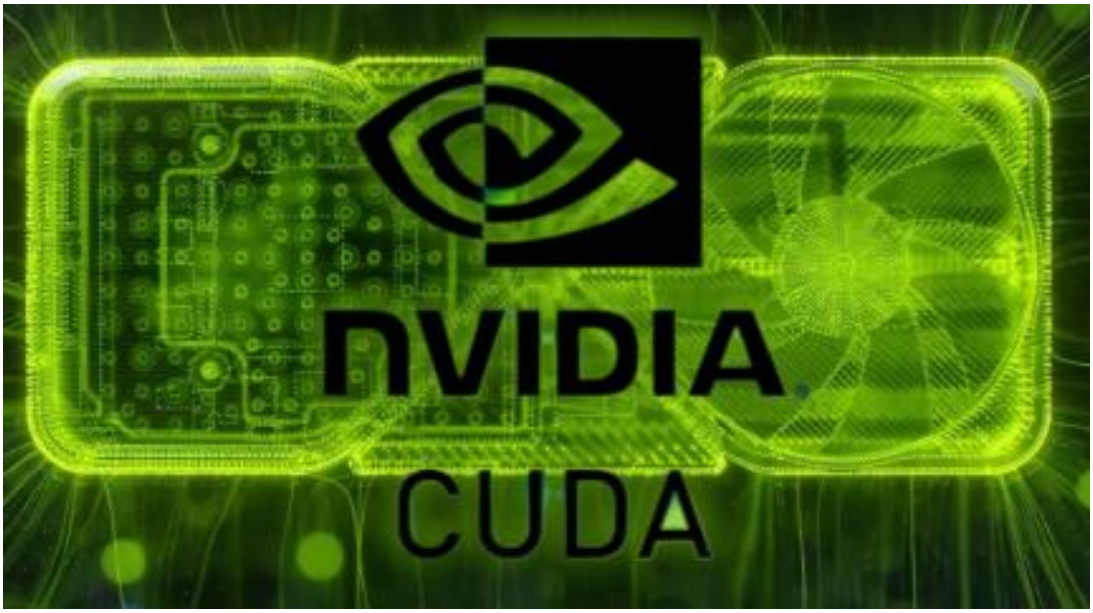
Talento Altamente Especializado – Inteligencia Artificial 2025 COCYTEN-Nayarit

Advanced IA: ECU's CUDA Introduction



Student: Cesar Eduardo Inda Ceniceros

Professor: Dr. German

Assignment Date: 15/11/2025

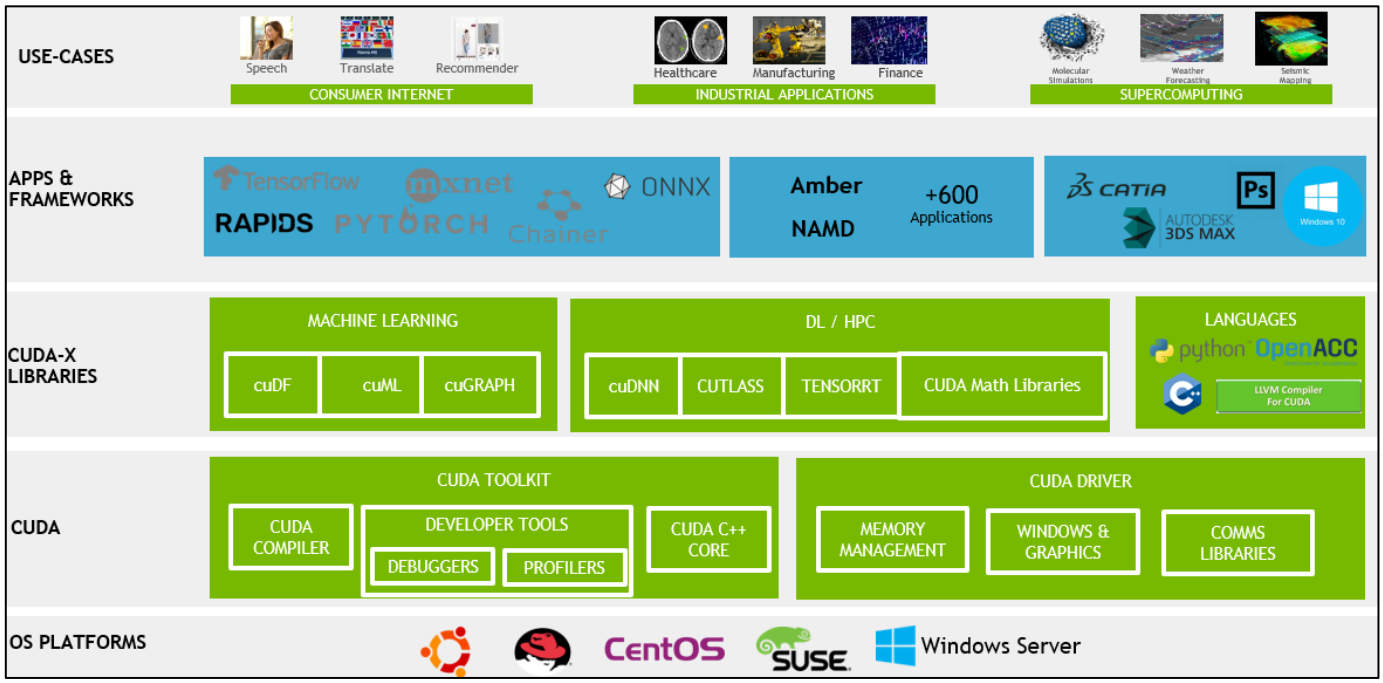
Submission Date: 17/11/2025

1.- Introducción

En esta parte del curso mas avanzado comenzamos aprendiendo acerca de conceptos mas complejos relacionados con la parte de lo que es **“CUDA”** y como es utilizado dentro de la parte de la inteligencia artificial, esto de la mano del Dr. German, quien trabaja en la empresa NVIDIA y tenemos la posibilidad de conocer una parte de como esto es de gran impacto y relevancia en la industria sobre todo en lo correspondiente a el área de desarrollo de inteligencia artificial y el desarrollo de lo que es la GPU.

