PyCUDA_multiplicar_matrices

Contenido

Instalación de la librería

```
!pip install pycuda
```

```
Requirement already satisfied: pycuda in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: pytools>=2011.2 in /usr/local/lib/python3.10/dis
Requirement already satisfied: appdirs>=1.4.0 in /usr/local/lib/python3.10/dist
Requirement already satisfied: mako in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: platformdirs>=2.2.0 in /usr/local/lib/python3.10
Requirement already satisfied: typing-extensions>=4.0 in /usr/local/lib/python3
Requirement already satisfied: MarkupSafe>=0.9.2 in /usr/local/lib/python3.10/d
```

Importando las librerías

```

```

```
# Para inicializar automáticamente PyCUDA y la GPU
import pycuda.autoinit

# Para interactuar directamente con el controlador de la GPU
import pycuda.driver as drv

# Para manipular arrays y operaciones en la CPU
import numpy as np

# Para compilar y cargar módulos CUDA en la GPU
from pycuda.compiler import SourceModule

# Para medir los tiempos
import time

# Para generar gráficos
import matplotlib.pyplot as plt
```

Verificación de los recursos de GPU

```
drv.init()
print("%d dispositivo(s) encontrado." % drv.Device.count())
for i in range(drv.Device.count()):
  dev = drv.Device(i)
  print("\n Dispositivo #%d: %s" % (i, dev.name()))
  print(" Memoria Total: %s GB" % (dev.total_memory() // (1024 * 1024 * 1024)))
  print(" Capacidad de Computación: %d.%d" % dev.compute_capability())
```

```
1 dispositivo(s) encontrado.

 Dispositivo #0: Tesla T4
 Memoria Total: 14 GB
 Capacidad de Computación: 7.5
```

Matrices a multiplicar

```
# Establece la semilla para reproducibilidad
np.random.seed(42)

# Genera dos matrices aleatorias enteras entre 0 y 10 en la CPU (host)
a = np.random.randint(0, 11, size=(128, 128)).astype(np.float32)
b = np.random.randint(0, 11, size=(128, 128)).astype(np.float32)

# Imprime las matrices originales
print("Matriz 'a':")
print(a)
print("\nMatriz 'b':")
print(b)
```

```
Matriz 'a':
[[ 6.  3. 10. ...  6.  6. 10.]
 [ 8.  9.  9. ...  0.  1.  0.]
 [ 4.  4. 10. ...  2.  8.  9.]
 ...
 [ 4.  4.  7. ...  9.  5.  6.]
 [ 8.  0.  3. ...  3.  9.  8.]
 [ 3.  3.  6. ...  0.  1.  5.]]

Matriz 'b':
[[ 8.  4.  3. ...  1.  7.  5.]
 [ 6.  0.  5. ...  9.  0.  7.]
 [ 0.  8.  5. ...  5.  0.  1.]
 ...
 [ 0.  1.  3. ...  6.  1.  9.]
 [ 1.  5.  8. ... 10.  1. 10.]
 [ 6.  0.  3. ...  8.  9.  4.]]
```

Usando la GPU

```

```

```
def matrix_multiply_gpu_block_size(a, b, block_size=(32, 32, 1)):
    # Verifica si las matrices son compatibles para la multiplicación
    if a.shape[1] != b.shape[0]:
        raise ValueError("Las dimensiones de las matrices no son compatibles pa

    # Transfiere las matrices a la GPU
    a_gpu = drv.mem_alloc(a.nbytes)
    b_gpu = drv.mem_alloc(b.nbytes)
    drv.memcpy_htod(a_gpu, a)
    drv.memcpy_htod(b_gpu, b)

    # Define un módulo CUDA con un kernel para multiplicar matrices
    mod = SourceModule("""
        __global__ void matrix_multiply(float *result, float *a, float *b, int
        {
            int row = threadIdx.y + blockIdx.y * blockDim.y;
            int col = threadIdx.x + blockIdx.x * blockDim.x;
            int idx = row * K + col;

            result[idx] = 0;

            for (int k = 0; k < M; ++k)
                result[idx] += a[row * M + k] * b[k * K + col];
        }
    """)

    # Obtiene el kernel de multiplicación de matrices
    matrix_multiply_kernel = mod.get_function("matrix_multiply")

    # Configura las dimensiones del bloque y de la cuadrícula
    grid_size = (int(np.ceil(a.shape[0] / block_size[0])),
                 int(np.ceil(b.shape[1] / block_size[1])))

    # Crea una matriz en la GPU para almacenar el resultado
    result_gpu = np.zeros((a.shape[0], b.shape[1]), dtype=np.float32)

    # Llama al kernel en la GPU para multiplicar las matrices
    matrix_multiply_kernel(
        drv.Out(result_gpu), a_gpu, b_gpu,
        np.int32(a.shape[0]), np.int32(a.shape[1]), np.int32(b.shape[1]),
        block=block_size, grid=grid_size)

    return result_gpu

# Mide el tiempo de inicio
start_time_gpu = time.time()

# Realiza la multiplicación matricial en la GPU
result_gpu = matrix_multiply_gpu_block_size(a, b, block_size=(32, 32, 1))

# Mide el tiempo de finalización
end_time_gpu = time.time()

# Imprime el resultado de la multiplicación matricial en la GPU
print("\nResultado de la multiplicación matricial en la GPU:")
```

```
print(result_gpu)

# Calcula el tiempo transcurrido en la GPU
elapsed_time_gpu = end_time_gpu - start_time_gpu
print(f"\nTiempo transcurrido en la GPU: {elapsed_time_gpu} segundos")
```

```
Resultado de la multiplicación matricial en la GPU:
[[2989. 3121. 3238. ... 3319. 3330. 3838.]
 [2875. 2998. 3065. ... 3057. 3073. 3606.]
 [2782. 2790. 3103. ... 3077. 2902. 3281.]
 ...
 [2553. 2664. 2917. ... 2919. 2840. 3026.]
 [2984. 3180. 3385. ... 3315. 3361. 3586.]
 [3057. 2932. 3266. ... 3318. 3434. 3845.]]

Tiempo transcurrido en la GPU: 0.002301454544067383 segundos
```

Usando la CPU

```
def multiply_matrices_with_loops(a, b):
    rows_a, cols_a = a.shape
    rows_b, cols_b = b.shape

    if cols_a != rows_b:
        raise ValueError("Las dimensiones de las matrices no son compatibles pa

    result = np.zeros((rows_a, cols_b), dtype=np.float32)

    for i in range(rows_a):
        for j in range(cols_b):
            for k in range(cols_a):
                result[i, j] += a[i, k] * b[k, j]

    return result

# Mide el tiempo de inicio
start_time_cpu = time.time()

# Realiza la multiplicación matricial en la CPU con bucles
result_cpu = multiply_matrices_with_loops(a, b)

# Mide el tiempo de finalización
end_time_cpu = time.time()

# Imprime el resultado de la multiplicación matricial en la CPU
print("\nResultado de la multiplicación matricial en la CPU:")
print(result_cpu)

# Calcula el tiempo transcurrido en la CPU
elapsed_time_cpu = end_time_cpu - start_time_cpu
print(f"\nTiempo transcurrido en la CPU: {elapsed_time_cpu} segundos")
```

```
Resultado de la multiplicación matricial en la CPU:
[[2989. 3121. 3238. ... 3319. 3330. 3838.]
 [2875. 2998. 3065. ... 3057. 3073. 3606.]
 [2782. 2790. 3103. ... 3077. 2902. 3281.]
 ...
 [2553. 2664. 2917. ... 2919. 2840. 3026.]
 [2984. 3180. 3385. ... 3315. 3361. 3586.]
 [3057. 2932. 3266. ... 3318. 3434. 3845.]]

Tiempo transcurrido en la CPU: 2.9058544635772705 segundos
```

```
np.array_equal(result_gpu ,result_cpu)
```

