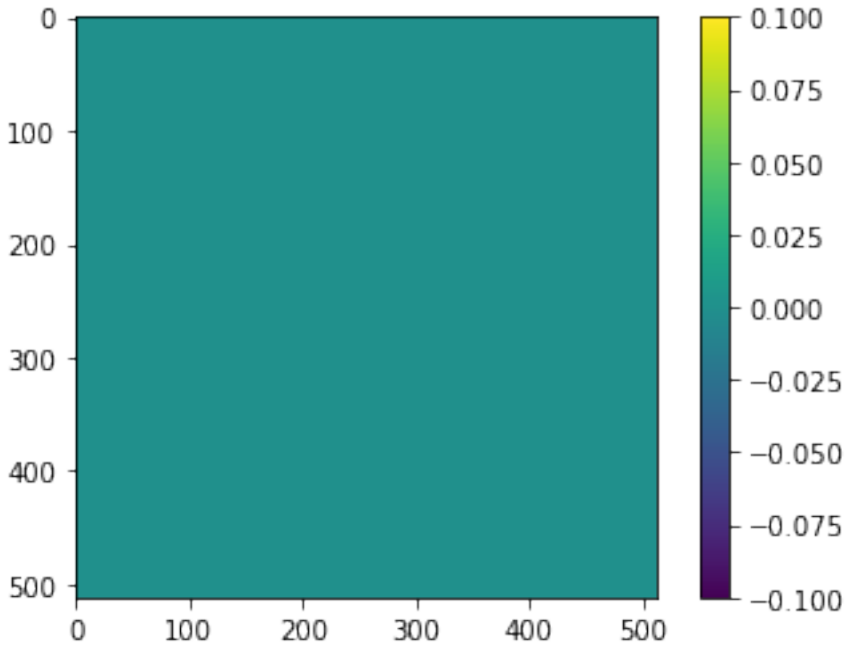
[75]:
```
c, c_cpu
```

[75]:
```
(array([[1.2318479 , 0.7115891 , 1.0627784 , …, 1.0367434 , 1.2572979 ,
         1.435501  ],
        [1.1625738 , 0.69545865, 1.1233734 , …, 1.7289808 , 0.98316765,
         1.472131  ],
        [0.31298584, 0.7491113 , 0.5359408 , …, 0.9131348 , 0.6572113 ,
         0.75546235],
        …,
        [1.0218427 , 1.3309641 , 0.73231196, …, 1.7113471 , 1.4630361 ,
         0.8734757 ],
        [1.1179438 , 1.7768974 , 1.6918807 , …, 1.1337817 , 0.73312706,
         0.67441964],
        [0.6375501 , 0.8411865 , 0.85855544, …, 0.9857786 , 0.13109833,
         0.6627484 ]], dtype=float32),
 array([[1.2318479 , 0.7115891 , 1.0627784 , …, 1.0367434 , 1.2572979 ,
         1.435501  ],
        [1.1625738 , 0.69545865, 1.1233734 , …, 1.7289808 , 0.98316765,
         1.472131  ],
        [0.31298584, 0.7491113 , 0.5359408 , …, 0.9131348 , 0.6572113 ,
         0.75546235],
        …,
        [1.0218427 , 1.3309641 , 0.73231196, …, 1.7113471 , 1.4630361 ,
         0.8734757 ],
        [1.1179438 , 1.7768974 , 1.6918807 , …, 1.1337817 , 0.73312706,
         0.67441964],
        [0.6375501 , 0.8411865 , 0.85855544, …, 0.9857786 , 0.13109833,
         0.6627484 ]], dtype=float32))
```

per confrontare meglio, guardiamo i plot

[76]:
```
plt.imshow(c-c_cpu,interpolation='none')
plt.colorbar()
```

[76]:
```
<matplotlib.colorbar.Colorbar at 0x7fcbfbfc38d0>
```



[77]:
```
np.sum(np.sum(np.abs(c_cpu-c)))
```

[77]:
```
0.0
```

in effetti i risultati sono uguali

7 Moltiplicazione tra matrici

scriviamo un kernel per la moltiplicazione di matrici

[78]:
```
cudaKernel2 = '''
__global__ void matrixMul(float *A, float *B, float *C)
{
    int tid_x = blockDim.x * blockIdx.x + threadIdx.x; // Row
    int tid_y = blockDim.y * blockIdx.y + threadIdx.y; // Column
    int matrixDim = gridDim.x * blockDim.x;
    int tid   = matrixDim * tid_y + tid_x; // element i,j

    float  aux=0.0f;
```

```
    for ( int i=0 ; i<matrixDim ; i++ ){
        //
        aux += A[matrixDim * tid_y + i]*B[matrixDim * i + tid_x] ;

    }

    C[tid] = aux;

}
'''
```

compiliamo e importiamo con SourceModule

[79]:
```
myCode = SourceModule(cudaKernel2)
mulMatrix = myCode.get_function("matrixMul")
```

eseguiamolo con la stessa struttura a blocchi definite per la somma di matrici

[80]:
```
mulMatrix(a_gpu,b_gpu,c_gpu,block=cuBlock,grid=cuGrid)
```

sulla CPU sarà invece

[81]:
```
dotAB = np.dot(a_cpu, b_cpu)
```

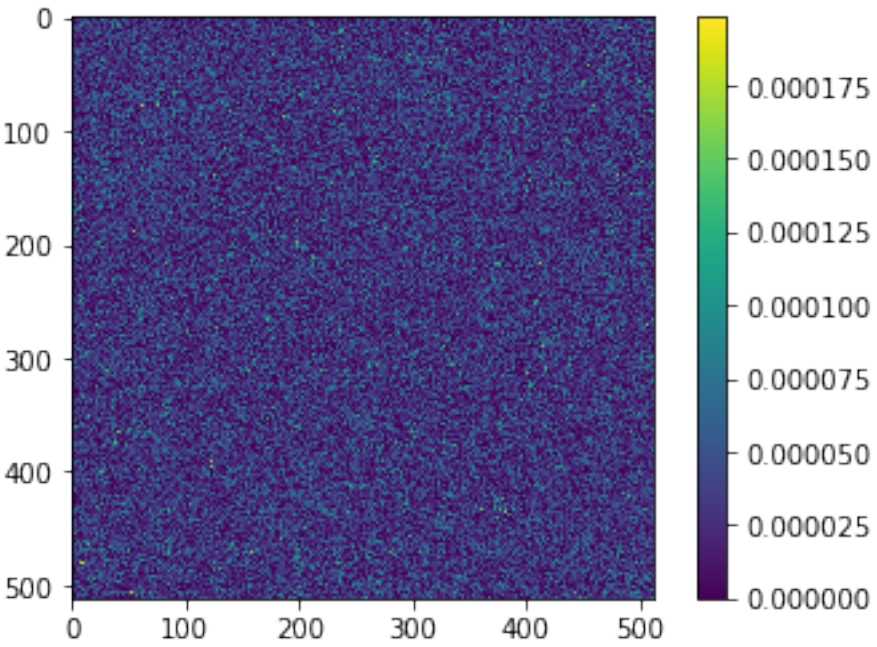
vediamo il risultato è lo stesso

[82]:
```
diff = np.abs(c_gpu.get()-dotAB)
np.sum(diff)
```

[82]:
```
8.635231
```

[83]:
```
plt.imshow(diff,interpolation='none')
plt.colorbar()
```

[83]:
```
<matplotlib.colorbar.Colorbar at 0x7fcbf970cb90>
```



[84]:
```
dotAB
```

[84]:
```
array([[121.42797 , 122.31373 , 120.10916 , …, 119.41626 , 121.96463 ,
        118.80441 ],
       [130.3919  , 129.17444 , 126.29321 , …, 130.36664 , 131.5235  ,
        128.30759 ],
       [124.68309 , 122.24606 , 120.65952 , …, 119.66283 , 123.355774,
        121.70793 ],
       …,
       [126.59974 , 126.97113 , 126.88341 , …, 124.75375 , 123.9772  ,
        127.03604 ],
       [129.1057  , 128.1775  , 128.33745 , …, 125.19934 , 126.26902 ,
        126.57745 ],
       [124.317215, 119.97131 , 122.27707 , …, 120.317505, 123.454956,
        122.56614 ]], dtype=float32)
```