CUDA (Compute Unified Device Architectures) es una plataforma y modelo de programación creada por NVIDIA que permite a los desarrolladores usar directamente la GPU para ejecutar operaciones en paralelo. A diferencia de la CPU, que trabaja con pocos núcleos optimizados para tareas secuenciales, **una GPU puede ejecutar miles de hilos simultáneamente, lo que acelera enormemente cálculos matemáticos.**

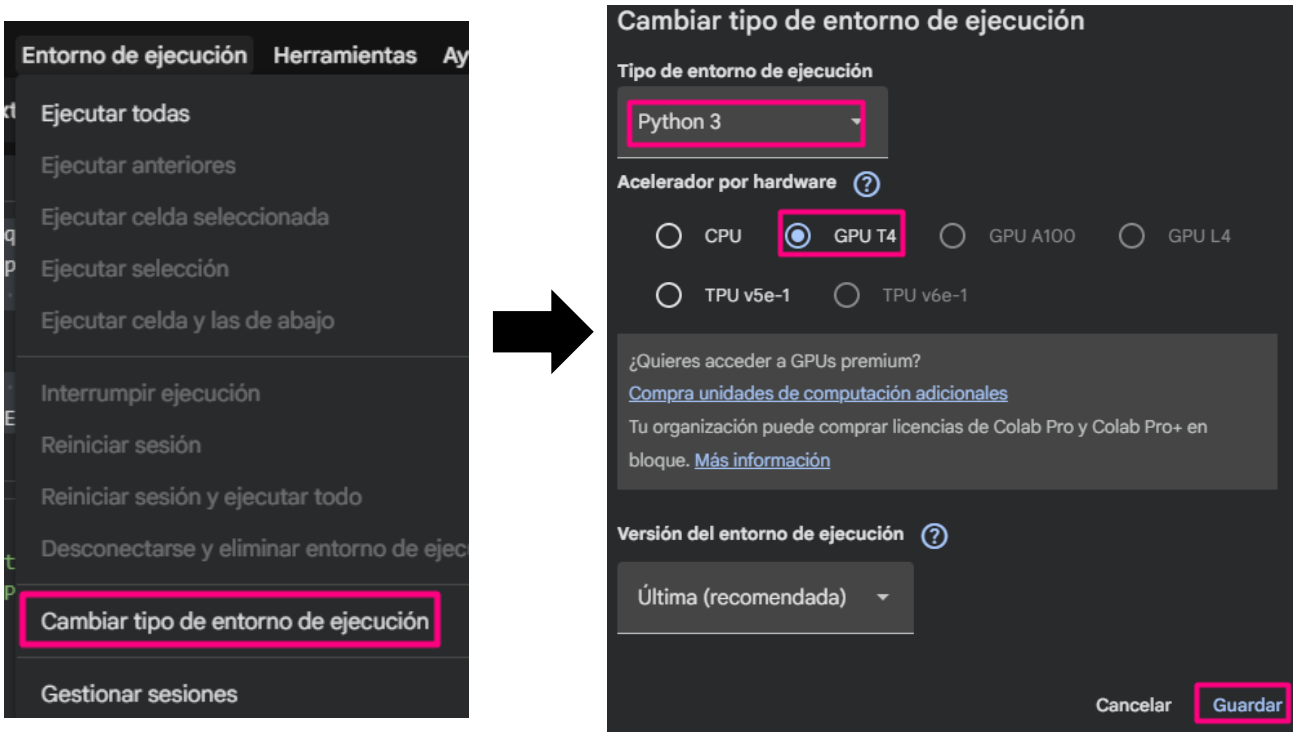
En el desarrollo de Inteligencia Artificial, CUDA ha sido fundamental porque las redes neuronales profundas requieren millones de operaciones repetitivas principalmente multiplicaciones de matrices que son ideales para ser ejecutadas en paralelo. Gracias a CUDA, frameworks como TensorFlow, PyTorch y cuDNN pueden aprovechar la potencia de la GPU para entrenar modelos mucho más rápido. En este sentido entonces “CUDA” es la base que permite que la IA moderna funcione a gran escala, acelerando desde el entrenamiento de modelos de DeepLearning hasta la inferencia en tiempo real para visión computacional, procesamiento de lenguaje natural y sistemas autónomos.



2.- Google Colab:

Como parte de este curso estuvimos realizando los laboratorios y elementos de practica en lo que fuera la herramienta colaborativa de **“Google Colab”.**

Para esta parte tenemos que considerar un punto importante cambiar el **“entorno de ejecución”** a algo que es **“GPU-T4”** es una computadora virtual que Google presta para correr temporalmente el codigo en sentido de: **CPU, RAM, Disco temporal, (Opcional) GPU y (Opcional) TPU**



Este entorno se crea desde cero al abrir o reiniciar una **“notebook”** en este caso para trabajar con lo que es **“CUDA”** necesitamos establecerlo como **“GPU-T4”**, es decir la **GPU NVIDIA Tesla T4**, es este sentido se pueden usar **2560 CUDA cores** (moles de hilos paralelos) y Tensor Cores (aceleran operaciones de DeepLearning).