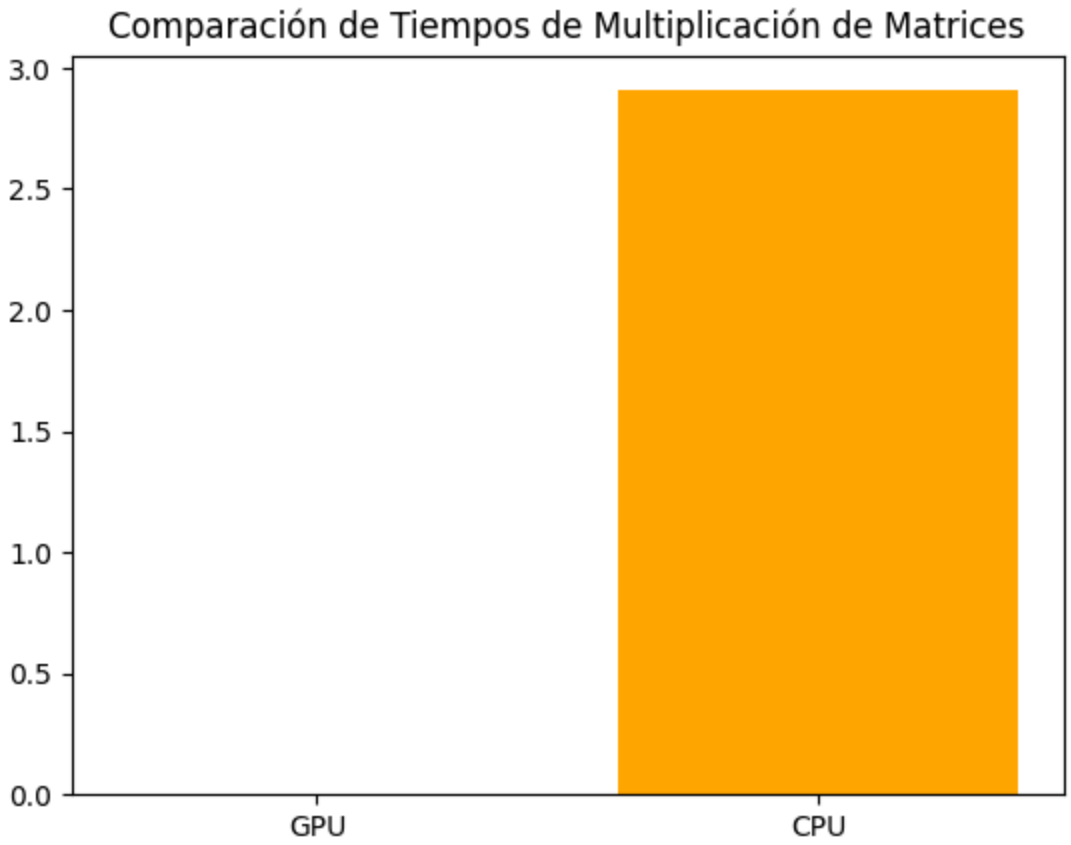
```
True
```

Comparando los tiempos

```
# Nombres de las operaciones
labels = ['GPU', 'CPU']

# Valores de tiempo
values = [elapsed_time_gpu, elapsed_time_cpu]

# Graficar en barras con escala logarítmica en el eje y
plt.bar(labels, values, color=['green', 'orange'])
plt.title('Comparación de Tiempos de Multiplicación de Matrices')
plt.show()
```




```
elapsed_time_cpu/elapsed_time_gpu
```

```
1262.616492282192
```

PyCUDA_multiplicar_sumar_elementos

Imprimir en PDF

Contenido

Instalación de la librería

```
!pip install pycuda
```

```
Requirement already satisfied: pycuda in /usr/lib/python3/dist-packages (2018.1.1)
```

Importando las librerías

```
# Para inicializar automáticamente PyCUDA y la GPU
import pycuda.autoinit

# Para interactuar directamente con el controlador de la GPU
import pycuda.driver as drv

# Para manipular arrays y operaciones en la CPU
import numpy as np

# Para compilar y cargar módulos CUDA en la GPU
from pycuda.compiler import SourceModule

# Para medir los tiempos
import time

# Para generar gráficos
import matplotlib.pyplot as plt
```

Verificación de los recursos de GPU

```
drv.init()
print("%d dispositivo(s) encontrado." % drv.Device.count())
for i in range(drv.Device.count()):
  dev = drv.Device(i)
  print("\n Dispositivo #%d: %s" % (i, dev.name()))
  print(" Memoria Total: %s GB" % (dev.total_memory() // (1024 * 1024 * 1024)))
  print(" Capacidad de Computación: %d.%d" % dev.compute_capability())
```

```
1 dispositivo(s) encontrado.

 Dispositivo #0: NVIDIA GeForce GTX 1070 with Max-Q Design
 Memoria Total: 7 GB
 Capacidad de Computación: 6.1
```

Ejemplo del uso de PyCUDA, multiplicación por número

```
# Establece la semilla para reproducibilidad
np.random.seed(42)

# Crea una matriz con números aleatorios enteros entre 0 y 10
a = np.random.randint(0, 11, size=(8,8))
# Convierte la matriz a formato de punto flotante de 32 bits
a = a.astype(np.float32)

a
```

```
array([[ 6.,  3., 10.,  7.,  4.,  6.,  9.,  2.],
       [ 6., 10., 10.,  7.,  4.,  3.,  7.,  7.],
       [ 2.,  5.,  4.,  1.,  7.,  5.,  1.,  4.],
       [ 0.,  9.,  5.,  8.,  0., 10., 10.,  9.],
       [ 2.,  6.,  3.,  8.,  2.,  4.,  2.,  6.],
       [ 4.,  8.,  6.,  1.,  3.,  8.,  1.,  9.],
       [ 8.,  9.,  4.,  1.,  3.,  6.,  7.,  2.],
       [ 0.,  3.,  1.,  7.,  3.,  1.,  5.,  5.]], dtype=float32)
```

GPU

```

```

```
# Asigna memoria en la GPU para la matriz
a_gpu = drv.mem_alloc(a.nbytes)

# Copia los datos de la matriz de la memoria del host a la memoria de la GPU
drv.memcpy_htod(a_gpu, a)

# Define un módulo de código CUDA
mod = SourceModule("""
  __global__ void doublify(float *a)
  {
    //int blockId = (gridDim.x * blockIdx.y) + blockIdx.x;
    //int idx = (blockId * (blockDim.x * blockDim.y)) + (threadIdx.y * blockDim.x) + threadIdx

    int idx= (blockIdx.x * blockDim.x + threadIdx.x) + (blockIdx.y * blockDim.y + threadIdx.y)

    //int idx = threadIdx.x + threadIdx.y * blockDim.x;

    a[idx] *= 2;

    printf("threadIdx.x: %d, threadIdx.y:%d, blockDim.x:%d, blockDim.y :%d idx: %d\\n",threadI
  }
  """)

# Obtiene la función 'doublify' del módulo CUDA
func = mod.get_function("doublify")

# Inicia el tiempo de medición para la operación CUDA
start_time_cuda = time.time()

# Ejecuta la función 'doublify' en un bloque de 4x4 hilos

func(a_gpu, block=(4,4,1), grid=(2,2))
#func(a_gpu, block=(8,8,1))

# Sincroniza el contexto de CUDA para asegurarse de que todas las operaciones en la GPU se hay
drv.Context.synchronize()

# Crea una matriz vacía con la misma forma y tipo que la matriz 'a'
a_doubled = np.empty_like(a)

# Copia los datos de la matriz procesada desde la memoria de la GPU al host
drv.memcpy_dtoh(a_doubled, a_gpu)

# Detiene el tiempo de medición para la operación CUDA
end_time_cuda = time.time()
# Calcula el tiempo total de la operación CUDA
time_cuda = end_time_cuda - start_time_cuda

print("Tiempo de ejecución CUDA: ", time_cuda)

# Imprime la matriz procesada
print("Matriz procesada:\n")
print(a_doubled)

# Imprime la matriz original
print("Matriz original:\n")
print(a)
```

```
threadIdx.x: 0, threadIdx.y:0, blockDim.x:4, blockDim.y :4 idx: 36
threadIdx.x: 1, threadIdx.y:0, blockDim.x:4, blockDim.y :4 idx: 37
threadIdx.x: 2, threadIdx.y:0, blockDim.x:4, blockDim.y :4 idx: 38
threadIdx.x: 3, threadIdx.y:0, blockDim.x:4, blockDim.y :4 idx: 39
threadIdx.x: 0, threadIdx.y:1, blockDim.x:4, blockDim.y :4 idx: 44
threadIdx.x: 1, threadIdx.y:1, blockDim.x:4, blockDim.y :4 idx: 45
threadIdx.x: 2, threadIdx.y:1, blockDim.x:4, blockDim.y :4 idx: 46
threadIdx.x: 3, threadIdx.y:1, blockDim.x:4, blockDim.y :4 idx: 47
threadIdx.x: 0, threadIdx.y:2, blockDim.x:4, blockDim.y :4 idx: 52
threadIdx.x: 1, threadIdx.y:2, blockDim.x:4, blockDim.y :4 idx: 53
threadIdx.x: 2, threadIdx.y:2, blockDim.x:4, blockDim.y :4 idx: 54
threadIdx.x: 3, threadIdx.y:2, blockDim.x:4, blockDim.y :4 idx: 55
threadIdx.x: 0, threadIdx.y:3, blockDim.x:4, blockDim.y :4 idx: 60
threadIdx.x: 1, threadIdx.y:3, blockDim.x:4, blockDim.y :4 idx: 61
threadIdx.x: 2, threadIdx.y:3, blockDim.x:4, blockDim.y :4 idx: 62
threadIdx.x: 3, threadIdx.y:3, blockDim.x:4, blockDim.y :4 idx: 63
threadIdx.x: 0, threadIdx.y:0, blockDim.x:4, blockDim.y :4 idx: 4
threadIdx.x: 1, threadIdx.y:0, blockDim.x:4, blockDim.y :4 idx: 5
threadIdx.x: 2, threadIdx.y:0, blockDim.x:4, blockDim.y :4 idx: 6
threadIdx.x: 3, threadIdx.y:0, blockDim.x:4, blockDim.y :4 idx: 7
threadIdx.x: 0, threadIdx.y:1, blockDim.x:4, blockDim.y :4 idx: 12
threadIdx.x: 1, threadIdx.y:1, blockDim.x:4, blockDim.y :4 idx: 13
threadIdx.x: 2, threadIdx.y:1, blockDim.x:4, blockDim.y :4 idx: 14
threadIdx.x: 3, threadIdx.y:1, blockDim.x:4, blockDim.y :4 idx: 15
threadIdx.x: 0, threadIdx.y:2, blockDim.x:4, blockDim.y :4 idx: 20
threadIdx.x: 1, threadIdx.y:2, blockDim.x:4, blockDim.y :4 idx: 21
threadIdx.x: 2, threadIdx.y:2, blockDim.x:4, blockDim.y :4 idx: 22
threadIdx.x: 3, threadIdx.y:2, blockDim.x:4, blockDim.y :4 idx: 23
threadIdx.x: 0, threadIdx.y:3, blockDim.x:4, blockDim.y :4 idx: 28
threadIdx.x: 1, threadIdx.y:3, blockDim.x:4, blockDim.y :4 idx: 29
threadIdx.x: 2, threadIdx.y:3, blockDim.x:4, blockDim.y :4 idx: 30
threadIdx.x: 3, threadIdx.y:3, blockDim.x:4, blockDim.y :4 idx: 31
threadIdx.x: 0, threadIdx.y:0, blockDim.x:4, blockDim.y :4 idx: 0
threadIdx.x: 1, threadIdx.y:0, blockDim.x:4, blockDim.y :4 idx: 1
threadIdx.x: 2, threadIdx.y:0, blockDim.x:4, blockDim.y :4 idx: 2
threadIdx.x: 3, threadIdx.y:0, blockDim.x:4, blockDim.y :4 idx: 3
threadIdx.x: 0, threadIdx.y:1, blockDim.x:4, blockDim.y :4 idx: 8
threadIdx.x: 1, threadIdx.y:1, blockDim.x:4, blockDim.y :4 idx: 9
threadIdx.x: 2, threadIdx.y:1, blockDim.x:4, blockDim.y :4 idx: 10
threadIdx.x: 3, threadIdx.y:1, blockDim.x:4, blockDim.y :4 idx: 11
threadIdx.x: 0, threadIdx.y:2, blockDim.x:4, blockDim.y :4 idx: 16
threadIdx.x: 1, threadIdx.y:2, blockDim.x:4, blockDim.y :4 idx: 17
threadIdx.x: 2, threadIdx.y:2, blockDim.x:4, blockDim.y :4 idx: 18
threadIdx.x: 3, threadIdx.y:2, blockDim.x:4, blockDim.y :4 idx: 19
threadIdx.x: 0, threadIdx.y:3, blockDim.x:4, blockDim.y :4 idx: 24
threadIdx.x: 1, threadIdx.y:3, blockDim.x:4, blockDim.y :4 idx: 25
threadIdx.x: 2, threadIdx.y:3, blockDim.x:4, blockDim.y :4 idx: 26
threadIdx.x: 3, threadIdx.y:3, blockDim.x:4, blockDim.y :4 idx: 27
threadIdx.x: 0, threadIdx.y:0, blockDim.x:4, blockDim.y :4 idx: 32
threadIdx.x: 1, threadIdx.y:0, blockDim.x:4, blockDim.y :4 idx: 33
threadIdx.x: 2, threadIdx.y:0, blockDim.x:4, blockDim.y :4 idx: 34
threadIdx.x: 3, threadIdx.y:0, blockDim.x:4, blockDim.y :4 idx: 35
threadIdx.x: 0, threadIdx.y:1, blockDim.x:4, blockDim.y :4 idx: 40
threadIdx.x: 1, threadIdx.y:1, blockDim.x:4, blockDim.y :4 idx: 41
threadIdx.x: 2, threadIdx.y:1, blockDim.x:4, blockDim.y :4 idx: 42
threadIdx.x: 3, threadIdx.y:1, blockDim.x:4, blockDim.y :4 idx: 43
threadIdx.x: 0, threadIdx.y:2, blockDim.x:4, blockDim.y :4 idx: 48
threadIdx.x: 1, threadIdx.y:2, blockDim.x:4, blockDim.y :4 idx: 49
threadIdx.x: 2, threadIdx.y:2, blockDim.x:4, blockDim.y :4 idx: 50
threadIdx.x: 3, threadIdx.y:2, blockDim.x:4, blockDim.y :4 idx: 51
threadIdx.x: 0, threadIdx.y:3, blockDim.x:4, blockDim.y :4 idx: 56
threadIdx.x: 1, threadIdx.y:3, blockDim.x:4, blockDim.y :4 idx: 57
threadIdx.x: 2, threadIdx.y:3, blockDim.x:4, blockDim.y :4 idx: 58
threadIdx.x: 3, threadIdx.y:3, blockDim.x:4, blockDim.y :4 idx: 59
Tiempo de ejecución CUDA:  0.0026025772094726562
```

```
Matriz procesada:

[[12.  6. 20. 14.  8. 12. 18.  4.]
 [12. 20. 20. 14.  8.  6. 14. 14.]
 [ 4. 10.  8.  2. 14. 10.  2.  8.]
 [ 0. 18. 10. 16.  0. 20. 20. 18.]
 [ 4. 12.  6. 16.  4.  8.  4. 12.]
 [ 8. 16. 12.  2.  6. 16.  2. 18.]
 [16. 18.  8.  2.  6. 12. 14.  4.]
 [ 0.  6.  2. 14.  6.  2. 10. 10.]]
Matriz original:

[[ 6.  3. 10.  7.  4.  6.  9.  2.]
 [ 6. 10. 10.  7.  4.  3.  7.  7.]
 [ 2.  5.  4.  1.  7.  5.  1.  4.]
 [ 0.  9.  5.  8.  0. 10. 10.  9.]
 [ 2.  6.  3.  8.  2.  4.  2.  6.]
 [ 4.  8.  6.  1.  3.  8.  1.  9.]
 [ 8.  9.  4.  1.  3.  6.  7.  2.]
 [ 0.  3.  1.  7.  3.  1.  5.  5.]]
```

CPU

```python
# Inicia el tiempo de medición para la operación con bucle en cpu
start_time_cpu = time.time()

# Duplica cada elemento de la matriz utilizando un bucle for
a_doubled_python = np.empty_like(a)
for i in range(a.shape[0]):
    for j in range(a.shape[1]):
        a_doubled_python[i, j] = a[i, j] * 2

# Detiene el tiempo de medición para la operación con bucle en Python
end_time_cpu = time.time()

# Calcula el tiempo total de la operación con bucle en Python
time_cpu= end_time_cpu - start_time_cpu

print("Tiempo de ejecución cpu: ", time_cpu)
```

```
Tiempo de ejecución cpu:  0.0015554428100585938
```

```python
np.array_equal(a_doubled_python,a_doubled)
```

```
True
```

Comparación GPU vs CPU

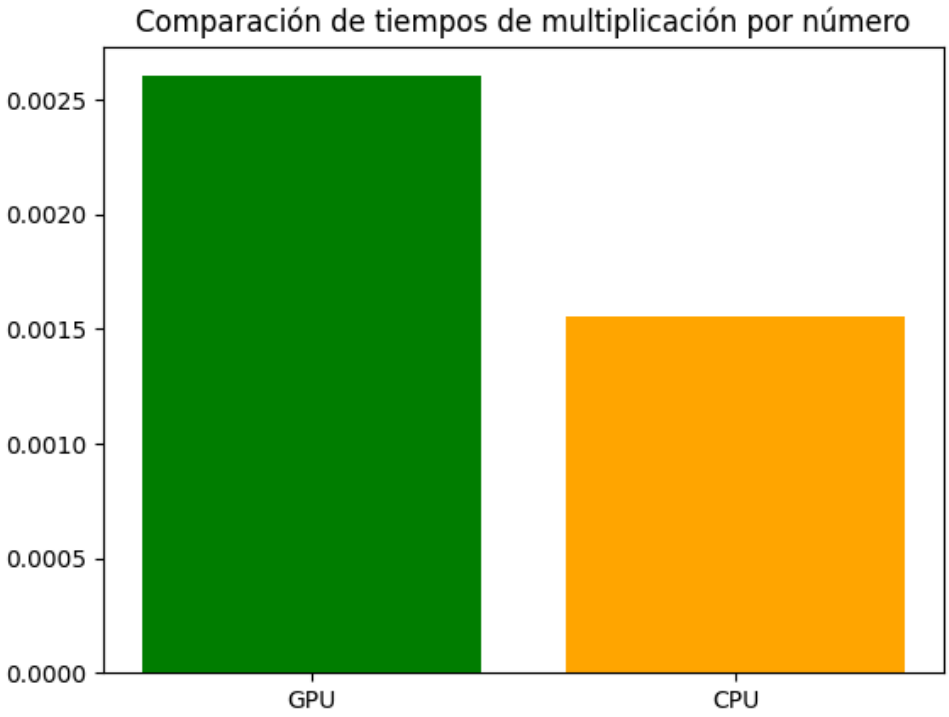
```python

```

```
# Nombres de las operaciones
labels = ['GPU', 'CPU']

# Valores de tiempo
values = [time_cuda, time_cpu]

# Graficar en barras con escala logarítmica en el eje y
plt.bar(labels, values, color=['green', 'orange'])
plt.title('Comparación de tiempos de multiplicación por número')
plt.show()
```



Suma de matrices

```
# Establece la semilla para reproducibilidad
np.random.seed(42)

# Crear dos matrices aleatorias
b = np.random.randint(0, 11, size=(128, 128)).astype(np.float32)
c = np.random.randint(0, 11, size=(128, 128)).astype(np.float32)

# Imprime las matrices originales
print("Matriz 'b':")
print(b)
print("\nMatriz 'c':")
print(c)
```

```
Matriz 'b':
[[ 6.  3. 10. ...  6.  6. 10.]
 [ 8.  9.  9. ...  0.  1.  0.]
 [ 4.  4. 10. ...  2.  8.  9.]
 ...
 [ 4.  4.  7. ...  9.  5.  6.]
 [ 8.  0.  3. ...  3.  9.  8.]
 [ 3.  3.  6. ...  0.  1.  5.]]

Matriz 'c':
[[ 8.  4.  3. ...  1.  7.  5.]
 [ 6.  0.  5. ...  9.  0.  7.]
 [ 0.  8.  5. ...  5.  0.  1.]
 ...
 [ 0.  1.  3. ...  6.  1.  9.]
 [ 1.  5.  8. ... 10.  1. 10.]
 [ 6.  0.  3. ...  8.  9.  4.]]
```

GPU

```
# Definición del módulo de CUDA que contiene la función del kernel
mod2 = SourceModule("""
  __global__ void add2(float *a, float *b)
  {
    int idx = (blockIdx.x * blockDim.x + threadIdx.x) + (blockIdx.y * blockDim.y + threadIdx.y
    a[idx] += b[idx];
  }
  """)

# Asignar memoria en la GPU para las matrices
b_gpu = drv.mem_alloc(b.nbytes)
c_gpu = drv.mem_alloc(c.nbytes)

# Copiar los datos de las matrices al dispositivo (GPU)
drv.memcpy_htod(b_gpu, b)
drv.memcpy_htod(c_gpu, c)

# Obtener la función del kernel compilado
func = mod2.get_function("add2")

# Iniciar el cronómetro
start_time = time.time()

# Llamar a la función del kernel con las matrices como argumentos
func(b_gpu, c_gpu, block=(32,32,1), grid=(4,4))

# Crear un array vacío del mismo tamaño que b para almacenar el resultado
added = np.empty_like(b)

# Copiar los resultados desde la memoria de la GPU al array creado
drv.memcpy_dtoh(added, b_gpu)

# Parar el cronómetro
end_time = time.time()

# Calcular la duración
duration1 = end_time - start_time

# Imprimir los resultados
print("Matriz B:\n")
print(b)
print("Matriz C:\n")
print(c)
print("Resultado suma:\n")
print(added)

# Imprimir el tiempo de ejecución
print("Tiempo de ejecución: {:.6f} segundos".format(duration1))
```

```
Matriz B:

[[ 6.  3. 10. ...  6.  6. 10.]
 [ 8.  9.  9. ...  0.  1.  0.]
 [ 4.  4. 10. ...  2.  8.  9.]
 ...
 [ 4.  4.  7. ...  9.  5.  6.]
 [ 8.  0.  3. ...  3.  9.  8.]
 [ 3.  3.  6. ...  0.  1.  5.]]
Matriz C:

[[ 8.  4.  3. ...  1.  7.  5.]
 [ 6.  0.  5. ...  9.  0.  7.]
 [ 0.  8.  5. ...  5.  0.  1.]
 ...
 [ 0.  1.  3. ...  6.  1.  9.]
 [ 1.  5.  8. ... 10.  1. 10.]
 [ 6.  0.  3. ...  8.  9.  4.]]
Resultado suma:

[[14.  7. 13. ...  7. 13. 15.]
 [14.  9. 14. ...  9.  1.  7.]
 [ 4. 12. 15. ...  7.  8. 10.]
 ...
 [ 4.  5. 10. ... 15.  6. 15.]
 [ 9.  5. 11. ... 13. 10. 18.]
 [ 9.  3.  9. ...  8. 10.  9.]]
Tiempo de ejecución: 0.000581 segundos
```

CPU

```python
# Iniciar el cronómetro
start_time = time.time()

# Crear un array vacío del mismo tamaño que b para almacenar el resultado
addedCPU = np.empty_like(b)

# Realizar la suma elemento a elemento
for i in range(b.shape[0]):
    for j in range(b.shape[1]):
        addedCPU[i, j] = b[i, j] + c[i, j]

# Parar el cronómetro
end_time = time.time()

# Calcular la duración
duration2 = end_time - start_time

# Imprimir los resultados
print("Matriz B:\n", b)
print("Matriz C:\n", c)
print("Resultado suma:\n", addedCPU)

# Imprimir el tiempo de ejecución
print("Tiempo de ejecución en CPU: {:.6f} segundos".format(duration2))
```

```
Matriz B:
 [[ 6.  3. 10. ...  6.  6. 10.]
 [ 8.  9.  9. ...  0.  1.  0.]
 [ 4.  4. 10. ...  2.  8.  9.]
 ...
 [ 4.  4.  7. ...  9.  5.  6.]
 [ 8.  0.  3. ...  3.  9.  8.]
 [ 3.  3.  6. ...  0.  1.  5.]]
Matriz C:
 [[ 8.  4.  3. ...  1.  7.  5.]
 [ 6.  0.  5. ...  9.  0.  7.]
 [ 0.  8.  5. ...  5.  0.  1.]
 ...
 [ 0.  1.  3. ...  6.  1.  9.]
 [ 1.  5.  8. ... 10.  1. 10.]
 [ 6.  0.  3. ...  8.  9.  4.]]
Resultado suma:
 [[14.  7. 13. ...  7. 13. 15.]
 [14.  9. 14. ...  9.  1.  7.]
 [ 4. 12. 15. ...  7.  8. 10.]
 ...
 [ 4.  5. 10. ... 15.  6. 15.]
 [ 9.  5. 11. ... 13. 10. 18.]
 [ 9.  3.  9. ...  8. 10.  9.]]
Tiempo de ejecución en CPU: 0.013484 segundos
```

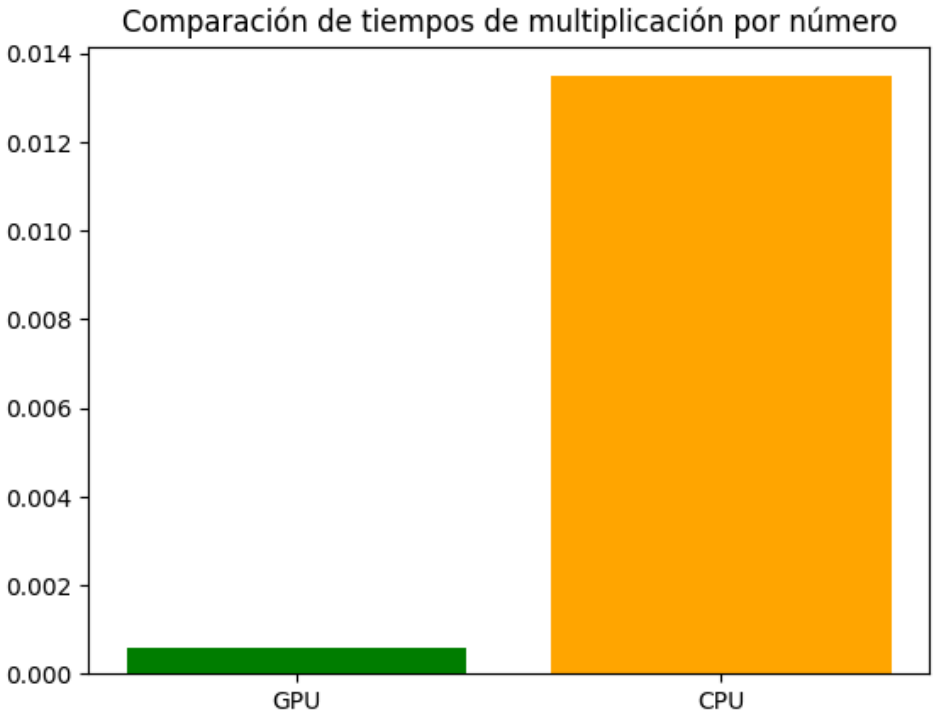
```
np.array_equal(added,addedCPU)
```

```
True
```

```
# Nombres de las operaciones
labels = ['GPU', 'CPU']

# Valores de tiempo
values = [duration1, duration2]

# Graficar en barras con escala logarítmica en el eje y
plt.bar(labels, values, color=['green', 'orange'])
plt.title('Comparación de tiempos de multiplicación por número')
plt.show()
```




```
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule

# Define el código del kernel de CUDA
kernel_code = """
__global__ void print_kernel()
{
    printf("Hello from block %d, thread %d\\n", blockIdx.x, threadIdx.x);
}
"""

# Compila el kernel
mod = SourceModule(kernel_code)

# Obtiene la función del kernel
print_kernel = mod.get_function("print_kernel")

# Lanza el kernel
print_kernel(block=(1, 1, 1), grid=(1, 1))

# Sincroniza para asegurar que se completa la ejecución del kernel
cuda.Context.synchronize()
```

```
Hello from block 0, thread 0
```

Stable_Diffusion_GPU_vs_CPU

Imprimir en PDF

Contenido

Stable Diffusion usando una GPU en Google Colaboratory con CUDA

Basado en Exploring Stable Diffusion in Google Colab using CUDA: A Step-by-Step Tutorial

Se utilizará la GPU, esta aceleración de GPU accesible a través de Google Colab mejorará significativamente la velocidad del proceso. Sin una GPU, ejecutar el modelo en una CPU podría tomar aproximadamente de 100 veces más tiempo que si se usa una GPU.

Recuerda activar un **Entorno de ejecución** con GPU.

Instalación de librerías

```
!pip install pycuda
!pip install diffusers
!pip install transformers
!pip install accelerate
```

```
Collecting pycuda
  Downloading pycuda-2023.1.tar.gz (1.7 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.7/1.7 MB 11.5 MB/s eta 0:00:00
?25h  Installing build dependencies ... ?25l?25hdone
  Getting requirements to build wheel ... ?25l?25hdone
  Preparing metadata (pyproject.toml) ... ?25l?25hdone
Collecting pytools>=2011.2 (from pycuda)
  Downloading pytools-2023.1.1-py2.py3-none-any.whl (70 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 70.6/70.6 kB 10.4 MB/s eta 0:00:0
?25hRequirement already satisfied: appdirs>=1.4.0 in /usr/local/lib/python3.10/d
Collecting mako (from pycuda)
  Downloading Mako-1.3.0-py3-none-any.whl (78 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 78.6/78.6 kB 12.2 MB/s eta 0:00:0
?25hRequirement already satisfied: platformdirs>=2.2.0 in /usr/local/lib/python3
Requirement already satisfied: typing-extensions>=4.0 in /usr/local/lib/python3
Requirement already satisfied: MarkupSafe>=0.9.2 in /usr/local/lib/python3.10/di
Building wheels for collected packages: pycuda
  Building wheel for pycuda (pyproject.toml) ... ?25l?25hdone
  Created wheel for pycuda: filename=pycuda-2023.1-cp310-cp310-linux_x86_64.whl
  Stored in directory: /root/.cache/pip/wheels/46/65/06/b997165edd2fd9690c3497ca
Successfully built pycuda
Installing collected packages: pytools, mako, pycuda
Successfully installed mako-1.3.0 pycuda-2023.1 pytools-2023.1.1
Collecting diffusers
  Downloading diffusers-0.23.1-py3-none-any.whl (1.7 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.7/1.7 MB 13.4 MB/s eta 0:00:00
?25hRequirement already satisfied: Pillow in /usr/local/lib/python3.10/dist-pacl
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: huggingface-hub>=0.13.2 in /usr/local/lib/python3
Requirement already satisfied: importlib-metadata in /usr/local/lib/python3.10/d
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.10/di
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: safetensors>=0.3.1 in /usr/local/lib/python3.10/d
Requirement already satisfied: fsspec>=2023.5.0 in /usr/local/lib/python3.10/dis
Requirement already satisfied: tqdm>=4.42.1 in /usr/local/lib/python3.10/dist-pa
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/pyth
Requirement already satisfied: packaging>=20.9 in /usr/local/lib/python3.10/dist
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.10/dist-packa
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-pa
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/d
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/d
Installing collected packages: diffusers
Successfully installed diffusers-0.23.1
Requirement already satisfied: transformers in /usr/local/lib/python3.10/dist-pa
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: huggingface-hub<1.0,>=0.16.4 in /usr/local/lib/py
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.10/di
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: tokenizers<0.19,>=0.14 in /usr/local/lib/python3
```

```
Requirement already satisfied: safetensors>=0.3.1 in /usr/local/lib/python3.10/d
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.10/dist-pacl
Requirement already satisfied: fsspec>=2023.5.0 in /usr/local/lib/python3.10/dis
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/pytl
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/pythor
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-pa
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/d
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/d
Collecting accelerate
  Downloading accelerate-0.24.1-py3-none-any.whl (261 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 261.4/261.4 kB 5.6 MB/s eta 0:00:
?25hRequirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.10/dist
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist
Requirement already satisfied: psutil in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: pyyaml in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: torch>=1.10.0 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: huggingface-hub in /usr/local/lib/python3.10/dist
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.10/di
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: triton==2.1.0 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: tqdm>=4.42.1 in /usr/local/lib/python3.10/dist-pa
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/pythor
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-pa
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/d
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/d
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.10/dist-pa
Installing collected packages: accelerate
Successfully installed accelerate-0.24.1
```

Carga de librerías

```
# Importando la biblioteca PyTorch para construir y entrenar redes neuronales
import torch

# Importando StableDiffusionPipeline para utilizar modelos preentrenados de Sta
from diffusers import StableDiffusionPipeline

# Image es una clase del módulo PIL para visualizar imágenes en un cuaderno de
from PIL import Image

# Para medir tiempos
import time

# Mostrar imágenes
import matplotlib.pyplot as plt
```

```
The cache for model files in Transformers v4.22.0 has been updated. Migrating y
```

Verificación de GPU y CUDA

```
import pycuda.driver as drv
import pycuda.autoinit
drv.init()
print("%d device(s) found." % drv.Device.count())
for i in range(drv.Device.count()):
  dev = drv.Device(i)
  print(" Device #%d: %s" % (i, dev.name()))
  print(" Compute Capability: %d.%d" % dev.compute_capability())
  print(" Total Memory: %s GB" % (dev.total_memory() // (1024 * 1024 * 1024)))
```

```
1 device(s) found.
 Device #0: Tesla T4
 Compute Capability: 7.5
 Total Memory: 14 GB
```

```
def generate_image_grid(text_prompt, n_images=4, rows=2, cols=2, output_size=(2
    prompt = [text_prompt] * n_images
    images = pipeline(prompt).images

    # Verifica si la cuadrícula tiene un solo elemento
    if n_images == 1:
        return images[0]

    # Muestra las imágenes por separado con matplotlib
    fig, axs = plt.subplots(rows, cols, figsize=(8, 8))

    for i, ax in enumerate(axs.flat):
        # Si hay menos imágenes que elementos en la cuadrícula, oculta los ejes
        if i < len(images):
            ax.imshow(images[i])
            ax.axis('off')
        else:
            ax.axis('off')

    plt.show()
```

Stable Diffusion en GPU

```
# Creating the pipeline for GPU
pipeline = StableDiffusionPipeline.from_pretrained("CompVis/stable-diffusion-v1
# Moving pipeline to GPU
pipeline = pipeline.to('cuda')
```

```
`text_config_dict` is provided which will be used to initialize `CLIPTextConfig
`text_config_dict` is provided which will be used to initialize `CLIPTextConfig
`text_config_dict` is provided which will be used to initialize `CLIPTextConfig
```

```
prompt = 'a cute magical flying german shepherd puppy wearing a superman cape,
fantasy art drawn by disney concept artists, golden colour, high quality, \
highly detailed, elegant, sharp focus, concept art, character concepts, \
digital painting, mystery, adventure'

start_time = time.time()

generate_image_grid(text_prompt=prompt, n_images=2, rows=1, cols=2, output_size

end_time = time.time()
elapsed_timeGPU = end_time - start_time
print(f"Tiempo transcurrido: {round(elapsed_timeGPU, 2)} segundos")
print(f"Tiempo transcurrido: {round(elapsed_timeGPU / 60, 2)} minutos")
```

```
Tiempo transcurrido: 21.2 segundos
Tiempo transcurrido: 0.35 minutos
```

Stable Diffusion en CPU

```
# Creating the pipeline for CPU
pipeline = StableDiffusionPipeline.from_pretrained("CompVis/stable-diffusion-v1
```

```
`text_config_dict` is provided which will be used to initialize `CLIPTextConfig
`text_config_dict` is provided which will be used to initialize `CLIPTextConfig
`text_config_dict` is provided which will be used to initialize `CLIPTextConfig
```

```
prompt = 'a cute magical flying german shepherd puppy wearing a superman cape,
fantasy art drawn by disney concept artists, golden colour, high quality, \
highly detailed, elegant, sharp focus, concept art, character concepts, \
digital painting, mystery, adventure'

start_time = time.time()

generate_image_grid(text_prompt=prompt, n_images = 2,  rows=1, cols = 2, output

end_time = time.time()
elapsed_timeCPU = end_time - start_time
print(f"Tiempo transcurrido: {round(elapsed_timeCPU, 2)} segundos")
print(f"Tiempo transcurrido: {round(elapsed_timeCPU / 60, 2)} minutos")
```

```
Tiempo transcurrido: 2182.16 segundos
Tiempo transcurrido: 36.37 minutos
```