[85]:
```
c_gpu
```

[85]:
```
array([[121.42807 , 122.31372 , 120.109215, …, 119.41626 , 121.9646  ,
        118.80444 ],
       [130.39189 , 129.17444 , 126.2932  , …, 130.36668 , 131.52351 ,
        128.30759 ],
       [124.68307 , 122.24605 , 120.659485, …, 119.662834, 123.35574 ,
        121.707886],
       …,
```

```
       [126.59972 , 126.97111 , 126.88342 , …, 124.753716, 123.9772  ,
        127.035995],
       [129.10559 , 128.17752 , 128.33742 , …, 125.19935 , 126.26911 ,
        126.57746 ],
       [124.317154, 119.971306, 122.2771  , …, 120.317474, 123.45495 ,
        122.56615 ]], dtype=float32)
```

[86]:
```
presCPU, presGPU = np.float64, 'double'
a_cpu = np.random.random((512,512)).astype(presCPU)
b_cpu = np.random.random((512,512)).astype(presCPU)
c_cpu = np.zeros((512,512), dtype=presCPU)
```

[87]:
```
a_gpu = gpuarray.to_gpu(a_cpu)
b_gpu = gpuarray.to_gpu(b_cpu)
c_gpu = gpuarray.to_gpu(c_cpu)
```

[88]:
```
a_cpu.dtype
```

[88]:
```
dtype('float64')
```

[89]:
```
cudaKernel3 = '''
__global__ void matrixMul64(double *A, double *B, double *C)
{
    int tid_x = blockDim.x * blockIdx.x + threadIdx.x; // Row
    int tid_y = blockDim.y * blockIdx.y + threadIdx.y; // Column
    int matrixDim = gridDim.x * blockDim.x;
    int tid   = matrixDim * tid_y + tid_x; // element i,j

    double aux = 0.0;
    for ( int i=0 ; i<matrixDim ; i++ ){
        //
        aux += A[matrixDim * tid_y + i]*B[matrixDim * i + tid_x] ;

    }

    C[tid] = aux;

}
'''
```

[90]:
```
myCode64 = SourceModule(cudaKernel3)
mulMatrix64 = myCode64.get_function("matrixMul64")
```

[91]:
```
mulMatrix64(a_gpu,b_gpu,c_gpu,block=cuBlock,grid=cuGrid)
```

[92]:
```
dotAB = np.dot(a_cpu, b_cpu)
```

[93]:
```
c_gpu.dtype
```

[93]:
```
dtype('float64')
```

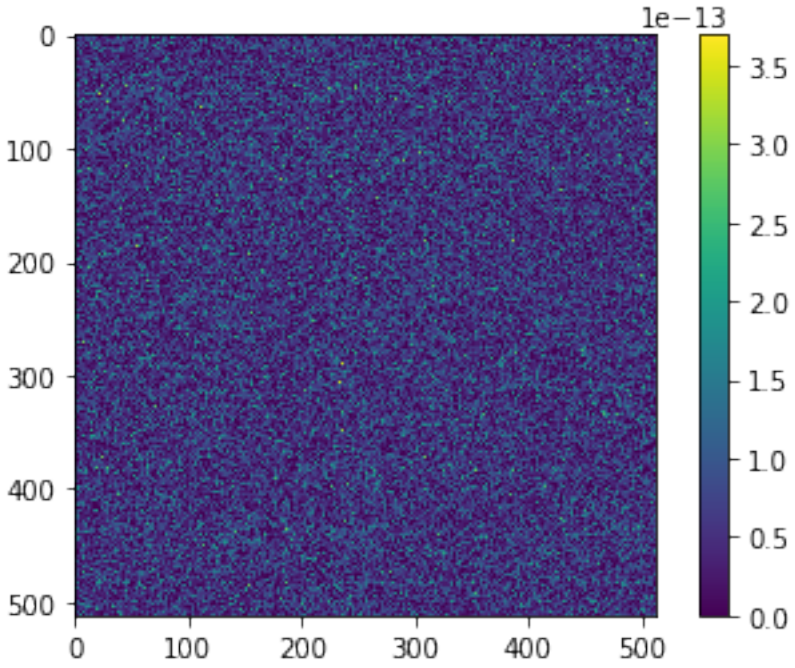
[94]:
```
dotAB.dtype
```

[94]:
```
dtype('float64')
```

[95]:
```
diff = np.abs(c_gpu.get()-dotAB)
```

[96]:
```
plt.imshow(diff,interpolation='none')
plt.colorbar()
```

[96]:
```
<matplotlib.colorbar.Colorbar at 0x7fcbf966b510>
```



[96]:
```

```

8 Ancora sulla somma di vettori

[97]:
```
%reset
```

```
Once deleted, variables cannot be recovered. Proceed (y/[n])? y
```

Vogliamo confrontare i tempi per la somma di vettori di dimensione variabile, tra CPU e GPU

Iniziamo con la versione CPU

[98]:
```
%matplotlib inline
from matplotlib import pyplot as plt
```

[99]:
```
import numpy as np
```

[100]:
```
from time import time
def myColorRand():
    return (np.random.random(),np.random.random(),np.random.random())
```

[101]:
```
dimension = [2**i for i in range(5,25) ]
myPrec = np.float32
```

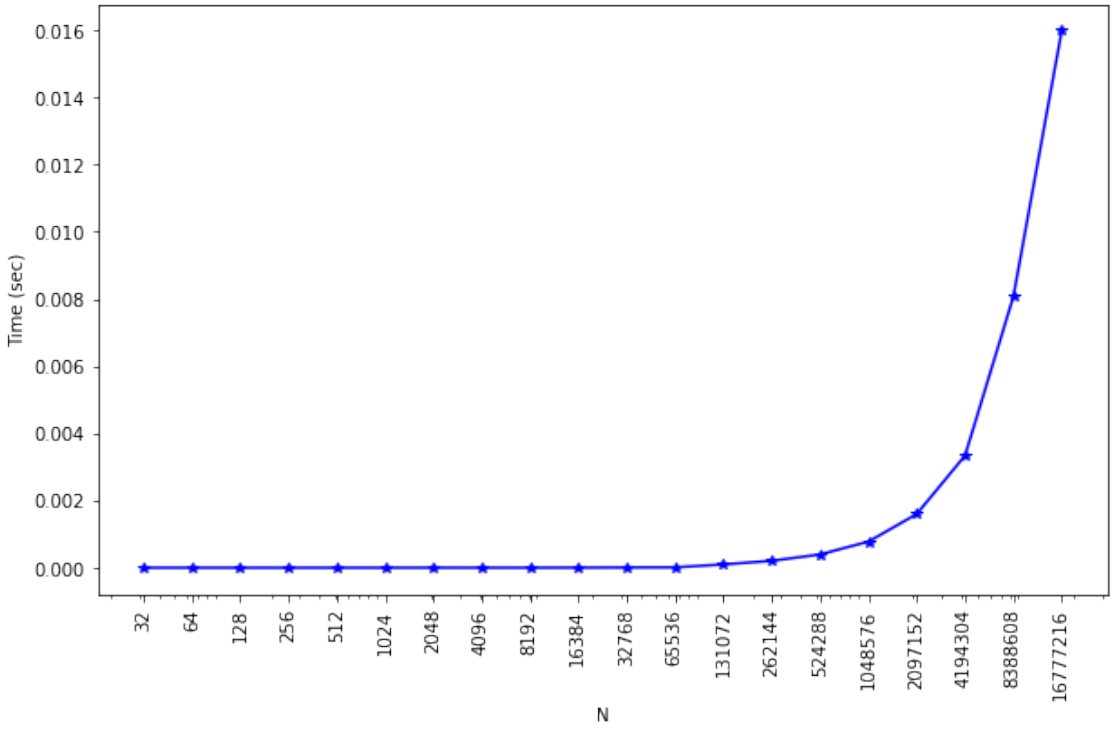
[102]:
```
dimension
```

[102]:
```
[32,
 64,
 128,
 256,
 512,
 1024,
 2048,
 4096,
 8192,
 16384,
 32768,
 65536,
 131072,
 262144,
 524288,
 1048576,
 2097152,
 4194304,
 8388608,
 16777216]
```