ECU1:

Como primera parte establecemos esto en la sección superior:

```
!uv pip install -q --system numba-cuda==0.4.0
import numpy as np
from numba import cuda
import time
import os
from numba import config
config.CUDA_ENABLE_PYNVJITLINK = 1
```

Aquí se le indica a “colab” que ejecute un comando del sistema operativo, se usa “UV” es un instalador y gestos de paquetes más rápido que pip con la parte de –System le indicamos que instale el paquete en el entorno del sistema de colab, no en entorno virtual. Al importar los cuda de numba estamos importando el módulo CUD de numba para crear los kernel’s GPU con @cuda.jit y de estar forma hacer uso de la memoria enviando y recibiendo datos apartir del uso de GPU → ← CPU, algo muy importante también es lo siguiente:

config.CUDA_ENABLE_PYNVJITLINK = 1

esto le permite a numba compilar y enlazar los kernel’s de CUDA, esto habilitando un método de enlazamiento de los kernel’s de CUDA.

```
# 1. CUDA kernel Device
@cuda.jit
def first_kernel(a, result):
    idx = cuda.grid(1)
    if idx < a.size:
        result[idx] = a[idx]
```

Esta función se ejecuta en la GPU, idx = cuda.grid(1) cada hilo de la GPU recibe un índice global único en ese sentido si el hilo esta fuera del arreglo, no hará nada, cada hilo copia un solo elemento. Esto genera que los hilos trabajaen en paralelo.

```python
# Host
def main():
    # 2. Initialize data on CPU
    N = 10_000_000
    a_cpu = np.arange(N, dtype=np.float32)

    # ----------------------------
    # CPU computation
    # ----------------------------
    start = time.time()
    result_cpu = a_cpu
    cpu_time = time.time() - start
    print(f"CPU time: {cpu_time * 1e3:.2f} ms")

    # ----------------------------
    # GPU computation
    # ----------------------------
    start = time.time()
    a_gpu = cuda.to_device(a_cpu)
    result_gpu = cuda.device_array_like(a_cpu)
    transfer_in_time = time.time() - start

    threads_per_block = 128
    blocks_per_grid = (N + threads_per_block - 1) // threads_per_block

    start = time.time()
    first_kernel[blocks_per_grid, threads_per_block](a_gpu, result_gpu)
    cuda.synchronize()
    kernel_time = time.time() - start

    start = time.time()
    result_from_gpu = result_gpu.copy_to_host()
    transfer_out_time = time.time() - start
```

Esta parte principal crea un arreglo con 10 millones de numeros que se encuentran en la RAM. Aquí se copia la referencia, así que el tiempo es prácticamente o milisegundos. En esto se copia el arreglo a la memoria global de la GPU y se crea un arreglo en GPU del mismo tamaño que la original. Se configura el grid del bloque, al igual que se lanza la ejecución del kernel, como parte final se realiza la transferencia de la GPU hacia la CPU y en el sentido del arreglo y también se obtiene en razón de tiempo cuanto fue de dicha transferencia.

```
    # REPORT
    print(f"Transfer to GPU time: {transfer_in_time * 1e3:.2f} ms")
    print(f"Kernel execution time: {kernel_time * 1e3:.2f} ms")
    print(f"GPU transfer to host: {transfer_out_time * 1e3:.2f} ms")
    print(f"Total GPU Time: {(transfer_in_time + kernel_time + transfer_out_time) * 1e3:.2f} ms")

    # Verificar que CPU y GPU coinciden
    print("¿Coinciden CPU y GPU?:", np.allclose(result_cpu, result_from_gpu))

    # Limpieza básica (opcional)
    del a_gpu, result_gpu
    cuda.close()

if __name__ == "__main__":
    main()
```

Como parte final se obtiene un reporte de dichos tiempos. Se comparan los tiempos también entre la CPU y GPU, en la parte de “del” se realiza una limpieza de memoria en GPU cerrando el contexto de la parte de CUDA.

```
CPU time: 0.00 ms
Transfer to GPU time: 94.76 ms
Kernel execution time: 53.48 ms
GPU transfer to host: 15.28 ms
Total GPU Time: 163.52 ms
¿Coinciden CPU y GPU?: True
```

ECU2:

Lo realizado en este segundo laboratorio también requirió establecer los elementos necesarios para simular el comportamiento de la parte de CUDA apartir de lo siguiente:

```
!uv pip install -q --system numba-cuda==0.4.0
import numpy as np

import time
import os

# Enable the CUDA simulator. This MUST be set BEFORE numba imports or kernel definitions.
os.environ["NUMBA_ENABLE_CUDASIM"] = "1"
from numba import cuda
from numba import config

# --- Configuration & Data Preparation ---

config.CUDA_ENABLE_PYNVJITLINK = 1
```

Esta consideración es importante porque aquí se agrega un elemento importante que es

```
# Enable the CUDA simulator. This MUST be set BEFORE numba imports or kernel definitions.
os.environ["NUMBA_ENABLE_CUDASIM"] = "1"
```

Con ello **os.environ** es un diccionario que contiene las variables de entornos del sistema operativo, es decir fuerza a CUDA que simule en CPU. En la siguiente sección

```
# ------------------------------------------------------------
# Prepare character data (ASCII values for A-H, 8 characters total)
# ------------------------------------------------------------
characters = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
data = np.array([ord(c) for c in characters], dtype=np.uint8)
data_size = len(data)  # 8 elements

# ------------------------------------------------------------
# 1D Kernel Definition (using gidx, bidx, tidx)
# ------------------------------------------------------------
@cuda.jit
def kernel_1d_dims(arr):
    # gidx: Global 1D index (Thread ID in the entire grid)
    gidx = cuda.grid(1)

    # bidx: Block ID (Block index in the grid)
    bidx = cuda.blockIdx.x

    # tidx: Thread ID (Thread index within the block)
    tidx = cuda.threadIdx.x

    if gidx < arr.size:
        # Standard Python print works via the simulator or in CUDA console
        print(f"BID: {bidx}, TID: {tidx}, GID: {gidx}, Char: {chr(arr[gidx])}")

 # Example 1: Block, 8 Threads per Block
blocks_per_grid_ex1 = 1
thread_per_bloc_ex1 = 8
#Total threads = 1 * 8 = 8

kernel_1d_dims[blocks_per_grid_ex1, thread_per_bloc_ex1](data)
cuda.synchronize()
```

Aquí en este ejemplo se hace una conversión de letras en codigo ASCII y al final se crea un arreglo de 8 bits sin signo con dichos valores. También se calcula el índice global del hilo en primera dimension.

Otra consideración es obtener el ID de los bloques y la consideración de que solo los hilos con índices validos trabajaran se obtiene la impresión de los ID de los bloques e índices, en la otra sección se configuran los bloques e hilos apartir de las consideraciones de hilos por bloque, se lanza posteriormente el kernel es decir en ese sentido cada hilo ejecutara el codigo del kernel de dicha dimension y se sincronizaran esperando a que todos los hilos terminen antes de seguir. Se obtiene el siguiente comportamiento:

1. **BID:** ID del bloque
2. **TID:** ID del hilo dentro del bloque
3. **GID:** índice global
4. **Char:** la letra correspondiente a arr[gidx], usando chr() para convertir el código ASCII a carácter.

```
BID: 0, TID: 0, GID: 0, Char: A
BID: 0, TID: 1, GID: 1, Char: B
BID: 0, TID: 2, GID: 2, Char: C
BID: 0, TID: 3, GID: 3, Char: D
BID: 0, TID: 4, GID: 4, Char: E
BID: 0, TID: 5, GID: 5, Char: F
BID: 0, TID: 6, GID: 6, Char: G
BID: 0, TID: 7, GID: 7, Char: H
```

ECU2_1:

También se realizó el siguiente programa partiendo del mismo encabezo para CUDA.

```
@cuda.jit
def whoami():
    # Compute block id in a 3D grid
    block_id = (
        cuda.blockIdx.x +
        cuda.blockIdx.y * cuda.gridDim.x +
        cuda.gridDim.x * cuda.gridDim.y
    )

    # Threads per block
    threads_per_block = (
        cuda.blockDim.x * cuda.blockDim.y
    )
```

```
# Offset of this block
block_offset = block_id * threads_per_block

# Compute thread id inside block
thread_offset = (
    cuda.threadIdx.x +
    cuda.threadIdx.y * cuda.blockDim.x +
    cuda.blockDim.x * cuda.blockDim.y
)

# Global thread id across all blocks
global_id = block_offset + thread_offset
```

Aquí se está generando una indexación de bloques 2D e hilos 2D. Apartir de lo que es el mapeo de los ID's de bloques e hilos asi como también la dimensión de los grids y bloques, correspondientes a un índice lineal de identificadores globales. Es decir, tomando un grid del bloque y convirtiéndolo a un número único del bloque apartir de su identificador.

```
print(f"{global_id:03d} | Block[x, y]({cuda.blockIdx.x} {cuda.blockIdx.y}) = {block_id:3d} | "
      f"Thread[x, y] ({cuda.threadIdx.x} {cuda.threadIdx.y} ) = {thread_offset:3d} BlockDim.x {cuda.blockDim.x} BlockDim.y {cuda.blockDim.y} GridDim.x {cuda.gridDim.x} GridDim.y {cuda.gridDim.y}"
```

Se obtiene una impresión de la referencia de los elementos mencionados.

```
b_x, b_y = 2, 2
t_x, t_y = 4, 1

blocks_per_grid = (b_x, b_y)
threads_per_block = (t_x, t_y)

total_blocks = b_x * b_y
total_threads = t_x * t_y
print(f"{total_blocks} blocks/grid")
print(f"{total_threads} threads/block")
print(f"{total_blocks * total_threads} total threads\n")

# Launch kernel
whoami[blocks_per_grid, threads_per_block]()

# Wait for GPU to finish (like cudaDeviceSynchronize)
cuda.synchronize()
```

Aquí una vez obtenida esa relación se imprimen los elementos de cada una de esas partes y por último una vez que se lanza el kernel de los grids multiplicados por los bloques y la referencia final de cuantos hilos se obtienen.

```
4 blocks/grid
4 threads/block
16 total threads

020 | Block[x, y](0 0) =   4 | Thread[x, y] (0 0 ) =   4 BlockDim.x 4 BlockDim.y 1 GridDim.x 2 GridDim.y 2
021 | Block[x, y](0 0) =   4 | Thread[x, y] (1 0 ) =   5 BlockDim.x 4 BlockDim.y 1 GridDim.x 2 GridDim.y 2
022 | Block[x, y](0 0) =   4 | Thread[x, y] (2 0 ) =   6 BlockDim.x 4 BlockDim.y 1 GridDim.x 2 GridDim.y 2
023 | Block[x, y](0 0) =   4 | Thread[x, y] (3 0 ) =   7 BlockDim.x 4 BlockDim.y 1 GridDim.x 2 GridDim.y 2
028 | Block[x, y](0 1) =   6 | Thread[x, y] (0 0 ) =   4 BlockDim.x 4 BlockDim.y 1 GridDim.x 2 GridDim.y 2
029 | Block[x, y](0 1) =   6 | Thread[x, y] (1 0 ) =   5 BlockDim.x 4 BlockDim.y 1 GridDim.x 2 GridDim.y 2
030 | Block[x, y](0 1) =   6 | Thread[x, y] (2 0 ) =   6 BlockDim.x 4 BlockDim.y 1 GridDim.x 2 GridDim.y 2
031 | Block[x, y](0 1) =   6 | Thread[x, y] (3 0 ) =   7 BlockDim.x 4 BlockDim.y 1 GridDim.x 2 GridDim.y 2
024 | Block[x, y](1 0) =   5 | Thread[x, y] (0 0 ) =   4 BlockDim.x 4 BlockDim.y 1 GridDim.x 2 GridDim.y 2
025 | Block[x, y](1 0) =   5 | Thread[x, y] (1 0 ) =   5 BlockDim.x 4 BlockDim.y 1 GridDim.x 2 GridDim.y 2
026 | Block[x, y](1 0) =   5 | Thread[x, y] (2 0 ) =   6 BlockDim.x 4 BlockDim.y 1 GridDim.x 2 GridDim.y 2
027 | Block[x, y](1 0) =   5 | Thread[x, y] (3 0 ) =   7 BlockDim.x 4 BlockDim.y 1 GridDim.x 2 GridDim.y 2
032 | Block[x, y](1 1) =   7 | Thread[x, y] (0 0 ) =   4 BlockDim.x 4 BlockDim.y 1 GridDim.x 2 GridDim.y 2
033 | Block[x, y](1 1) =   7 | Thread[x, y] (1 0 ) =   5 BlockDim.x 4 BlockDim.y 1 GridDim.x 2 GridDim.y 2
034 | Block[x, y](1 1) =   7 | Thread[x, y] (2 0 ) =   6 BlockDim.x 4 BlockDim.y 1 GridDim.x 2 GridDim.y 2
035 | Block[x, y](1 1) =   7 | Thread[x, y] (3 0 ) =   7 BlockDim.x 4 BlockDim.y 1 GridDim.x 2 GridDim.y 2
```

ECU3:

Para estas prácticas se requirió la siguiente consideración definirla en la parte de configuración en la parte superior, se agrego otro elemento que es instalar el backend de NVIDIA para JIT linking.

```
# @title
!uv pip install -q --system numba-cuda==0.4.0
!pip install pynvjitlink-cu12

import numpy as np
from numba import cuda
import time
import os
from numba import config
import numba
config.CUDA_ENABLE_PYNVJITLINK=1
```

```
Collecting pynvjitlink-cu12
  Downloading pynvjitlink_cu12-0.7.0-cp312-cp312-manylinux_2_27_x86_64.manylinux_2_28_x86_64.whl.metadata (1.5 kB)
Downloading pynvjitlink_cu12-0.7.0-cp312-cp312-manylinux_2_27_x86_64.manylinux_2_28_x86_64.whl (46.9 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 46.9/46.9 MB 13.7 MB/s eta 0:00:00
Installing collected packages: pynvjitlink-cu12
Successfully installed pynvjitlink-cu12-0.7.0
```

En esta parte se esta generando 2 vectores grandes “a” y “b” con 10 millones de números aleatorios se reserva memoria en GPU y se copia esos vectores a una sección, se calienta un poco una etapa con el **“warmup”** y despues se ejecuta el kernel con un tiempo que se evalúa en cuanto se suman los vectores, se hace el mismo proceso en CPU con Numpy y se evalúa el tiempo, al final se evalúan los tiempos de ambos y se verifican los resultados.

```
# @title
# ex1_vector_add.py
import numpy as np
from numba import cuda
import math
import time

@cuda.jit
def vector_add_kernel(a, b, c):
    """
    Each thread computes one element: c[i] = a[i] + b[i]
    """

    # Compute global thread index
    idx = cuda.grid(1)

    # Boundary check
    if idx < c.size:
        c[idx] = a[idx] + b[idx]
```

```
def main():
    N_large = 10_000_000

    a = np.random.randn(N_large).astype(np.float32)
    b = np.random.randn(N_large).astype(np.float32)
    c = np.zeros(N_large, dtype=np.float32)

    d_a = cuda.to_device(a)
    d_b = cuda.to_device(b)
    d_c = cuda.to_device(c)

    threads_per_block = 256
    blocks_per_grid = math.ceil(N_large / threads_per_block)

    # Warmup
    vector_add_kernel[blocks_per_grid, threads_per_block](d_a, d_b, d_c)
    cuda.synchronize()
```

```
    # Run timed kernel
    start = time.time()
    vector_add_kernel[blocks_per_grid, threads_per_block](d_a, d_b, d_c)
    cuda.synchronize()
    gpu_time = (time.time() - start) * 1000

    print(f"GPU time: {gpu_time:.2f} ms")

    # Copy result to CPU
    result = d_c.copy_to_host()

    # CPU timing
    cpu_start = time.time()
    expected =  a + b
    cpu_time = (time.time() - cpu_start) * 1000
```

```
    print(f"GPU Kernel Time: {gpu_time:.3f} ms")
    print(f"CPU Numpy Time: {cpu_time:.3f} ms")
    print(f"Speedup:{cpu_time / gpu_time:.2f}x")
    print(f"Correct:", np.allclose(result, expected))

"""
    # Optional: validate correctness
    if np.allclose(result, a + b):
        print("Vector add correct!")
    else:
        print("Error in GPU computation.")
"""

if __name__ == "__main__":
    main()
```

Una vez generando el “run” del programa:

```
GPU time: 0.59 ms
GPU Kernel Time: 0.585 ms
CPU Numpy Time: 16.280 ms
Speedup:27.83x
Correct: True
```

ECU3_1:

En este otro programa se realizan las mismas consideraciones generales que el anterior, pero en esta ocasión se está calculadora “Pitágoras” es decir **“√(a² + b²)”** para los 10 millones de datos en GPU y CPU, comparando los tiempos evaluando que sean correctos.

```
# @title
import numpy as np
from numba import cuda
import math
import time

@numba.cuda.jit
def dummy_compute_kernel(a, b, c):
    """
    Simple compute to measure timing: c[i] = sqrt(a[i]^2 + b[i]^2)
    """

    idx = cuda.grid(1)
    if idx < c.size:
        c[idx] = math.sqrt(a[idx]**2 + b[idx]**2)
```

```python
def main():
    N_large = 10_000_000

    a = np.random.randn(N_large).astype(np.float32)
    b = np.random.randn(N_large).astype(np.float32)
    c = np.zeros(N_large, dtype=np.float32)

    d_a = cuda.to_device(a)
    d_b = cuda.to_device(b)
    d_c = cuda.to_device(c)

    threads_per_block = 256
    blocks_per_grid = math.ceil(N_large / threads_per_block)

    # Warmup
    dummy_compute_kernel[blocks_per_grid, threads_per_block](d_a, d_b, d_c)
    cuda.synchronize()

    # Run timed kernel
    start = time.time()
    dummy_compute_kernel[blocks_per_grid, threads_per_block](d_a, d_b, d_c)
    cuda.synchronize()
    gpu_time = (time.time() - start) * 1000

    print(f"GPU time: {gpu_time:.2f} ms")

    # Copy result to CPU
    result = d_c.copy_to_host()

    # CPU timing
    cpu_start = time.time()
    expected = np.sqrt(a**2 + b**2)
    cpu_end = time.time()
    cpu_time = (cpu_end - cpu_start) * 1000
```

```python
    print(f"GPU Kernel Time: {gpu_time:.3f} ms")
    print(f"CPU Numpy Time: {cpu_time:.3f} ms")
    print(f"Speedup:{cpu_time / gpu_time:.2f}x")
    print(f"Correct:", np.allclose(result, expected))

"""
    # Optional: validate correctness
    if np.allclose(result, a + b):
        print("Vector add correct!")
    else:
        print("Error in GPU computation.")
"""

if __name__ == "__main__":
    main()
```

```
GPU time: 0.55 ms
GPU Kernel Time: 0.548 ms
CPU Numpy Time: 38.992 ms
Speedup:71.14x
Correct: True
```

ECU3_2:

En este otro programa se realizan generalidades como en los anteriores, pero en esta ocasión se crea una matriz de 4096x4096 con valore aleatorios se define un kernel CUDA de 2D que escala cada elemento de la matriz, se copia la matriz a la GPU.

```
if row < out.shape[0] and col < out.shape[1]:
    out[row, col] = mat[row, col] * scalar
```

posteriormente se ejecuta 2D en CUDA usando bloques de 32x32 hilos o warms, posteriormente se prepara la GPU con un warmup y se comparan tiempos y resultados de GPU con CPU usando Numpy.

```
# @title
import numpy as np
from numba import cuda
import math
import time

@cuda.jit
def matrix_scale_kernel(mat, scalar, out):
    """
    Escala cada elemento:
    out[row, col] = mat[row, col] * scalar
    """
    row, col = cuda.grid(2)

    if row < out.shape[0] and col < out.shape[1]:
        out[row, col] = mat[row, col] * scalar
```

```
def main():

    rows_large, cols_large = 4096, 4096

    # CORRECCIÓN: np.random.rand
    mat = np.random.rand(rows_large, cols_large).astype(np.float32)
    out = np.zeros_like(mat)
    scalar = 2.5

    # Copia GPU
    d_mat = cuda.to_device(mat)
    d_out = cuda.to_device(out)

    # Configuración CUDA 2D
    threads_per_block = (32, 32)
    blocks_per_grid_x = math.ceil(rows_large / threads_per_block[0])
    blocks_per_grid_y = math.ceil(cols_large / threads_per_block[1])
    blocks_per_grid = (blocks_per_grid_x, blocks_per_grid_y)
```

```
# Warmup
matrix_scale_kernel[blocks_per_grid, threads_per_block](d_mat, scalar, d_out)
cuda.synchronize()

# Timed kernel
start = time.time()
matrix_scale_kernel[blocks_per_grid, threads_per_block](d_mat, scalar, d_out)
cuda.synchronize()
gpu_time = (time.time() - start) * 1000

# CORRECCIÓN: agregar paréntesis
result = d_out.copy_to_host()
```

En esta otra sección se obtienen los tiempos e impresiones respectivamente.

```
    # CPU timing
    cpu_start = time.time()
    expected = mat * scalar
    cpu_time = (time.time() - cpu_start) * 1000

    print(f"GPU Kernel Time: {gpu_time:.3f} ms")
    print(f"CPU Numpy Time: {cpu_time:.3f} ms")
    print(f"Speedup: {cpu_time / gpu_time:.2f}x")
    print("Correct:", np.allclose(result, expected))

if __name__ == "__main__":
    main()
```

```
GPU Kernel Time: 4.189 ms
CPU Numpy Time: 20.104 ms
Speedup: 4.80x
Correct: True
```

ECU3_3:

Aquí en este programa se define un kernel de CUDA 3D para multiplicación de matrices donde cada hilo representa un triplete (row, col, k) y ese hilo calcula solo un producto parcial, el kernel no realiza la suma final, el kernel revisa los limites para no salirse de las dimensiones de las matrices, aquí haciendo usa de un elemento **cuda.atomic.add**

```
import numpy as np
from numba import cuda
import math
import time


@cuda.jit
def matmul_no_for(A, B, C):
    """
    Multiplicación de matrices C = A x B sin usar for en Python.
    Cada hilo calcula un producto parcial A[row, k] * B[k, col]
    y lo suma atómicamente en C[row, col].
    """

    # Índices globales en grid 3D
    row, col, k = cuda.grid(3)

    M, K = A.shape       # A: M x K
    K2, N = B.shape      # B: K2 x N
```

```
# Verificamos que las dimensiones sean compatibles
if K2 != K:
    return

# Verificación de límites
if row < M and col < N and k < K:
    val = A[row, k] * B[k, col]
    # Suma atómica en C[row, col]
    cuda.atomic.add(C, (row, col), val)
```

```python
def main():
    # Tamaño de matrices: A (M x K), B (K x N), C (M x N)
    M, K, N = 512, 512, 512

    # Matrices en CPU
    A = np.random.randn(M, K).astype(np.float32)
    B = np.random.randn(K, N).astype(np.float32)
    C = np.zeros((M, N), dtype=np.float32)

    # Copia a GPU
    d_A = cuda.to_device(A)
    d_B = cuda.to_device(B)
    d_C = cuda.to_device(C)  # inicializada en ceros
```

```python
# Configuración del grid 3D y bloques 3D
threads_per_block = (8, 8, 8)  # (tx, ty, tz) → 8*8*8 = 512 hilos por bloque
blocks_per_grid_x = math.ceil(M / threads_per_block[0])
blocks_per_grid_y = math.ceil(N / threads_per_block[1])
blocks_per_grid_z = math.ceil(K / threads_per_block[2])
blocks_per_grid = (blocks_per_grid_x, blocks_per_grid_y, blocks_per_grid_z)

# Warmup
matmul_no_for[blocks_per_grid, threads_per_block](d_A, d_B, d_C)
cuda.synchronize()

# Tiempo GPU
start = time.time()
matmul_no_for[blocks_per_grid, threads_per_block](d_A, d_B, d_C)
cuda.synchronize()
gpu_time = (time.time() - start) * 1000

# Copiar resultado a CPU
result = d_C.copy_to_host()
```

```python
    # Tiempo CPU (NumPy)
    cpu_start = time.time()
    expected = A @ B        # equivalente a np.dot(A, B)
    cpu_time = (time.time() - cpu_start) * 1000

    print(f"GPU Kernel Time: {gpu_time:.3f} ms")
    print(f"CPU Numpy Time: {cpu_time:.3f} ms")
    print(f"Speedup: {cpu_time / gpu_time:.2f}x")
    print("Correct:", np.allclose(result, expected, rtol=1e-3, atol=1e-3))


if __name__ == "__main__":
    main()
```

```
GPU Kernel Time: 11.264 ms
CPU Numpy Time: 7.270 ms
Speedup: 0.65x
Correct: False
```