Comparemos la GPU con la CPU

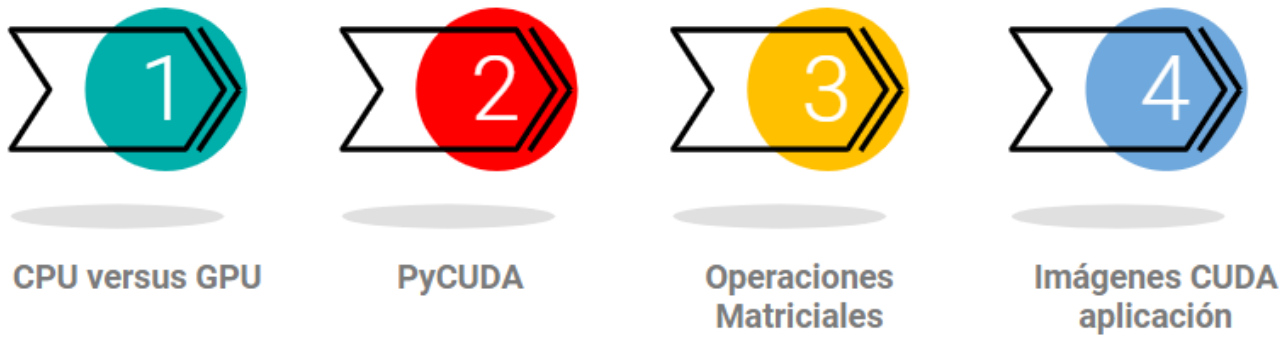
```
speedup_factor = elapsed_timeCPU / elapsed_timeGPU
print(f"La ejecución en GPU fue aproximadamente {speedup_factor:.2f} veces más
```

```
La ejecución en GPU fue aproximadamente 102.91 veces más rápida que en CPU.
```

Unidad 11: PyCUDA

Contenido

Contenido de la unidad



Comparemos a las CPU contra las GPU

CPU

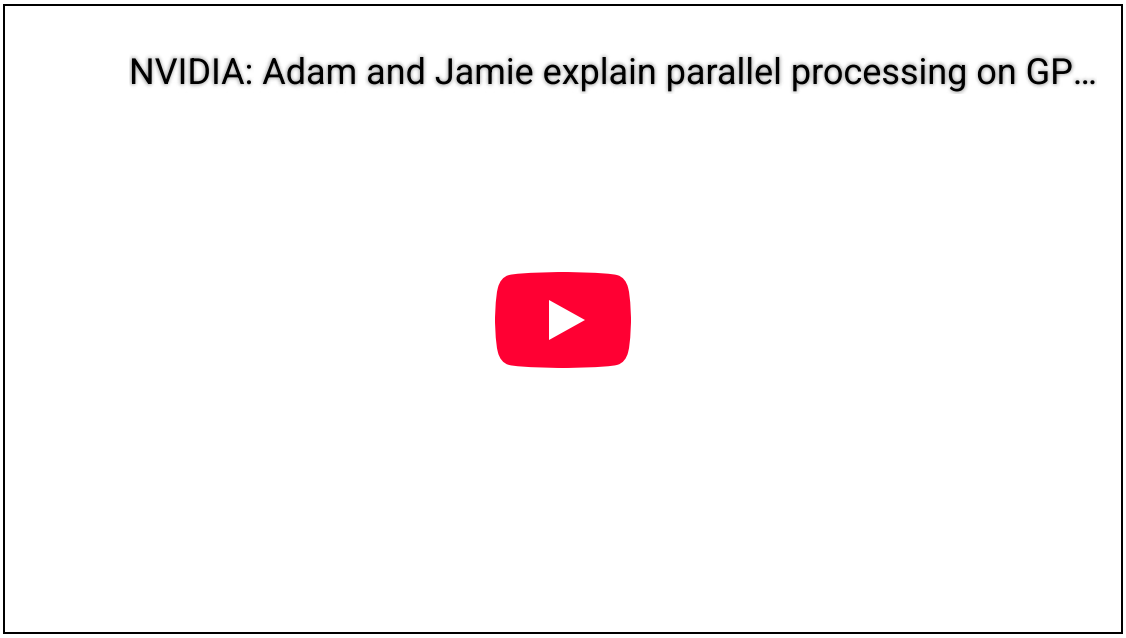
- Varios núcleos, pero generalmente menos que una

GPU

- Muchos núcleos para procesamiento en paralelo.

GPU.
- Baja latencia.
- Bueno para procesamiento en serie.
- Puede realizar un puñado de operaciones a la vez.

- Alto rendimiento.
- Bueno para procesamiento en paralelo.
- Puede realizar miles de operaciones a la vez.



Truco

- Leer un blog de Nvidia comparando GPU con las CPU

PyCUDA

PyCUDA ofrece un acceso sencillo y pythonico a la API de computación paralela Compute Unified Device Architecture (CUDA), de Nvidia. Aunque ya existen varios envoltorios para la API de CUDA, ¿por qué elegir PyCUDA?

Gestión automática de objetos vinculada a su ciclo de vida. Este enfoque, simplifica significativamente la escritura de código que es correcto, libre de fugas y resistente a caídas.

PyCUDA también maneja las dependencias, lo que significa que no se desvinculará de un contexto antes de liberar toda la memoria asignada en él.

Completitud. PyCUDA pone a tu disposición toda la potencia de la API de controladores de CUDA, si así lo deseas.

Verificación automática de errores. Todos los errores de CUDA se traducen automáticamente en excepciones de Python.

Velocidad. Dado que la capa base de PyCUDA está escrita en C++, todas las ventajas mencionadas anteriormente prácticamente no tienen coste adicional.

> **Truco**
>
> Ver el siguiente recurso:
>
> PyCUDA Tutorial

Usar la GPU para multiplicar y sumar elementos

> **Truco**
>
> Ver el Notebook: `PyCUDA_multiplicar_sumar_elementos.ipynb`
>
> O acceder a el desde PyCUDA_multiplicar_sumar_elementos_working.ipynb

Usar la GPU para multiplicar matrices

> **Truco**
>
> Ver el Notebook: `PyCUDA_multiplicar_matrices.ipynb`
>
> O acceder a el desde PyCUDA_multiplicar_matrices_working.ipynb

Comparativa en la creación de imágenes desde textos en GPU y en CPU

Truco

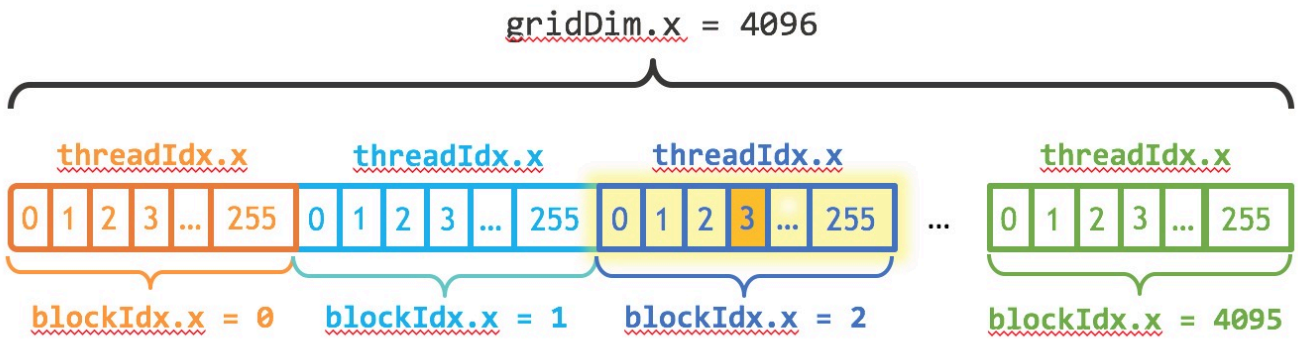
Ver el Notebook: `Stable_Diffusion_GPU_vs_CPU.ipynb`

