

handson_gpu_2020

November 1, 2020

1 Setup Iniziale

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

```
[ ]: import pycuda
```

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

```
[ ]: import pycuda
```

4. Controlla la versione di CUDA installata

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

2 Esplorare la Bash

```
[ ]: !ls
```

```
[ ]: mkdir test_dir
```

```
[ ]: cd test_dir
```

```
[ ]: ls
```

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

```
[ ]: ls
```

```
[ ]: rm ciao
```

```
[ ]: ls
```

```
[ ]: pwd
```

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

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

```
[ ]:
```

3 Caratteristiche della GPU in uso

Proviamo a capire le caratteristiche della GPU che abbiamo a disposizione.

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

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

```
[ ]: 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)
```

oppure anche con il metodo DeviceData()

```
[ ]: from pycuda import autoint
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))
```

4 Esempio GPU in C

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

```
[ ]: %%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;
}

```

```
[ ]: ls
```

```
[ ]: !nvcc -o VecAdd VecAdd.cu
```

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

5 Implementazione con pycuda

Facciamo un primo esempio con pycuda

importiamo i moduli che ci servono

```
[ ]: 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

```
[ ]: 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

```
[ ]: 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 è semplicemente usare il +

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

stampiamo i risultati

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

```
[ ]: c_gpu
```

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

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

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

```
[ ]: c_gpu
```

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

```
[ ]: 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")
```

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

```
[ ]: c_gpu
```

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

```
[ ]: 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)

```
[ ]: !ls
```

```
[ ]: cudaCode = open("VecAdd.cu", "r")
myCUDACode = cudaCode.read()
```

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

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

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

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

definiamo la "geometria" della GPU che vogliamo usare

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

resettiamo il vettore c_gpu (per essere sicuri sia vuoto)

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

Il puntatore nella memoria gpu è dato dall'attributo gpudata

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

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

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

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

6 Somma di Matrici

Puliamo la memoria

```
[ ]: %reset
```

importiamo le cose che ci servono

```
[ ]: 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

```
[ ]: 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

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

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

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

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

copiamo gli array sulla gpu

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

```
[ ]: c_gpu
```

facciamo la somma prima sull'host

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

```
[ ]: c_cpu
```

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

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

definiamo il kernel gpu per fare la somma

```
[ ]: 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

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

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

[ ]: type(addMatrix)
```

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

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

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

```
[ ]: 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 somma 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 function
kernelFunction.prepared_call(grid,block,arg1.gpudata,arg2,...) # When using GPU arrays, they should be GPU arrays

il primo metodo è, per noi

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

con la preparation è possibile misurare il tempo di esecuzione

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

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

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

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

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

controlliamo il risultato per cpu e gpu

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

per confrontare meglio, guardiamo i plot

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

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

in effetti i risultati sono uguali

7 Moltiplicazione tra matrici

scriviamo un kernel per la moltiplicazione di matrici

```
[ ]: 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

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

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

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

sulla CPU sarà invece

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


vediamo il risultato è lo stesso

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

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

[ ]: dotAB

[ ]: c_gpu

[ ]: 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)

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

[ ]: a_cpu.dtype

[ ]: 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;

      }
      '''

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

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

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

[ ]: c_gpu.dtype

[ ]: dotAB.dtype
```

```
[ ]: diff = np.abs(c_gpu.get()-dotAB)
[ ]: plt.imshow(diff,interpolation='none')
plt.colorbar()
[ ]:
```

8 Ancora sulla somma di vettori

```
[ ]: %reset

Vogliamo confrontare i tempi per la somma di vettori di dimensione variabile, tra CPU e GPU
Iniziamo con la versione CPU
[ ]: %matplotlib inline
from matplotlib import pyplot as plt
[ ]: import numpy as np
[ ]: from time import time
def myColorRand():
    return (np.random.random(),np.random.random(),np.random.random())
[ ]: dimension = [2**i for i in range(5,25) ]
myPrec = np.float32
[ ]: dimension
[ ]: 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)
[ ]: 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)

```

[ ]: import pycuda
    from pycuda import gpuarray

[ ]: 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()

[ ]: 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,labelspace=0.5,fancybox=True, handlelength=1.5,
        →borderaxespad=0.25, borderpad=0.25)
    plt.show()

[ ]: 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, labelspace=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

```
[ ]: 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, labelspace=0.5, fancybox=True, handlelength=1.5,
→borderaxespad=0.25, borderpad=0.25)
plt.show()
```

proviamo ad usare elementwise (secondo metodo)

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

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

```
[ ]: 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()
```

```
[ ]: 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, labelspace=0.5, fancybox=True, handlelength=1.5,
→borderaxespad=0.25, borderpad=0.25)
```

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

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

[ ]: 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')

[ ]: timeGPU3 = []
occupancyMeasure = []
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)

```

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

```

[ ]: 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,labelspace=0.5,fancybox=True, handlelength=1.5,
→borderaxespad=0.25, borderpad=0.25)

```

```
[ ]:
```

9 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)

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

si deve copiare il notebook nella directory della macchina virtuale

```
[9]: !cp "drive/My Drive/Colab Notebooks/handson_gpu_2020.ipynb" ./
```

ora si puo' convertire in pdf

```
[8]: !jupyter nbconvert --to PDF "handson_gpu_2020.ipynb"
```

[NbConvertApp] Converting notebook handson_gpu_2020.ipynb to PDF

[NbConvertApp] Writing 102045 bytes to ./notebook.tex

```
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times: [u'xelatex', u'./notebook.tex',
'-quiet']
[NbConvertApp] Running bibtex 1 time: [u'bibtex', u'./notebook']
[NbConvertApp] WARNING | bibtex had problems, most likely because there were no
citations
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 76597 bytes to handson_gpu_2020.pdf
```

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

[]: