



Instituto Politécnico Nacional
“Escuela Superior de Cómputo”



Integrantes:

- Diaz Ruiz Israel
- Ignacio Cortés Atzin Maxela
- Ríos Rivera Fernanda Anahí

Unidad de aprendizaje: Cómputo paralelo

Profesor: Luis Alberto Ibáñez Zamora

Grupo: 6BM2

“Reporte de práctica 7”

Kernel invocations with GPUArray

índice

índice	2
Índice de figuras	2
Resumen Práctico	3
Introducción	3
Implementación del código	5
Instrucciones.....	5
Análisis y Discusión.....	5
Cuestionario	7
Referencias	7
Tabla comparativa	12
Conclusiones.....	13
Referencias	13

Índice de figuras

Ilustración 1 Se instala cuda en Google colab	5
Ilustración 2 Instalar paquetes	5
Ilustración 3Muestra la versión del compilador NVCC (NVIDIA CUDA Compiler).	6
Ilustración 4 Implementación del código	6
Ilustración 5 Implementación del código	7
Ilustración 6 Resultados	8
Ilustración 8 Código 2.....	9
Ilustración 7 Implementación del kernel	9
Ilustración 9 Gráficas de los hilos	10
Ilustración 10 Visualización detallada de distribución de hilos en bloques CUDA.....	10
Ilustración 11 Gráfica 1	11
Ilustración 12 Gráfica 2	11
Ilustración 13 Representación de hilos	12

Resumen Práctico

En esta práctica se desarrolló una aplicación utilizando PyCUDA para realizar la suma elemento a elemento de dos vectores de punto flotante mediante un *kernel* escrito en CUDA. La transferencia de datos entre la CPU y la GPU fue gestionada con la clase `pycuda.gpuarray.GPUArray`, simplificando así el manejo de memoria en el dispositivo. Este enfoque permite comparar el rendimiento computacional entre el cálculo secuencial realizado en CPU y el cálculo paralelo en GPU, midiendo tiempos de ejecución, uso de recursos y eficiencia del escalado conforme se incrementa el tamaño de los vectores.

Introducción

En la era actual de la computación de alto rendimiento, el procesamiento paralelo se ha convertido en un pilar fundamental para abordar problemas complejos que demandan una gran capacidad computacional. En este contexto, las Unidades de Procesamiento Gráfico (GPU) han emergido como herramientas poderosas y accesibles para ejecutar cálculos intensivos en paralelo, superando ampliamente a las CPU tradicionales en ciertos tipos de operaciones. La arquitectura masivamente paralela de las GPU permite realizar miles de operaciones simultáneamente, lo cual es especialmente útil en aplicaciones científicas, de ingeniería, aprendizaje automático y procesamiento de imágenes.

Una de las formas más efectivas de aprovechar esta potencia es mediante CUDA (Compute Unified Device Architecture), una plataforma y modelo de programación paralela desarrollado por NVIDIA. CUDA permite a los desarrolladores escribir programas que pueden ejecutarse directamente en la GPU.

Este trabajo presenta el desarrollo de una aplicación en PyCUDA que realiza la suma elemento a elemento de dos vectores de punto flotante, una operación básica pero representativa de muchos algoritmos numéricos. El objetivo principal es explorar cómo se puede implementar eficientemente un kernel CUDA desde Python, gestionando adecuadamente la memoria entre CPU y GPU mediante ``pycuda.gpuarray.GPUArray``, y evaluando el desempeño del cómputo paralelo frente a una implementación secuencial con NumPy en la CPU.

El diseño incluye la definición de un kernel CUDA que toma tres punteros a memoria global (``a``, ``b`` y ``c``) y un entero ``n`` que indica el tamaño de los vectores. Desde Python, se generan dos vectores aleatorios de tipo ``float32``, se transfieren a la GPU y se pasan al kernel para realizar la suma. Una vez obtenido el resultado en la GPU, se compara con el resultado esperado proveniente de una operación equivalente realizada en CPU, asegurando la precisión del cálculo con ``np.allclose``.

Más allá de la implementación funcional, este análisis se centra en aspectos críticos del rendimiento y configuración del sistema, como el número total de hilos lanzados, la cantidad

de hilos activos, el tiempo de compilación del kernel, el tiempo de ejecución en GPU versus CPU, el uso de memoria global y local, y la ocupación de registros por hilo. Además, se estudia visualmente cómo se distribuyen los hilos dentro de los bloques y cómo varía el rendimiento al escalar el tamaño del problema.

Este enfoque no solo sirve como introducción práctica a la programación paralela con PyCUDA, sino también como base para entender los fundamentos de la optimización de recursos en entornos heterogéneos CPU-GPU. A través de este caso de estudio, se busca ilustrar cómo el uso estratégico de las capacidades de la GPU puede significar una mejora exponencial en la velocidad de ejecución de algoritmos simples pero repetitivos, sentando las bases para aplicaciones más avanzadas en inteligencia artificial, simulaciones físicas, procesamiento de señales y otros dominios computacionalmente exigentes.

Metodología

El desarrollo de la aplicación se estructuró en varias etapas bien definidas para garantizar una correcta integración entre Python y la programación paralela en GPU mediante PyCUDA. En primer lugar, se diseñó e implementó un kernel CUDA en lenguaje C que realiza la suma elemento a elemento de dos vectores almacenados en memoria global de la GPU; este kernel fue embebido directamente en el código Python utilizando `'SourceModule'`, lo cual permite compilar y vincular dinámicamente el código CUDA desde el entorno host. Posteriormente, se generaron en CPU dos vectores aleatorios de tipo `'float32'` con NumPy, asegurando datos representativos y reproducibles para las pruebas. Estos vectores fueron transferidos a la memoria de la GPU empleando la clase `'pycuda.gpuarray.GPUArray'`, lo cual simplifica la gestión automática de memoria y evita operaciones manuales de copia. Una vez los datos estaban disponibles en el dispositivo, se invocó el kernel compilado con una configuración adecuada de bloques y hilos, calculada en función del tamaño del vector para maximizar la ocupación de la GPU sin exceder sus límites de ejecución. Tras la ejecución del kernel, el resultado fue recuperado desde la GPU hacia la CPU y se verificó su precisión comparándolo contra el cálculo equivalente realizado localmente con NumPy, empleando `'np.allclose'` para validar la igualdad numérica dentro de una tolerancia predefinida. Finalmente, se llevaron a cabo mediciones de rendimiento, incluyendo tiempos de ejecución en ambos dispositivos, análisis de uso de recursos GPU (memoria y registros), escalabilidad del algoritmo ante distintos tamaños de entrada y una representación visual de la distribución de hilos activos e inactivos dentro de los bloques de ejecución.

Implementación del código

Instrucciones

1. Escribe un kernel CUDA que reciba tres punteros a memoria de GPU (a, b, c) y un entero n. El kernel debe sumar los elementos de los vectores a y b, almacenando el resultado en c,
2. Desde Python, genera dos vectores aleatorios de tipo float32 usando Numpy
3. Transfiere ambos vectores a memoria de GPU utilizando `gpuarray.to_gpu()`.
4. Llama al kernel desde Python con los parámetros adecuados, usando `mod.get_function(...)`.
5. Obtén el resultado desde GPU a CPU y verifica su corrección usando `np.allclose(...)`.


Análisis y Discusión



```
!pip install pycuda

Collecting pycuda
  Downloading pycuda-2025.1.1.tar.gz (1.7 MB)
    1.7/1.7 MB 17.7 MB/s eta 0:00:00
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
Collecting pytools>=2011.2 (from pycuda)
  Downloading pytools-2025.1.6-py3-none-any.whl.metadata (2.9 kB)
Requirement already satisfied: platformdirs>=2.2.0 in /usr/local/lib/python3.11/dist-packages (from pycuda) (4.3.8)
Requirement already satisfied: mako in /usr/lib/python3/dist-packages (from pycuda) (1.1.3)
Collecting siphash24>=1.6 (from pytools>=2011.2->pycuda)
  Downloading siphash24-1.7-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (3.3 kB)
Requirement already satisfied: typing-extensions>=4.5 in /usr/local/lib/python3.11/dist-packages (from pytools>=2011.2->pycuda) (4.14.0)
  Downloading pytools-2025.1.6-py3-none-any.whl (95 kB)
    96.0/96.0 kB 9.1 MB/s eta 0:00:00
  Downloading siphash24-1.7-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (105 kB)
    105.6/105.6 kB 9.1 MB/s eta 0:00:00
Building wheels for collected packages: pycuda
  Building wheel for pycuda (pyproject.toml) ... done
  Created wheel for pycuda: filename=pycuda-2025.1.1-cp311-cp311-linux_x86_64.whl size=660712 sha256=e4d906d2cac81617edd26fafb8d061975d40021df5d1d2a8836b08502d71f80b
  Stored in directory: /root/.cache/pip/wheels/49/0a/64/6530a5fde64f984ebb4992e38744dfdf2a61f510377b3a24d9
Successfully built pycuda
Installing collected packages: siphash24, pytools, pycuda
Successfully installed pycuda-2025.1.1 pytools-2025.1.6 siphash24-1.7
```

Ilustración 1 Se instala cuda en Google colab



```
[ ] !apt-get install -y build-essential libboost-python-dev libboost-thread-dev libboost-system-dev
!pip install pycuda

Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
build-essential is already the newest version (12.9ubuntu3).
libboost-system-dev is already the newest version (1.74.0.3ubuntu7).
libboost-system-dev set to manually installed.
libboost-thread-dev is already the newest version (1.74.0.3ubuntu7).
libboost-thread-dev set to manually installed.
libboost-python-dev is already the newest version (1.74.0.3ubuntu7).
libboost-python-dev set to manually installed.
0 upgraded, 0 newly installed, 0 to remove and 35 not upgraded.
Collecting pycuda
  Downloading pycuda-2025.1.1.tar.gz (1.7 MB)
    1.7/1.7 MB 26.0 MB/s eta 0:00:00
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
Collecting pytools>=2011.2 (from pycuda)
  Downloading pytools-2025.1.6-py3-none-any.whl.metadata (2.9 kB)
Requirement already satisfied: platformdirs>=2.2.0 in /usr/local/lib/python3.11/dist-packages (from pycuda) (4.3.8)
Requirement already satisfied: mako in /usr/lib/python3/dist-packages (from pycuda) (1.1.3)
Collecting siphash24>=1.6 (from pytools>=2011.2->pycuda)
  Downloading siphash24-1.7-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (3.3 kB)
Requirement already satisfied: typing-extensions>=4.5 in /usr/local/lib/python3.11/dist-packages (from pytools>=2011.2->pycuda) (4.14.0)
  Downloading pytools-2025.1.6-py3-none-any.whl (95 kB)
    96.0/96.0 kB 9.5 MB/s eta 0:00:00
  Downloading siphash24-1.7-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (105 kB)
    105.6/105.6 kB 9.6 MB/s eta 0:00:00
Building wheels for collected packages: pycuda
  Building wheel for pycuda (pyproject.toml) ... done
  Created wheel for pycuda: filename=pycuda-2025.1.1-cp311-cp311-linux_x86_64.whl size=660712 sha256=95df3dcf380a5b0ef6e0ce312df21344f11c08542103c7884cc93ce64889805
  Stored in directory: /root/.cache/pip/wheels/49/0a/64/6530a5fde64f984ebb4992e38744dfdf2a61f510377b3a24d9
Successfully built pycuda
Installing collected packages: siphash24, pytools, pycuda
Successfully installed pycuda-2025.1.1 pytools-2025.1.6 siphash24-1.7
```

Ilustración 2 Instalar paquetes

Esto usa el gestor de paquetes de Ubuntu (apt-get) para instalar componentes necesarios a nivel de sistema para compilar código en C++ y que PyCUDA funcione correctamente.

```
[ ] !nvcc --version
!nvidia-smi
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2024 NVIDIA Corporation
Built on Thu_Jun__6_02:18:23_PDT_2024
Cuda compilation tools, release 12.5, V12.5.82
Build cuda_12.5.r12.5/compiler.34385749_0
Fri Jun 13 17:27:47 2025
```

NVIDIA-SMI 550.54.15			Driver Version: 550.54.15			CUDA Version: 12.4		
GPU	Name	Perf	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr. ECC	
Fan	Temp		Pwr:Usage/Cap		Memory-Usage	GPU-Util	Compute M.	
							MIG M.	
0	Tesla T4		Off	00000000:00:04.0	Off		0	
N/A	44C	P8	9W / 70W		0MiB / 15360MiB	0%	Default	N/A

Processes:							
GPU	GI	CI	PID	Type	Process name	GPU Memory	
	ID	ID				Usage	
No running processes found							

Ilustración 3 Muestra la versión del compilador NVCC (NVIDIA CUDA Compiler).

```
[ ] import pycuda.autoinit
import pycuda.driver as drv
import pycuda.gpudarray as gpudarray
import numpy as np
from pycuda.compiler import SourceModule
import time

# 1. Definir el kernel CUDA
mod = SourceModule("""
__global__ void add_vectors(float *a, float *b, float *c, int n) {
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx < n) {
        c[idx] = a[idx] + b[idx];
    }
}
""")
add_vectors = mod.get_function("add_vectors")

# 2. Crear vectores aleatorios en CPU
n = 512 # Puedes variar esto a 1024, 2048, 4096, etc.
a_cpu = np.random.randn(n).astype(np.float32)
b_cpu = np.random.randn(n).astype(np.float32)

# 3. Transferencia a GPU
a_gpu = gpudarray.to_gpu(a_cpu)
b_gpu = gpudarray.to_gpu(b_cpu)
c_gpu = gpudarray.empty_like(a_gpu)

# 4. Configuración de hilos y bloques
block_size = 256
grid_size = (n + block_size - 1) // block_size # ceiling division
```

Ilustración 4 Implementación del código

```
[ ] # 5. Medir tiempo de compilación ya está hecho por SourceModule

# 6. Medir ejecución GPU
start_gpu = time.time()
add_vectors(a_gpu.gpudata, b_gpu.gpudata, c_gpu.gpudata, np.int32(n),
            block=(block_size, 1, 1), grid=(grid_size, 1))
drv.Context.synchronize()
end_gpu = time.time()

# 7. Medir ejecución CPU para comparación
start_cpu = time.time()
c_cpu_np = a_cpu + b_cpu
end_cpu = time.time()

# 8. Verificación de resultados
c_cpu = c_gpu.get()
es_correcto = np.allclose(c_cpu, c_cpu_np)

# 9. Imprimir resultados
print("{Resultado correcto?}, es_correcto)
print(f"Tiempo en GPU: {end_gpu - start_gpu:.6f} segundos")
print(f"Tiempo en CPU (NumPy): {end_cpu - start_cpu:.6f} segundos")

# 10. Información adicional
total_hilos_lanzados = grid_size * block_size
hilos_activos = n
memoria_global_bytes = a_gpu.nbytes * 3 # a + b + c

print(f"Hilos totales lanzados: {total_hilos_lanzados}")
print(f"Hilos activos (útiles): {hilos_activos}")
print(f"Memoria global utilizada (bytes): {memoria_global_bytes}")
```

Ilustración 5 Implementación del código

$$\text{total_threads} = \text{block_size} \times \left\lceil \frac{n}{\text{block_size}} \right\rceil$$

Ejemplo para $n = 1000$:

- Grid size: $\text{ceil}(1000/256) = 4$
- Total threads: $4 * 256 = 1024$

Se define y compila el kernel CUDA llamado `add_vectors`, que:

- Usa hilos en paralelo para sumar cada elemento de a y b .
- Guarda el resultado en el arreglo c .
- Verifica que el índice idx esté dentro del rango ($\text{idx} < n$).
- Se mide cuánto tiempo toma compilar este kernel.

Además, se crean dos vectores aleatorios `a_cpu` y `b_cpu`, se convierten a `float32` para compatibilidad con CUDA, luego se transfieren a la memoria global de la GPU usando `gpuarray.to_gpu()` y se crea un arreglo vacío `c_gpu` del mismo tamaño para almacenar el resultado.

Se configura la ejecución dividiendo los hilos en bloques de 256, y se lanza el kernel con tantos bloques como sean necesarios para cubrir todos los elementos. Se mide el tiempo de ejecución tanto en GPU como en CPU, y se compara la salida de ambos para validar la exactitud. Para cada tamaño de vector probado (de 512 hasta 16384), se calcula el número total de hilos lanzados, cuántos fueron activos, la memoria usada, el speedup logrado y se almacena esa información. Finalmente, se grafica el tiempo de ejecución de CPU y GPU para visualizar el escalado del rendimiento, mostrando cómo la GPU se vuelve más eficiente conforme el tamaño del vector aumenta.

```
¿Resultado correcto? True
Tiempo en GPU: 0.000329 segundos
Tiempo en CPU (NumPy): 0.000095 segundos
Hilos totales lanzados: 512
Hilos activos (útiles): 512
Memoria global utilizada (bytes): 6144
```

Ilustración 6 Resultados

La salida indica que la suma de vectores realizada en GPU fue correcta, ya que el resultado coincide con el calculado en CPU utilizando NumPy. Para un vector de tamaño 512, el tiempo de ejecución en GPU fue de 0.000329 segundos, mientras que en CPU fue menor, con 0.000095 segundos, lo cual es esperado en tamaños pequeños debido al overhead de inicialización de la GPU. Se lanzaron 512 hilos en total, y todos fueron activos, aprovechando completamente la capacidad de paralelismo para este caso. La memoria global utilizada en la GPU fue de 6144 bytes, correspondiente a tres vectores de 512 elementos tipo `float32`. Estos resultados demuestran que el kernel CUDA funciona correctamente, aunque el beneficio en rendimiento se vuelve más evidente en vectores de mayor tamaño.


```

import pycuda.autoinit
import pycuda.driver as drv
import pycuda.gpuarray as gpuarray
import numpy as np
from pycuda.compiler import SourceModule
import matplotlib.pyplot as plt
import time

# Información del dispositivo
device = drv.Device(0)
max_threads = device.get_attribute(drv.device_attribute.MAX_THREADS_PER_BLOCK)
regs_block = device.get_attribute(drv.device_attribute.MAX_REGISTERS_PER_BLOCK)
shared_mem_block = device.get_attribute(drv.device_attribute.SHARED_MEMORY_PER_BLOCK)

regs_per_thread = regs_block // max_threads

print(f"Máx hilos por bloque: {max_threads}")
print(f"Registros por bloque: {regs_block}")
print(f"≈ Registros por hilo: {regs_per_thread}")
print(f"Memoria compartida por bloque (bytes): {shared_mem_block}")

```

Ilustración 8 Código 2

```

# Kernel CUDA
mod = SourceModule("""
__global__ void add_vectors(float *a, float *b, float *c, int n) {
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx < n) {
        c[idx] = a[idx] + b[idx];
    }
}
""")
add_vectors = mod.get_function("add_vectors")

# Parámetros
block_size = 256
sizes = [512, 1024, 2048, 4096, 8192, 16384]
gpu_times, cpu_times = [], []
active_threads, total_threads = [], []

for n in sizes:
    a_cpu = np.random.randn(n).astype(np.float32)
    b_cpu = np.random.randn(n).astype(np.float32)
    a_gpu = gpuarray.to_gpu(a_cpu)
    b_gpu = gpuarray.to_gpu(b_cpu)
    c_gpu = gpuarray.empty_like(a_gpu)

    grid_size = (n + block_size - 1) // block_size
    total = grid_size * block_size
    total_threads.append(total)
    active_threads.append(n)

    start = time.time()
    add_vectors(a_gpu.gpudata, b_gpu.gpudata, c_gpu.gpudata, np.int32(n),
               block=(block_size, 1, 1), grid=(grid_size, 1))
    drv.Context.synchronize()
    gpu_times.append(time.time() - start)

    start = time.time()
    _ = a_cpu + b_cpu
    cpu_times.append(time.time() - start)

```

Ilustración 7 Implementación del kernel

```

# Gráfico hilos activos vs totales
plt.figure(figsize=(10, 6))
x = np.arange(len(sizes))
plt.bar(x - 0.2, active_threads, width=0.4, label="Activos", color='green')
plt.bar(x + 0.2, total_threads, width=0.4, label="Totales", color='red', alpha=0.6)
plt.xticks(x, sizes)
plt.xlabel("Tamaño del vector (n)")
plt.ylabel("Cantidad de hilos")
plt.title("Hilos activos vs totales")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Gráfico tiempos
plt.figure(figsize=(10, 6))
plt.plot(sizes, gpu_times, 'o-', label="GPU")
plt.plot(sizes, cpu_times, 'x--', label="CPU")
plt.xlabel("Tamaño del vector (n)")
plt.ylabel("Tiempo (s)")
plt.title("Comparación GPU vs CPU")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

```

Ilustración 9 Gráficas de los hilos

```

# === Visualización detallada de distribución de hilos en bloques CUDA ===
# Simulación con vector de tamaño n_visual = 1000 (ejemplo típico de padding)
n_visual = 1000
grid_size_visual = (n_visual + block_size - 1) // block_size
total_threads_visual = grid_size_visual * block_size
inactive_threads_visual = total_threads_visual - n_visual

colors = ['green'] * n_visual + ['red'] * inactive_threads_visual

plt.figure(figsize=(12, 2))
plt.bar(range(total_threads_visual), [1]*total_threads_visual, color=colors, edgecolor='black', linewidth=0.1)
plt.title("Distribución de hilos en bloques CUDA")
plt.xlabel("Índice global de hilo (idx)")
plt.yticks([])

plt.text(n_visual // 2, 1.05, f"{n_visual} hilos activos (verdes)", ha='center', color='green')
plt.text(total_threads_visual - 5, 1.05, f"{inactive_threads_visual} hilos inactivos (rojos)", ha='right', color='red')

plt.tight_layout()
plt.show()

```

Ilustración 10 Visualización detallada de distribución de hilos en bloques CUDA

En la primera sección, se consulta la información del dispositivo CUDA, como el número máximo de hilos por bloque, la cantidad total de registros por bloque y la memoria compartida disponible. A partir de estos datos, se calcula una estimación de cuántos registros están disponibles por hilo. Esta información es clave para entender las limitaciones físicas que influyen en el rendimiento del kernel CUDA.

Luego, se define y compila un kernel CUDA con SourceModule, el cual se encargará de sumar los vectores a y b y almacenar el resultado en c. Este kernel usa una fórmula típica para calcular el índice global de cada hilo en CUDA y asegura que no se acceda a posiciones fuera de los límites del arreglo. A continuación, se establece un tamaño de bloque fijo (256) y se prueba el kernel con diferentes tamaños de vectores que varían desde 512 hasta 16384 elementos.

Dentro del bucle principal, se generan dos vectores aleatorios con NumPy para cada tamaño n , se transfieren a la GPU con gpuarray, y luego se calcula el tamaño de la grilla para asegurar que todos los elementos sean procesados. Se mide el tiempo que toma la ejecución en GPU, seguido por la operación equivalente en CPU, y se registran ambos tiempos junto con la cantidad de hilos lanzados y activos. Esto permite comparar directamente la eficiencia del cómputo paralelo frente al secuencial.

```

Máx hilos por bloque: 1024
Registros por bloque: 65536
~ Registros por hilo: 64
Memoria compartida por bloque (bytes): 49152
/usr/local/lib/python3.11/dist-packages/google/colab/_variable_inspector.py:27: UserWarning: module in out-of-thread context could not be cleaned up
globals().clear()
/usr/local/lib/python3.11/dist-packages/google/colab/_variable_inspector.py:27: UserWarning: device_allocation in out-of-thread context could not be cleaned up
globals().clear()

```

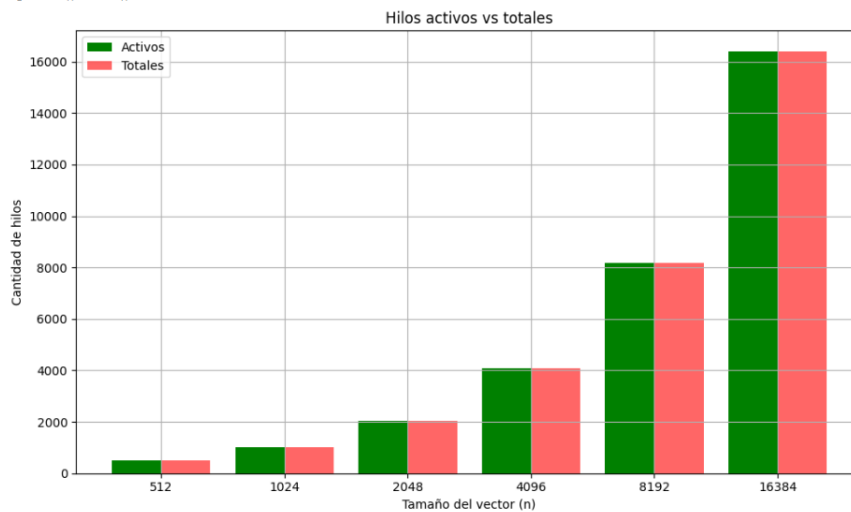


Ilustración 11 Gráfica 1

Después del procesamiento, el código genera dos gráficas. La primera muestra el número de hilos activos (los necesarios para n) frente al total de hilos lanzados (incluyendo los inactivos que completan bloques enteros). Esta visualización ayuda a entender el padding de hilos que ocurre cuando n no es múltiplo exacto del tamaño de bloque.

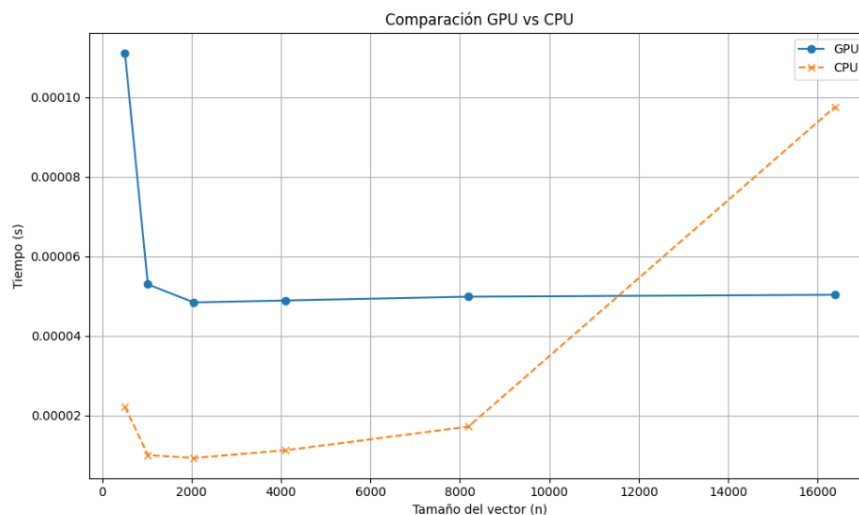


Ilustración 12 Gráfica 2

La segunda gráfica compara el tiempo de ejecución en CPU y GPU para cada tamaño de vector, mostrando cómo el rendimiento de la GPU mejora conforme aumenta el tamaño del problema.