O acceder a el desde Stable_Diffusion_GPU_vs_CPU_working.ipynb

Comprendamos la indexación

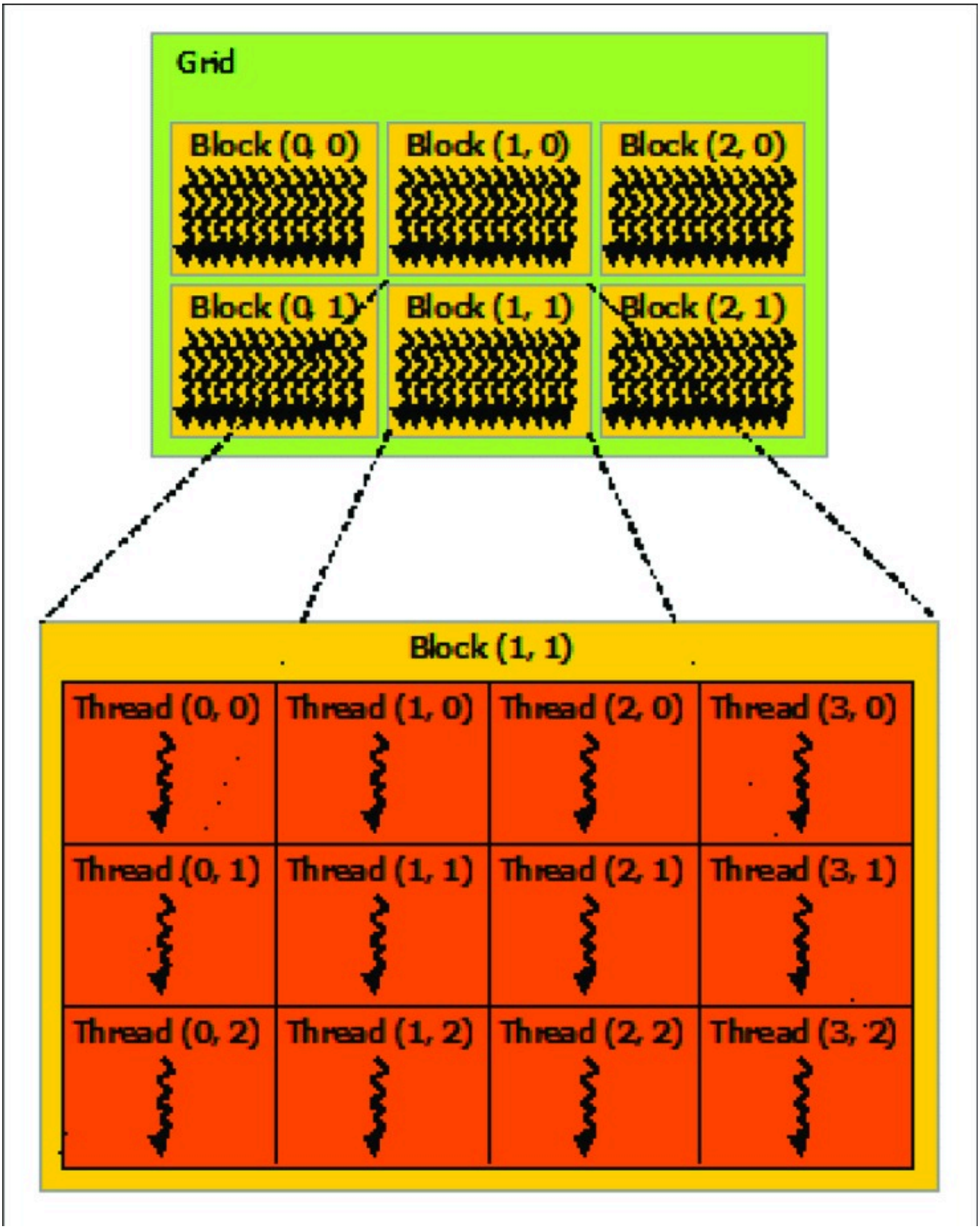
Truco

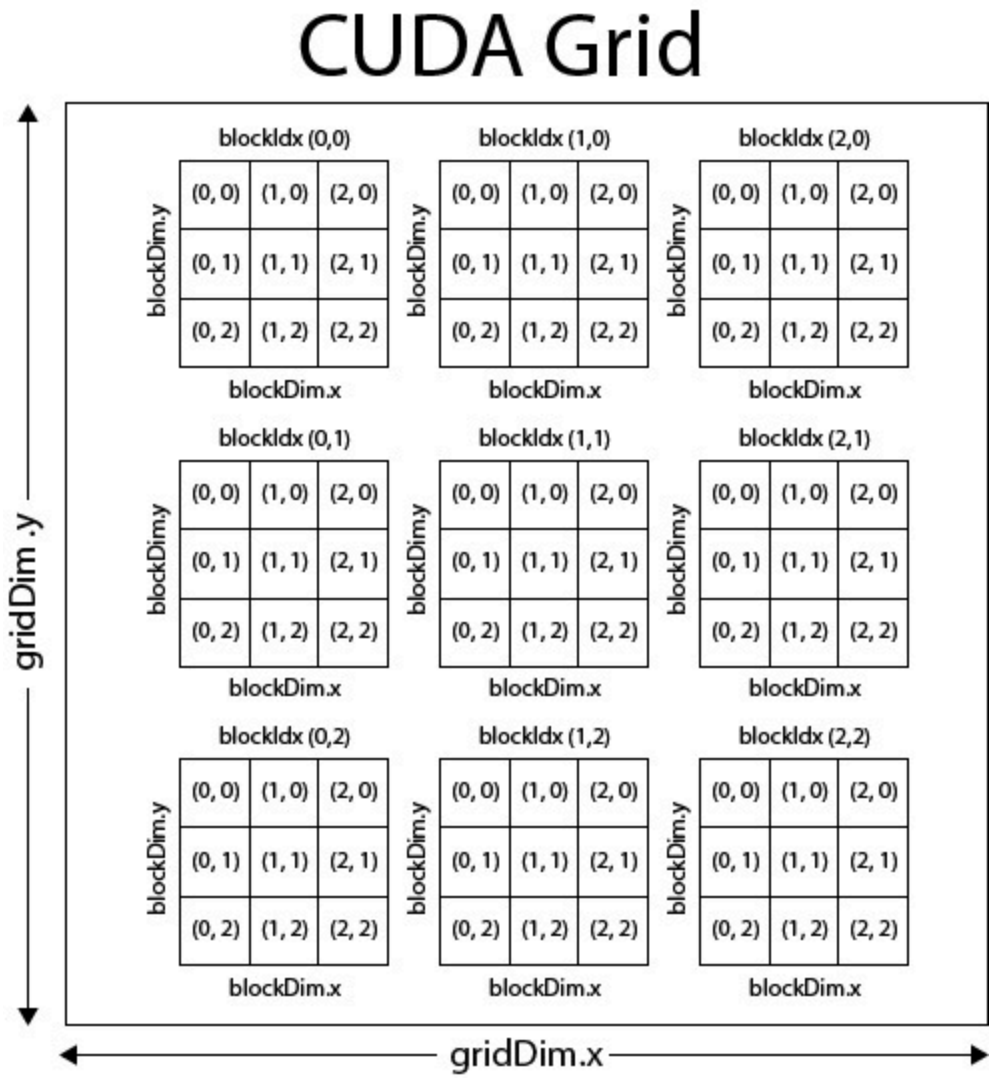
Acceder a el desde https://anuradha-15.medium.com/cuda-thread-indexing-fb9910cba084



index = blockIdx.x * blockDim.x + threadIdx.x

index = (2) * (256) + (3) = 515


Grid
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Block (1, 1)
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)


CUDA Grid
blockIdx (0,0)
blockIdx (1,0)
blockIdx (2,0)
blockIdx (0,1)
blockIdx (1,1)
blockIdx (2,1)
blockIdx (0,2)
blockIdx (1,2)
blockIdx (2,2)
(0, 0) (1, 0) (2, 0)
(0, 1) (1, 1) (2, 1)
(0, 2) (1, 2) (2, 2)
blockDim.y
blockDim.x
gridDim .y
gridDim.x