ECU3_4:

Esta parte consistió en generar 2 matrices grandes A(1000x1000) Y B (1000x1000), definiendo un kernel de CUDA donde cada hilo calcula un elemento de la matriz C usando la formula de la sumatoria con la multiplicación.

```
total += A[row, k] * B[k, col]
```

Y con la ayuda de un ciclo for dentro del hilo para acumular la suma.

```python
# Matrix Multiplication
# Matrix Multiplication
import numpy as np
import numba.cuda as cuda
import math
import time

# C = A @ B

@numba.cuda.jit
def matmul_naive_kernel(A, B, C):
    """
    Naive matrix multiply: C = A @ B
    Each thread computes one element of C.
    All reads from A and B go to global memory (slow). Alter: shared memory

    A: (M, K)
    B: (K, N)
    C: (M, N)
    """
    row, col = cuda.grid(2)

    M, K = A.shape
    K2, N = B.shape

    if row < M and col < N:
      total = 0.0
      for k in range(K):
          total += A[row, k] * B[k, col]
      C[row, col] = total
```

Se copian las matrices a la GPU y se prepara la memoria para “C”, configurando 2D de CUDA con bloques de 32x32 hilos, calentando la GPU y ejecutando el kernel para la multiplicación.

```python
def main():
    M, K, N = 1000, 1000, 1000
    A = np.random.randn(M, K).astype(np.float32)
    B = np.random.randn(K, N).astype(np.float32)
    C = np.zeros((M, N), dtype=np.float32)

    threads_per_block = (32, 32)
    d_A = cuda.to_device(A)
    d_B = cuda.to_device(B)
    d_C = cuda.to_device(C)

    blocks_per_grid_x = (M + threads_per_block[0] - 1) // threads_per_block[0]
    blocks_per_grid_y = (N + threads_per_block[1] - 1) // threads_per_block[1]
    blocks_per_grid = (blocks_per_grid_x, blocks_per_grid_y)
```

```python
# Warmup
matmul_naive_kernel[blocks_per_grid, threads_per_block](d_A, d_B, d_C)
cuda.synchronize()

# GPU timing
start = time.time()
matmul_naive_kernel[blocks_per_grid, threads_per_block](d_A, d_B, d_C)
cuda.synchronize()
gpu_time = (time.time() - start) * 1000

C_gpu = d_C.copy_to_host()
```

Se realiza la operación tanto en GPU como en CPU realizando una comparativa de los tiempos y resultados.

```
    # CPU timing
    cpu_start = time.time()
    C_cpu = A @ B
    cpu_time = (time.time() - cpu_start) * 1000

    print(f"GPU kernel time: {gpu_time:.4f} ms")
    print(f"CPU NumPy time: {cpu_time:.4f} ms")
    print(f"Speedup: {cpu_time / gpu_time:.2f}x")
    print(f"Correct: {np.allclose(C_gpu, C_cpu, atol=1e-3)}")

main()
```

```
GPU kernel time: 52.0527 ms
CPU NumPy time: 51.3628 ms
Speedup: 0.99x
Correct: True
```

ECU3_5:

En esta ultima sección de estos laboratorios se realizo la siguiente configuración principal como encabezado anteriormente:

```
# @title
!uv pip install -q --system numba-cuda==0.4.0
!pip install pynvjitlink-cu12

import numpy as np
from numba import cuda
import time
import os
from numba import config
import numba
config.CUDA_ENABLE_PYNVJITLINK=1
```

Existieron consideraciones especiales debido a la complejidad del siguiente programa la parte principal consistió en agregar las siguientes librerias:

La importación de **cv2** fue para procesamiento de imágenes asi como de lectura y escritura con ello obtener diferentes criterios para detección e bordes, rostros y objetos siendo una librería clásica para IA. Así como también la importación de **urllib.request** para descargar y abrir archivos desde internet, la parte de **PIL** e importar imágenes es una alternativa ligera a cv2.

```
import numpy as np
import numba.cuda as cuda
import time
import cv2
import urllib.request
from PIL import Image
from matplotlib import pyplot as plt
```