[103]:
```
nLoops = 100
timeCPU = []
for n in dimension:
    v1_cpu = np.random.random(n).astype(myPrec)
    v2_cpu = np.random.random(n).astype(myPrec)
    tMean = 0
    for i in range(nLoops):
        t = time()
        v = v1_cpu+v2_cpu
        t = time() - t
        tMean += t/nLoops
```

```
    timeCPU.append(tMean)
```

[104]:
```
plt.figure(1,figsize=(10,6))
plt.semilogx(dimension,timeCPU,'b-*')
plt.ylabel('Time (sec)')
plt.xlabel('N')
plt.xticks(dimension, dimension, rotation='vertical')
plt.show()
```



Proviamo a fare la versione GPU

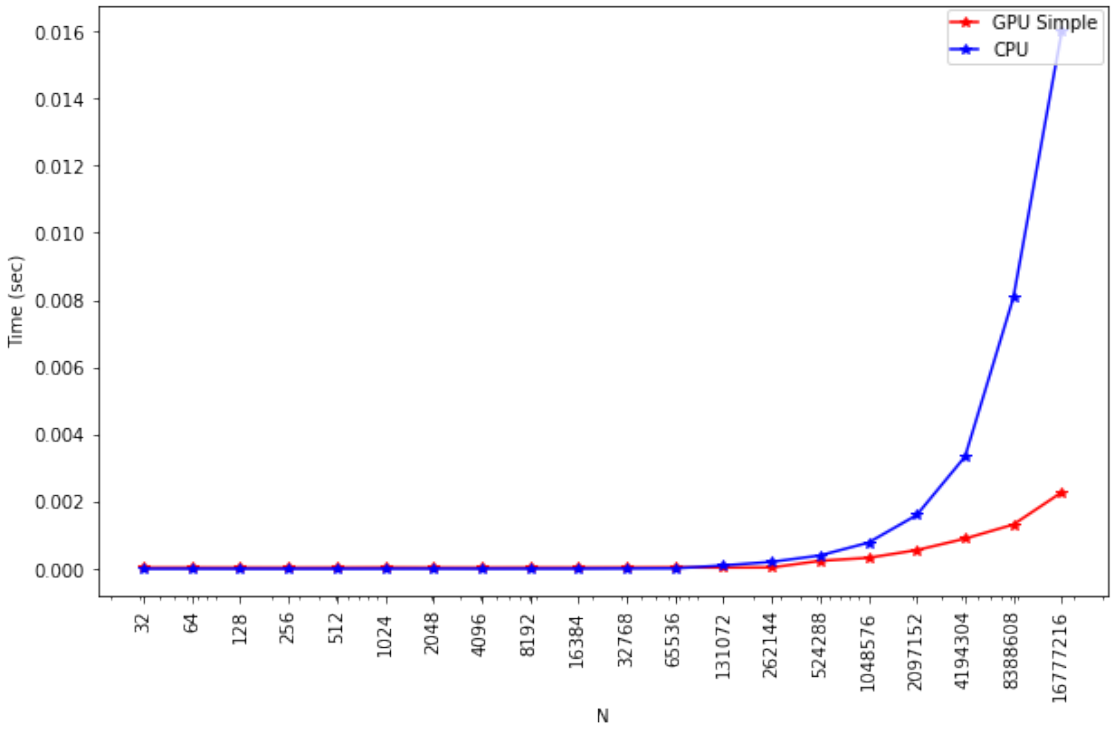
Per prima cosa guardiamo la semplice somma (primo metodo)

[105]:
```
import pycuda
from pycuda import gpuarray
```

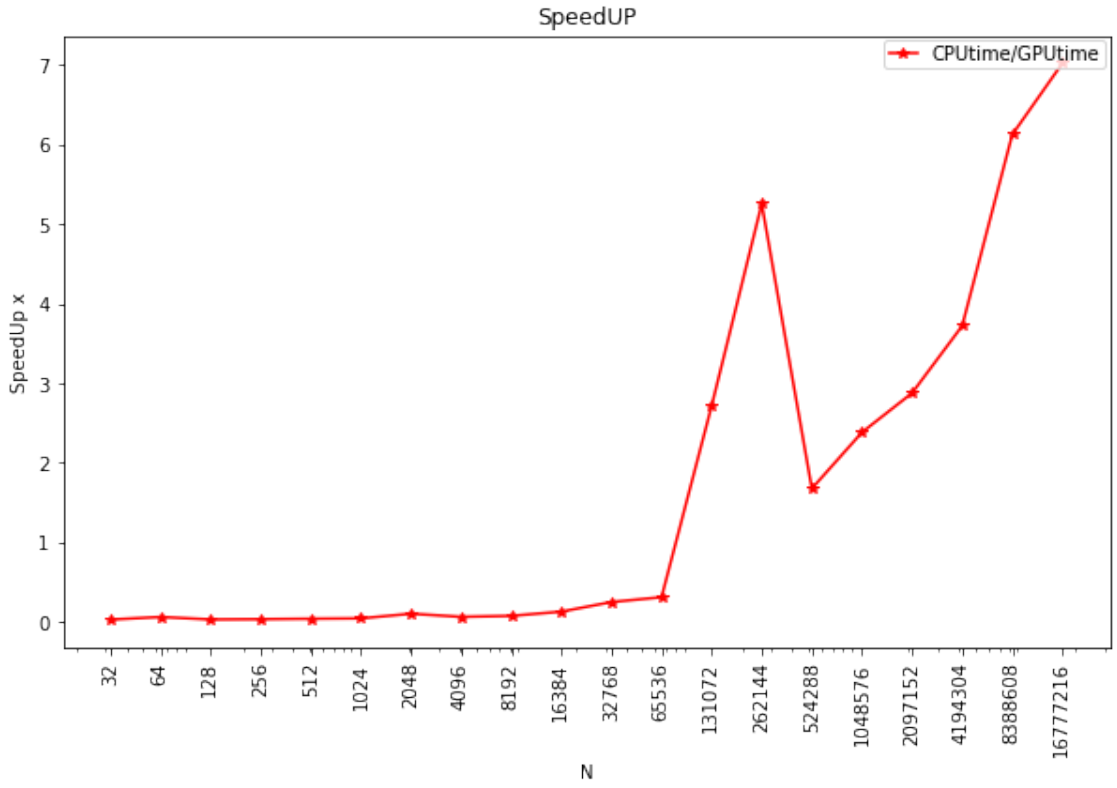
[106]:
```
timeGPU1 = []
bandWidth1 = []
for n in dimension:
    v1_cpu = np.random.random(n).astype(myPrec)
    v2_cpu = np.random.random(n).astype(myPrec)
    t1Mean = 0
    t2Mean = 0
    for i in range(nLoops):
```

```
        t = time()
        vaux = gpuarray.to_gpu(v1_cpu)
        t = time() -t
        t1Mean += t/nLoops
    bandWidth1.append(t1Mean)
    v1_gpu = gpuarray.to_gpu(v1_cpu)
    v2_gpu = gpuarray.to_gpu(v2_cpu)
    for i in range(nLoops):
        t = time()
        v = v1_gpu+v2_gpu
        t = time() -t
        t2Mean += t/nLoops
    timeGPU1.append(t2Mean)
    v1_gpu.gpudata.free()
    v2_gpu.gpudata.free()
    v.gpudata.free()
```

[107]:
```
plt.figure(1,figsize=(10,6))
plt.semilogx(dimension,timeGPU1,'r-*',label='GPU Simple')
plt.semilogx(dimension,timeCPU,'b-*',label='CPU')
plt.ylabel('Time (sec)')
plt.xlabel('N')
plt.xticks(dimension, dimension, rotation='vertical')
plt.legend(loc=1,labelspacing=0.5,fancybox=True, handlelength=1.5,␣
 ↪borderaxespad=0.25, borderpad=0.25)
plt.show()
```



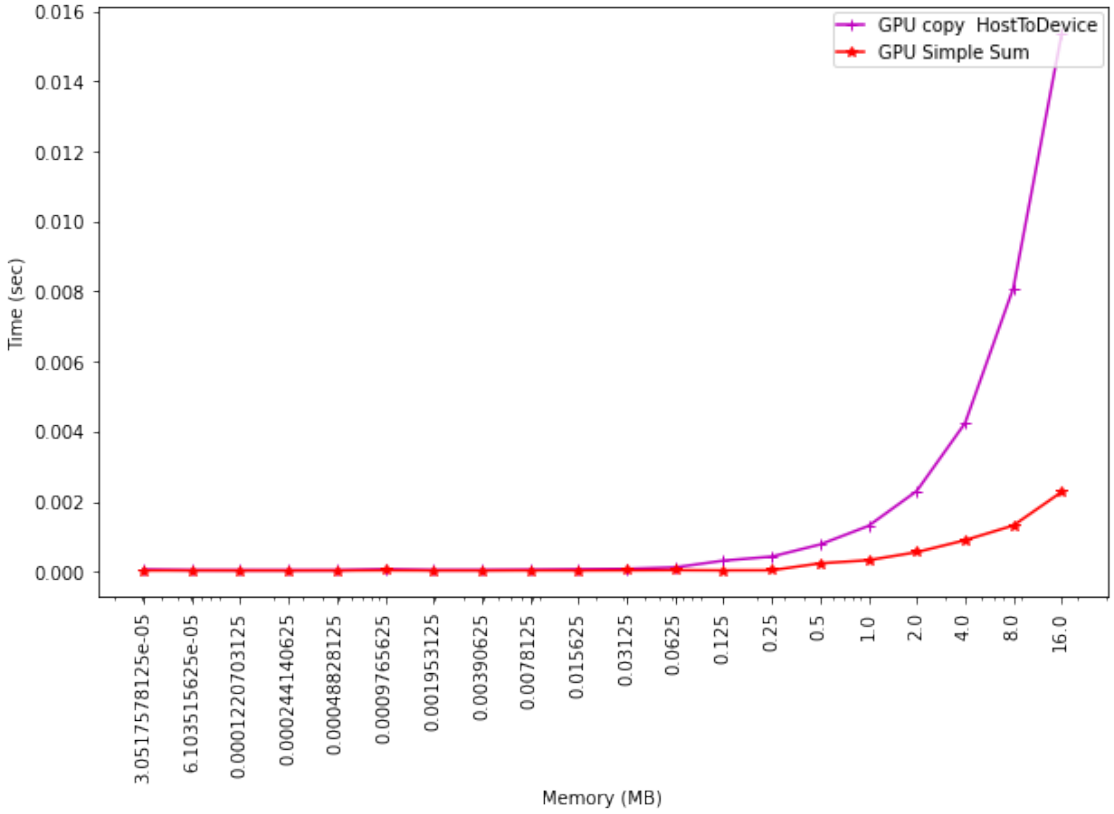
[108]:
```
plt.figure(1,figsize=(10,6))

a = np.array(timeGPU1)
b = np.array(timeCPU)
plt.semilogx(dimension,b/a,'r-*',label='CPUtime/GPUtime')
plt.ylabel('SpeedUp x')
plt.xlabel('N')
plt.title('SpeedUP')
plt.xticks(dimension, dimension, rotation='vertical')
plt.legend(loc=1,labelspacing=0.5,fancybox=True, handlelength=1.5,␣
 ↪borderaxespad=0.25, borderpad=0.25)
plt.show()
```



proviamo anche a valutare il tempo di trasferimento su GPU

[109]:
```
plt.figure(1,figsize=(10,6))
sizeMB = np.array(dimension)/(2.**20)
plt.semilogx(sizeMB,bandWidth1,'m-+',label='GPU copy  HostToDevice')
plt.semilogx(sizeMB,timeGPU1,'r-*',label='GPU Simple Sum')
plt.ylabel('Time (sec)')
```

```
plt.xlabel('Memory (MB)')
plt.xticks(sizeMB, sizeMB, rotation='vertical')
plt.legend(loc=1,labelspacing=0.5,fancybox=True, handlelength=1.5,␣
 ↪borderaxespad=0.25, borderpad=0.25)
plt.show()
```



proviamo ad usare elementwise (secondo metodo)

[110]:
```
from pycuda.elementwise import ElementwiseKernel
myCudaFunc = ElementwiseKernel(arguments = "float *a, float *b, float *c",
                               operation = "c[i] = a[i]+b[i]",
                               name = "mySumK")
```

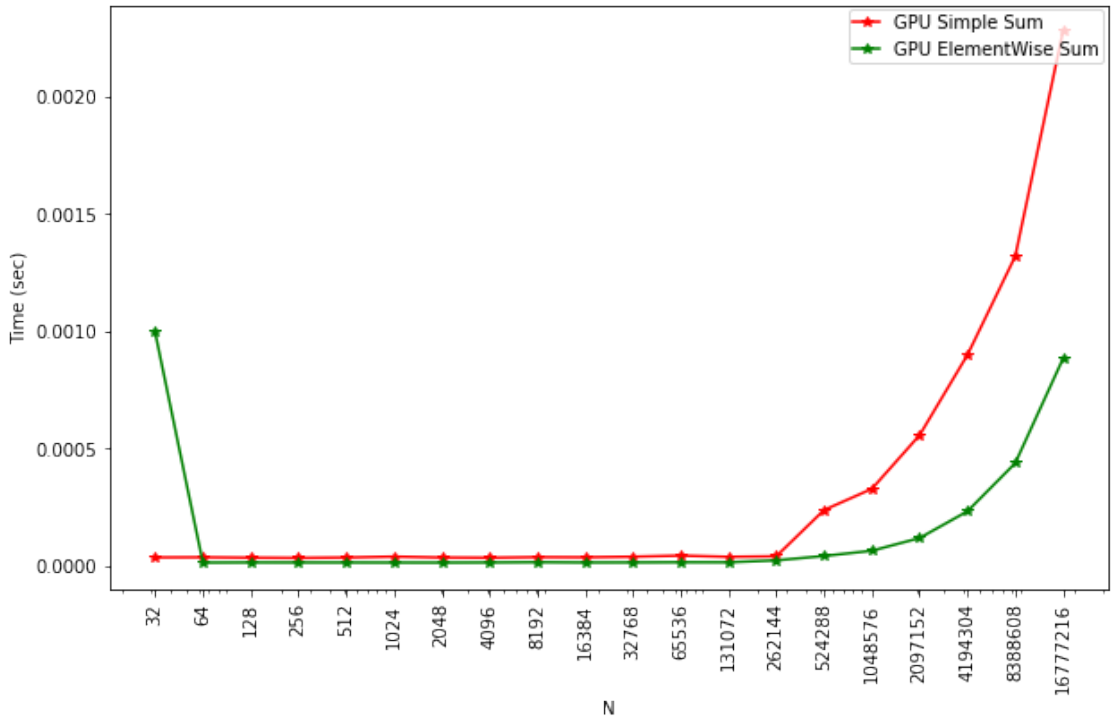
[111]:
```
import pycuda.driver as drv
start = drv.Event()
end = drv.Event()
```

[112]:
```
timeGPU2 = []
for n in dimension:
    v1_cpu = np.random.random(n).astype(myPrec)
    v2_cpu = np.random.random(n).astype(myPrec)
```

```
    v1_gpu = gpuarray.to_gpu(v1_cpu)
    v2_gpu = gpuarray.to_gpu(v2_cpu)
    vr_gpu  = gpuarray.to_gpu(v2_cpu)
    t3Mean=0
    for i in range(nLoops):
        start.record()
        myCudaFunc(v1_gpu,v2_gpu,vr_gpu)
        end.record()
        end.synchronize()
        secs = start.time_till(end)*1e-3
        t3Mean+=secs/nLoops
    timeGPU2.append(t3Mean)
    v1_gpu.gpudata.free()
    v2_gpu.gpudata.free()
    vr_gpu.gpudata.free()
```

[113]:
```
plt.figure(1,figsize=(10,6))
plt.semilogx(dimension,timeGPU1,'r-*',label='GPU Simple Sum')
plt.semilogx(dimension,timeGPU2,'g-*',label='GPU ElementWise Sum')
plt.ylabel('Time (sec)')
plt.xlabel('N')
plt.xticks(dimension, dimension, rotation='vertical')
plt.legend(loc=1,labelspacing=0.5,fancybox=True, handlelength=1.5,␣
 ↪borderaxespad=0.25, borderpad=0.25)
```

[113]: <matplotlib.legend.Legend at 0x7fcbf8fd4d90>



Implementazione con SourceModule. E' possibile variare la geometria di griglia e blocchi

[114]:
```
from pycuda.compiler import SourceModule
```

[115]:
```
presCPU, presGPU = np.float32, 'float'
cudaCode = open("VecAdd.cu","r")
cudaCode = cudaCode.read()
cudaCode = cudaCode.replace('float',presGPU )
myCode = SourceModule(cudaCode)
vectorAddKernel = myCode.get_function("vectorAdd")
vectorAddKernel.prepare('PPP')
```

[115]:
```
<pycuda._driver.Function at 0x7fcc1618db20>
```

[ ]:
```
timeGPU3 = []
occupancyMesure=[]
for nt in [32,64,128,256,512,1024]:
    aux = []
    auxOcc = []
    for n in dimension:
        v1_cpu = np.random.random(n).astype(myPrec)
        v2_cpu = np.random.random(n).astype(myPrec)
        v1_gpu = gpuarray.to_gpu(v1_cpu)
        v2_gpu = gpuarray.to_gpu(v2_cpu)
        vr_gpu  = gpuarray.to_gpu(v2_cpu)
        cudaBlock = (nt,1,1)
        cudaGrid    = (int((n+nt-1)/nt),1,1)

        cudaCode = open("VecAdd.cu","r")
        cudaCode = cudaCode.read()
        cudaCode = cudaCode.replace('float',presGPU )
        downVar = ['blockDim.x','blockDim.y','blockDim.z','gridDim.x','gridDim.
 ↪y','gridDim.z']
        upVar      = [str(cudaBlock[0]),str(cudaBlock[1]),str(cudaBlock[2]),
                      str(cudaGrid[0]),str(cudaGrid[1]),str(cudaGrid[2])]
        dicVarOptim = dict(zip(downVar,upVar))
        for i in downVar:
            cudaCode = cudaCode.replace(i,dicVarOptim[i])
        #print cudaCode
        myCode = SourceModule(cudaCode)
        vectorAddKernel = myCode.get_function("vectorAdd")
        vectorAddKernel.prepare('PPP')

        print ('Size= '+str(n)+" threadsPerBlock= "+str(nt))
        print (str(cudaBlock)+" "+str(cudaGrid))
```

```
        t5Mean = 0
        for i in range(nLoops):
            timeAux = vectorAddKernel.
↪prepared_timed_call(cudaGrid,cudaBlock,v1_gpu.gpudata,v2_gpu.gpudata,vr_gpu.
↪gpudata)
            t5Mean += timeAux()/nLoops
        aux.append(t5Mean)
        v1_gpu.gpudata.free()
        v2_gpu.gpudata.free()
        vr_gpu.gpudata.free()
    timeGPU3.append(aux)
    occupancyMesure.append(auxOcc)
```

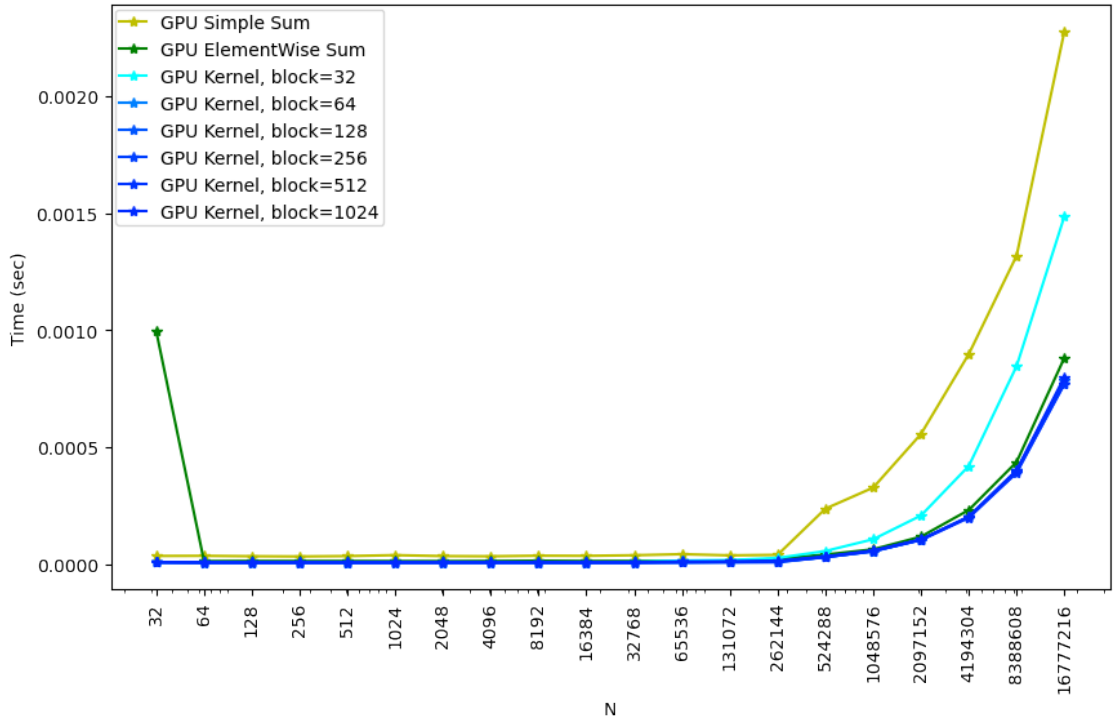
[117]:
```
timeGPU3[0]
```

[117]:
```
[7.949759974144397e-06,
 7.21632000524551e-06,
 7.254079999402168e-06,
 7.3756800033152106e-06,
 7.390080024488268e-06,
 7.483840039931238e-06,
 7.038079998455943e-06,
 8.753280001692472e-06,
 8.093119999393824e-06,
 8.041280033066873e-06,
 9.224319923669098e-06,
 1.487168000079691e-05,
 1.6904639974236497e-05,
 2.615999998524785e-05,
 5.546847987920043e-05,
 0.00010585823982954026,
 0.00020728544026613226,
 0.00041910720199346525,
 0.0008457052761316303,
 0.0014863404846191404]
```

[118]:
```
plt.figure(1,figsize=(10,6),dpi=100)
plt.semilogx(dimension,timeGPU1,'y-*',label='GPU Simple Sum')
plt.semilogx(dimension,timeGPU2,'g-*',label='GPU ElementWise Sum')
count = 0
for nt in [32,64,128,256,512,1024]:
    plt.semilogx(dimension,timeGPU3[count],'-*',label='GPU Kernel, block={0}'.
↪format(nt),color=(0,1./(count+1),1))
    count+=1
plt.ylabel('Time (sec)')
plt.xlabel('N')
plt.xticks(dimension, dimension, rotation='vertical')
```

```
plt.legend(loc=2,labelspacing=0.5,fancybox=True, handlelength=1.5,␣
 ↪borderaxespad=0.25, borderpad=0.25)
```

[118]: <matplotlib.legend.Legend at 0x7fcbf8ef8a50>



le sfumature di blu corrispondono a un diverso numero di thread per blocco

[118]:

9 Numba

9.1 Numba è un metodo generico per utilizzare funzioni di C (tra cui roba di GPU) in python.

[2]:
```
%reset
```

```
Once deleted, variables cannot be recovered. Proceed (y/[n])? y
```

E' necessario far trovare due librerie che normalmente non sono nel path

[3]:
```
import os
os.environ['NUMBAPRO_LIBDEVICE'] = "/usr/local/cuda-10.0/nvvm/libdevice"
```

```
os.environ['NUMBAPRO_NVVM'] = "/usr/local/cuda-10.0/nvvm/lib64/libnvvm.so"
```

abbiamo già visto che numpy sfrutta il fatto che molte funzioni (ufunc, universal functions) sono compilate in C e aggiscono sugli elementi dei vettori in maniera automatia. Nel seguente esempio confrontiamo le performance di numpy con quelle della normale radice quadrata su opportuni vettori

[4]:
```
import numpy as np
```

[5]:
```
import math
x = np.arange(int(1e7), dtype=np.float32)
%timeit np.sqrt(x)
%timeit [math.sqrt(xx) for xx in x]
```

```
5.64 ms ± 102 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
1.83 s ± 137 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

proviamo a compilare una funzione utente con numba. Per far questo usiamo il decoratore @vectorize

[6]:
```
import math
import numpy as np
from numba import vectorize
@vectorize
def cpu_sqrt(x):
  return math.sqrt(x)

%timeit cpu_sqrt(x)
```

```
9.01 ms ± 135 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

proviamo ora a fare una versione GPU in cui la ufunc è compilata per le essere eseguita sulla GPU. A differenza della funzione CPU è necessario specificare i tipi di output e input nel decoratore: output(input). L'array di input deve avere il tipo corretto.

[7]:
```
@vectorize(['float32(float32)'], target='cuda')
def gpu_sqrt(x):
    return math.sqrt(x)
```

[8]:
```
%timeit gpu_sqrt(x)
```

```
24.7 ms ± 3.96 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

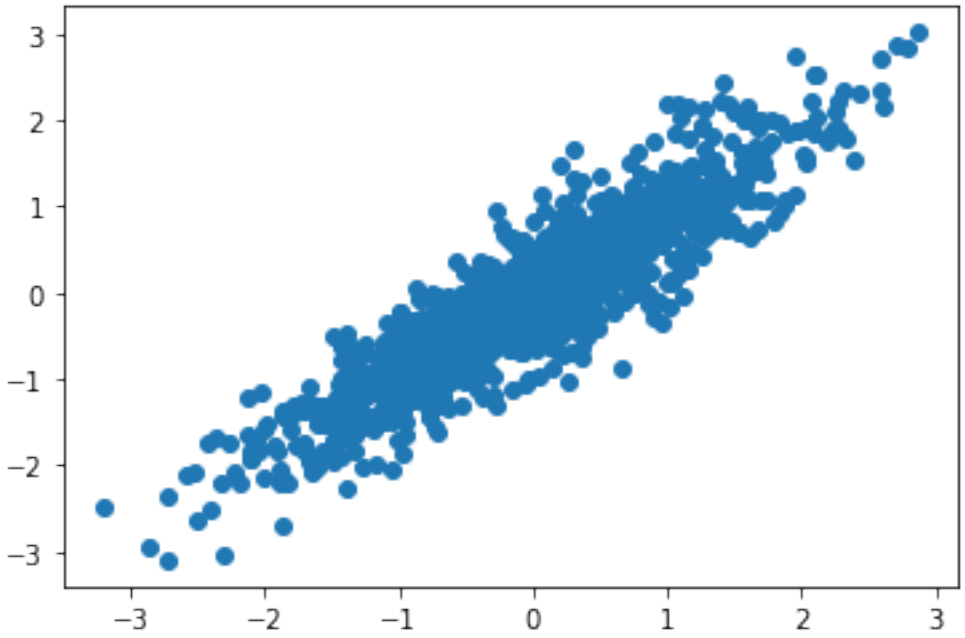
apparentemente la versione GPU è piu' lenta della versione CPU. La ragione di questo è che l'operazione che stiamo facendo è troppo semplice e quindi non abbiamo vantaggio computazionale rispetto all'overhead di copiatura dell'array sul device.

facciamo un esempio piu' complicato. Generiamo dei punti in 2D con correlazione.

[9]:
```
points = np.random.multivariate_normal([0,0], [[1.,0.9], [0.9,1.]], 1000).
↪astype(np.float32)
```

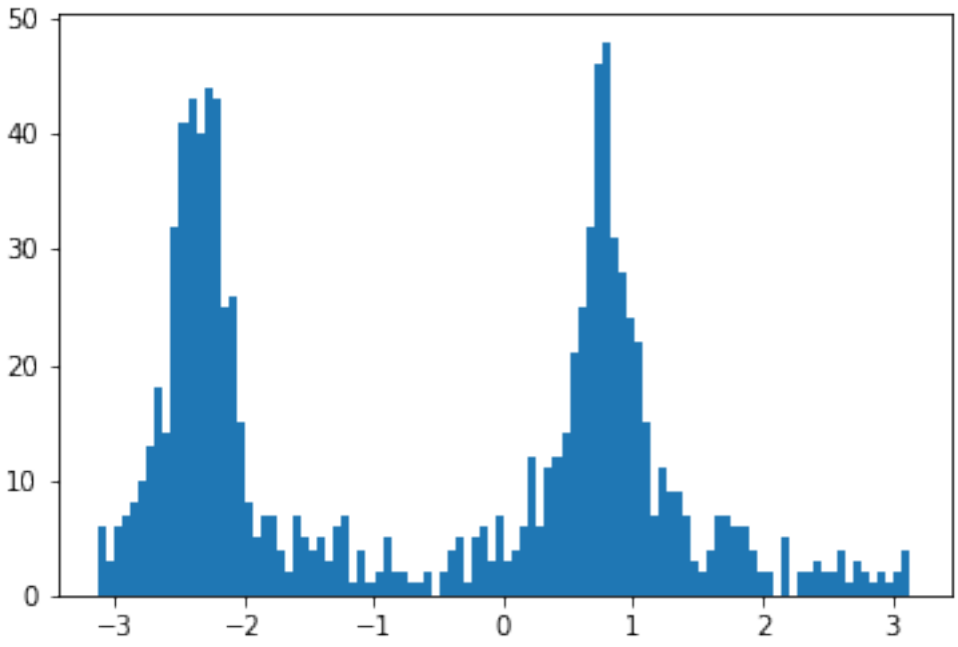
[10]:
```
import matplotlib.pyplot as plt
plt.scatter(points[:,0], points[:,1])
```

[10]:
```
<matplotlib.collections.PathCollection at 0x7fe0b7f4a2d0>
```



ora proviamo a trasformare in coordinate polari

[11]:
```
theta = np.arctan2(points[:,1], points[:,0])
_ = plt.hist(theta, bins=100)
```



vediamo 2 picchi perchè la correlazione puo' essere pi/4 o 3/4 pi. Proviamo a fare la stessa cosa con la GPU. Definiamo una ufunc gpu

[12]:
```
@vectorize(['float32(float32, float32)'],target='cuda')
def gpu_arctan2(y, x):
    theta = math.atan2(y,x)
    return theta
```

[13]:
```
theta = gpu_arctan2(points[:,1], points[:,0])
```

```
␣
↪---------------------------------------------------------------------------

ValueError                                Traceback (most recent call␣
↪last)

<ipython-input-13-2811dabcd6c9> in <module>
----> 1 theta = gpu_arctan2(points[:,1], points[:,0])