Aquí el kernel **sobel_kernel** aplica el filtro de sobel, que sirve para detectar bordes en una imagen, cada hilo de la GPU procesa un pixel de la imagen y cada hilo obtiene su posición de renglón, columna dentro de la imagen, calcula los grandientes usando la mascara de sobel y combina ambos para obtener la magnitud del borde, cada hilo calcula la intensidad del borde para un pixel usando las mascaras sobel horizontal y vertical generando una imagen de bordes.

```
@cuda.jit
def sobel_kernel(img, out):
    """
    Apply Sobel edge detection - each thread processes one pixel.
    """
    row, col = cuda.grid(2)   # Each thread → one pixel

    H, W = img.shape

    # Avoid borders
    if 0 < row < H-1 and 0 < col < W-1:

        # -------- Horizontal Gradient (Gx) --------
        gx = (
            -img[row-1, col-1] + img[row-1, col+1]
            -2*img[row,   col-1] + 2*img[row,   col+1]
            -img[row+1, col-1] + img[row+1, col+1]
        )

        # -------- Vertical Gradient (Gy) --------
        gy = (
            -img[row-1, col-1] - 2*img[row-1, col] - img[row-1, col+1]
            + img[row+1, col-1] + 2*img[row+1, col] + img[row+1, col+1]
        )

        # Edge magnitude
        out[row, col] = (gx*gx + gy*gy)**0.5
```

Aquí la función sobel_opencv aplica el filtro a los bordes y devuelve la magnitud del borde, es lo mismo que hace la GPU en la parte de arriba, pero ahora relacionado esta parte con la CPU. Una vez realizado eso se descarga y se prepara la imagen pasándola por Numpy. Y esta es configurada para la GPU, se caliente con el warmup y posteriormente se ejecuta el kernel para poder aplicar los filtros de sobel.

```python
def sobel_opencv(img):
    """OpenCV CPU version using Sobel"""
    gx = cv2.Sobel(img, cv2.CV_32F, 1, 0, ksize=3)
    gy = cv2.Sobel(img, cv2.CV_32F, 0, 1, ksize=3)
    return np.sqrt(gx**2 + gy**2)

  # Load 4K image from internet
urllib.request.urlretrieve("https://picsum.photos/3840/2160", "image.jpg")
img = Image.open("image.jpg").convert('L')   # Convert to grayscale
img = np.array(img, dtype=np.float32)

H, W = img.shape
print(f"Image: {W}x{H} ({W*H:,} pixels)")

# Copiar imagen a la GPU
d_img = cuda.to_device(img)
d_out = cuda.to_device(np.zeros_like(img))

# Configuración de grid y bloques
threads = (32, 32)
blocks = ((W + 15) // 16, (H + 15) // 16)

print(f"Grid: {blocks} blocks x {threads} threads")
```

```python
# Warmup (primer lanzamiento para "despertar" la GPU)
sobel_kernel[blocks, threads](d_img, d_out)
cuda.synchronize()

# --- Ejecución cronometrada en GPU ---
start = time.time()
sobel_kernel[blocks, threads](d_img, d_out)
cuda.synchronize()
gpu_time = (time.time() - start) * 1000  # ms

# Copiar resultado de la GPU a la CPU
out_gpu = d_out.copy_to_host()

# --- Ejecución cronometrada en CPU (OpenCV) ---
start = time.time()
out_cpu = sobel_opencv(img)
cpu_time = (time.time() - start) * 1000  # ms

# Results
print("\n" + "="*60)
print("Results")
print("="*60)
print(f"GPU: {gpu_time:.2f} ms")
print(f"CPU: {cpu_time:.2f} ms")
print(f"Speedup: {cpu_time/gpu_time:.1f}x")
print(f"Correct: {np.allclose(out_gpu, out_cpu, atol=1e-3)}")
```

Una vez concluida esta parte se procede a mostrar las imágenes utilizando lo que es la librería de matplotlib y con ello poder redimensionar los elementos en la pantalla. Esto con la parte siguiente:

```python
# Resize for display
H, W = img.shape
target_w = 256
target_h = int(target_w * H / W)

def resize_for_plot(array):
    normalized = (array / array.max() * 255).astype(np.uint8)
    return np.array(Image.fromarray(normalized).resize((target_w, target_h), Image.LANCZOS))

plt.figure(figsize=(20, 10))
```

En estas secciones se agregan los elementos adicionales de la librería para posicionar los elementos de la imagen con el ploteo apropiado.

```
plt.subplot(1, 3, 1)
plt.imshow(resize_for_plot(img), cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(resize_for_plot(out_gpu), cmap='gray')
plt.title('GPU Sobel Edges')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(resize_for_plot(out_cpu), cmap='gray')
plt.title('OpenCV CPU Sobel Edges')
plt.axis('off')

plt.tight_layout()
plt.show()
```

Una vez concluida esa parte se procede a obtener una imagen random con los valores en la terminal:

5.- Conclusion

Cesar Eduardo Inda Ceniceros:

Este laboratorio fue un tanto extenso pues cubre desde las 2 sesiones sabatinas del curso, donde comenzamos comprendiendo conceptos como paralelismo y concurrencia, asi como también la introducción a lo que es toda esta parte de CUDA, definiciones clave como hilos, bloques, identificadores, ejes y cada una de las consideraciones apartir de estos conceptos que son clave y super importantes para la IA, en el sentido de quizás lo pesado que fue para en mi caso no ser algo de mi área de conocimiento directo, me pareció super importante comprender el trasfondo de todo esto pues indirectamente muchas veces asumimos como usuarios que las cosas pasan de una forma cuando en realidad los elementos abstractos nos hacen ver que no es asi, sobre todo en elementos tan importantes como es el paralelismo que muchas veces lo comprendemos como la multitarea o que muchas cosas usando todo se pueden hacer prácticamente al mismo y en esa parte existen muchas cosa criticas e importantes a tomar en cuenta, fue divertido también comprender como es la relevancia de muchas de estas cosas sobre todo utilizando la parte de CUDA para el extenso mundo de lo que es la GPU.

6.- Referencias

https://numba.pydata.org/numba-doc/0.13/CUDAJit.html

https://nvidia.github.io/numba-cuda/

7.- GitHub

https://github.com/CeicGitHub/CUDA_IA_Cinvestav_CesarInda