/usr/local/lib/python3.7/dist-packages/numba/cuda/vectorizers.py in␣
↪__call__(self, *args, **kws)
     34                         the input arguments.
     35         """
---> 36         return CUDAUFuncMechanism.call(self.functions, args, kws)
```

```
        37
        38     def reduce(self, arg, stream=0):


       /usr/local/lib/python3.7/dist-packages/numba/np/ufunc/deviceufunc.py in␣
↪call(cls, typemap, args, kws)
       285                 any_device = True
       286             else:
   --> 287                 dev_a = cr.to_device(a, stream=stream)
       288                 devarys.append(dev_a)
       289


       /usr/local/lib/python3.7/dist-packages/numba/cuda/vectorizers.py in␣
↪to_device(self, hostary, stream)
       168
       169     def to_device(self, hostary, stream):
   --> 170         return cuda.to_device(hostary, stream=stream)
       171
       172     def to_host(self, devary, stream):


       /usr/local/lib/python3.7/dist-packages/numba/cuda/cudadrv/devices.py in␣
↪_require_cuda_context(*args, **kws)
       230     def _require_cuda_context(*args, **kws):
       231         with _runtime.ensure_context():
   --> 232             return fn(*args, **kws)
       233
       234     return _require_cuda_context


       /usr/local/lib/python3.7/dist-packages/numba/cuda/api.py in␣
↪to_device(obj, stream, copy, to)
       119     if to is None:
       120         to, new = devicearray.auto_device(obj, stream=stream,␣
↪copy=copy,
   --> 121                                           user_explicit=True)
       122         return to
       123     if copy:


       /usr/local/lib/python3.7/dist-packages/numba/cuda/cudadrv/devicearray.py␣
↪in auto_device(obj, stream, copy, user_explicit)
       872                 copy=False,
       873                 subok=True)
   --> 874             sentry_contiguous(obj)
       875             devobj = from_array_like(obj, stream=stream)
```

```
        876         if copy:


        /usr/local/lib/python3.7/dist-packages/numba/cuda/cudadrv/devicearray.py␣
↪in sentry_contiguous(ary)
        846     core = array_core(ary)
        847     if not core.flags['C_CONTIGUOUS'] and not core.
↪flags['F_CONTIGUOUS']:
    --> 848         raise ValueError(errmsg_contiguous_buffer)
        849
        850


        ValueError: Array contains non-contiguous buffer and cannot be␣
↪transferred as a single memory region. Please ensure contiguous buffer with␣
↪numpy .ascontiguousarray()
```

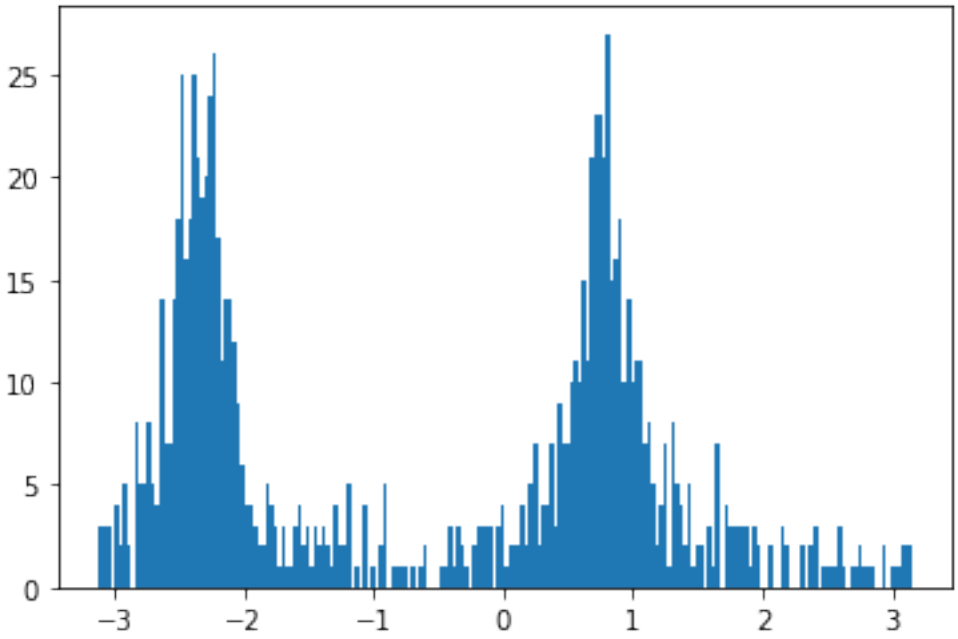
il fatto è che stiamo scrivendo una funzione in C, che lavora con i puntatori. la y del primo punto è accanto alla x del primo punto. Quindi la prima colonna (delle x) contiene degli elementi a salti di uno

non funziona perchè le slice che abbiamo considerato non sono valori contigui in memoria, invece si devono passare array contigui come argomento. Per fortuna c'è una funzione per renderli contigui.

[14]:
```
x = np.ascontiguousarray(points[:,0])
y = np.ascontiguousarray(points[:,1])
```

[15]:
```
theta = gpu_arctan2(y, x)
_ = plt.hist(theta, bins=200)
```

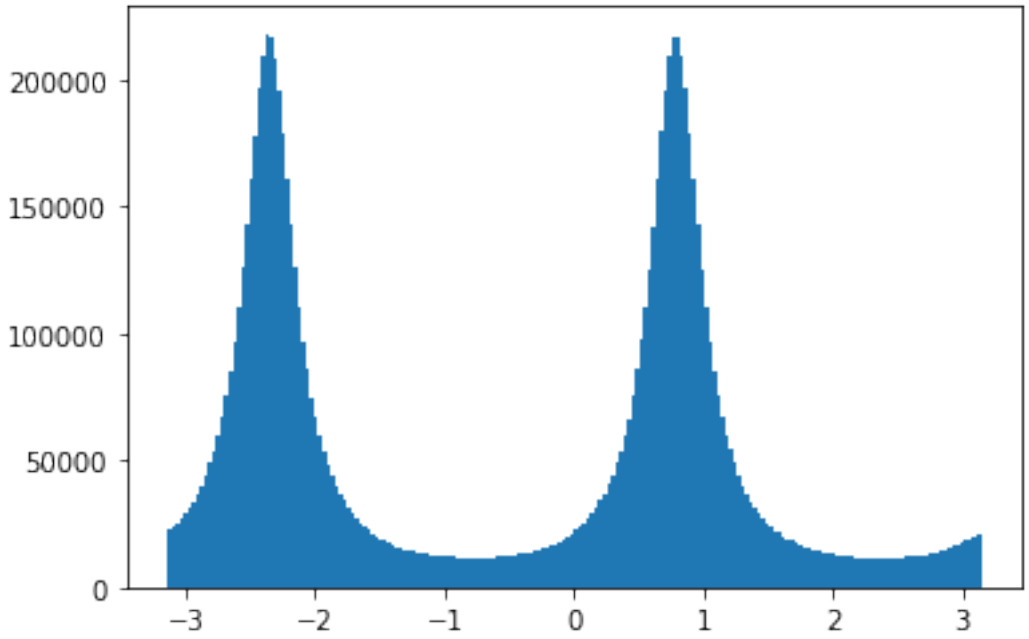
```
/usr/local/lib/python3.7/dist-packages/numba/cuda/dispatcher.py:488:
NumbaPerformanceWarning: Grid size 1 will likely result in GPU under-utilization
due to low occupancy.
  warn(NumbaPerformanceWarning(msg))
```



funziona. Proviamo a farlo con piu' punti

[16]:
```
points = np.random.multivariate_normal([0,0], [[1.,0.9], [0.9,1.]], int(1e7)).
↪astype(np.float32)
x = np.ascontiguousarray(points[:,0])
y = np.ascontiguousarray(points[:,1])
```

[17]:
```
_ = plt.hist(gpu_arctan2(y, x), bins=200)
```



quantifichiamo il tempo

[18]:
```
%timeit np.arctan2(y, x)
```

```
299 ms ± 5.28 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[19]:
```
%timeit gpu_arctan2(y, x)
```

```
36 ms ± 769 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

visto che ci siamo confrontiamo anche con plain python

[20]:
```
%timeit [math.atan2(point[1], point[0]) for point in points]
```

```
5.12 s ± 49.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

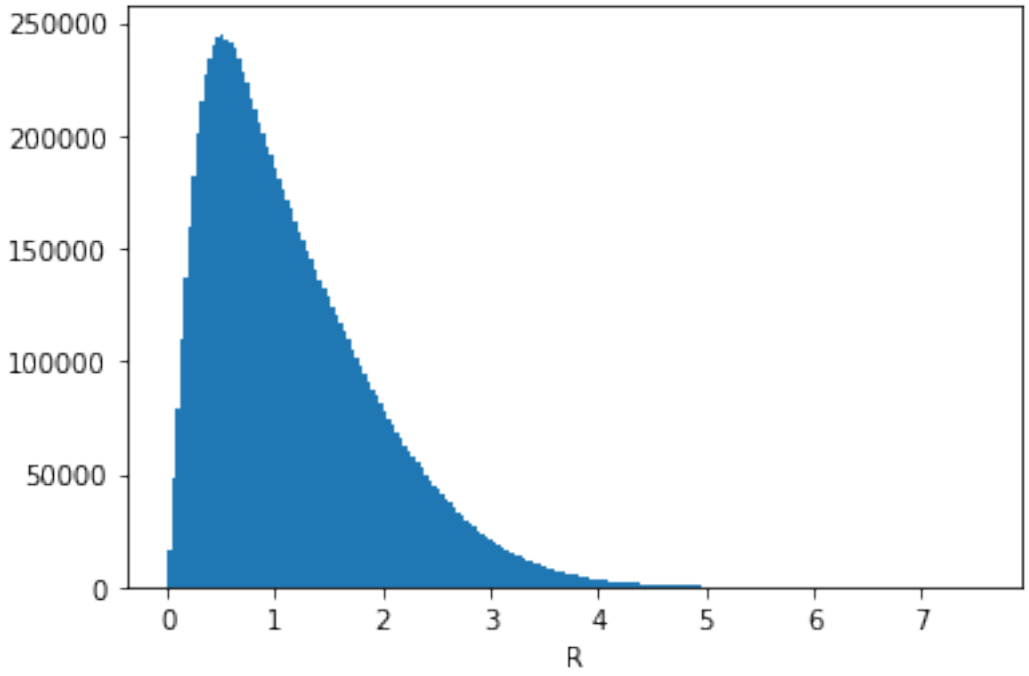
Nelle ufunc (su GPU o meno) che abbiamo visto fino ad ora, l'argomento è un array e il risultato è un array di scalari delle stesse dimensioni ottenuto applicando una funzione su ogni elemento dell'array di input. Vogliamo generalizzare questa cosa permettendo cosa piu' complicate, come il fatto che il calcolo avvenga solo su una parte dell'array di input e che l'output possa essere anche un array di dimensioni differenti da quello di input. Si usa guvectorize

[21]:
```
from numba import guvectorize

@guvectorize(['(float32[:], float32[:])'],
             '(n)->(n)',
             target='cuda')
def gpu_polar(vec, out):
```

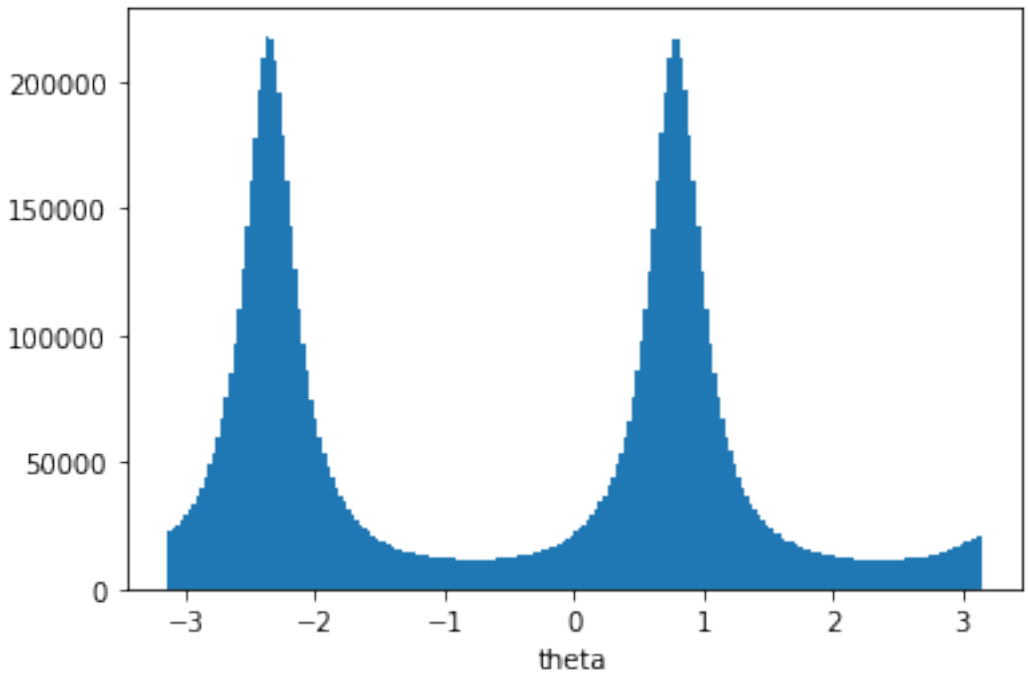
```
    x = vec[0]
    y = vec[1]
    out[0] = math.sqrt(x**2 + y**2)
    out[1] = math.atan2(y,x)
```

[22]:
```
polar_coords = gpu_polar(points)
```

[23]:
```
_ = plt.hist(polar_coords[:,0], bins=200)
_ = plt.xlabel('R')
```



[24]:
```
_ = plt.hist(polar_coords[:,1], bins=200)
_ = plt.xlabel('theta')
```



facciamo un altro esempio su guvectorize, ovvero la media per righe in un vettore 2D

[25]:
```
@guvectorize(['(float32[:], float32[:])'],
             '(n)->()',
             target='cuda')
def gpu_average(array, out):
    acc = 0
    for val in array:
        acc += val
    out[0] = acc/len(array)
    print(len(array))
```

definiamo il vettore 2D

[26]:
```
a = np.arange(100).reshape(20, 5).astype(np.float32)
a
```

[26]:
```
array([[ 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.]], dtype=float32)
```

[27]:
```
gpu_average(a)
```

```
/usr/local/lib/python3.7/dist-packages/numba/cuda/dispatcher.py:488:
NumbaPerformanceWarning: Grid size 1 will likely result in GPU under-utilization
due to low occupancy.
  warn(NumbaPerformanceWarning(msg))
```

[27]:
```
array([ 2.,  7., 12., 17., 22., 27., 32., 37., 42., 47., 52., 57., 62.,
       67., 72., 77., 82., 87., 92., 97.], dtype=float32)
```

10 Generare il PDF del Notebook

[ ]:
```
!apt-get install texlive texlive-xetex texlive-latex-extra pandoc
!pip install pypandoc
```

si deve montare il proprio google drive (seguire il link per ottenere la chiave di accesso)

[29]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

si deve copiare il notebook nella directory della macchina virtuale

[30]:
```
!cp "drive/My Drive/Colab Notebooks/handson_gpu_2022.ipynb" ./
```

ora si puo' convertire in pdf

[ ]:
```
!jupyter nbconvert --to PDF "handson_gpu_2022.ipynb"
```

scaricare il file pdf prodotto dal menu files nel pannelo di sinistra (premere il destro sul file e fare download)

[ ]:
```
!jupyter nbconvert --to LaTeX "handson_gpu_2022.ipynb"
```