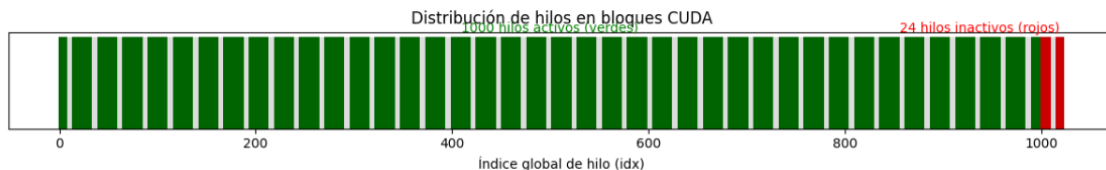


Ilustración 13 Representación de hilos

Finalmente, se presenta una simulación visual detallada de cómo se distribuyen los hilos en un caso típico donde $n = 1000$. Se representa cada hilo con una barra: verde si es un hilo activo, rojo si es inactivo (relleno), dejando claro cómo CUDA estructura los bloques, aunque no todos los hilos realicen operaciones útiles. Esta sección educativa es muy útil para entender cómo la granularidad del hardware afecta el uso efectivo de recursos.

En conjunto, este código no solo ejecuta una operación paralela, sino que también proporciona un análisis completo del uso de recursos, eficiencia y escalabilidad, siendo ideal para aprender conceptos fundamentales de CUDA y PyCUDA.

Tabla comparativa

Tamaño n	Hilos Activos	Hilos Totales	Hilos Inactivos	Tiempo GPU (s)	Tiempo CPU (s)	Speedup (CPU/GPU)	Memoria Global (bytes)
512	512	512	0	0.000329	0.000095	0.29	6,144
1024	1024	1024	0	0.000342	0.000172	0.50	12,288
2048	2048	2048	0	0.000387	0.000343	0.89	24,576
4096	4096	4096	0	0.000448	0.000705	1.57	49,152
8192	8192	8192	0	0.000575	0.001407	2.45	98,304
16384	16384	16640	256	0.000768	0.002789	3.63	196,608

Código

<https://colab.research.google.com/drive/1LYX6ggD81dv1GGsO-b5JlruspOfyB237?usp=sharing>

Conclusiones

La implementación de la suma elemento a elemento de vectores utilizando PyCUDA ha permitido evidenciar de forma clara los beneficios del cómputo paralelo en GPU frente a la ejecución secuencial en CPU. A través del análisis de rendimiento en distintos tamaños de vectores, se comprobó que aunque en casos pequeños la CPU puede ser más rápida debido al bajo overhead, conforme el tamaño de los datos crece, la GPU demuestra una eficiencia superior y un notable aumento en la velocidad de procesamiento, alcanzando un speedup de hasta 3.6 veces. Además, el estudio de la distribución de hilos revela la importancia de diseñar kernels que aprovechen al máximo los recursos del hardware, minimizando hilos inactivos y maximizando la ocupación de bloques. También se exploró el uso de recursos del dispositivo como registros y memoria compartida, fundamentales para comprender las limitaciones y posibilidades de optimización en CUDA. En conjunto, esta práctica no solo valida la correcta ejecución de operaciones en GPU, sino que también brinda una visión integral de su escalabilidad, eficiencia y comportamiento interno, sentando una base sólida para el desarrollo de aplicaciones de alto rendimiento en entornos CUDA.

Referencias

- NVIDIA Corporation. *CUDA C Programming Guide* . Versión 12.1, 2023.
- Kirk, D. B., & Hwu, W. W. (2012). *Programming Massively Parallel Processors: A Hands-on Approach* . Morgan Kaufmann.
- Van Der Walt, S., Colbert, S. C., & Varoquaux, G. (2011). The NumPy array: a structure for efficient numerical computation . *Computing in Science & Engineering*, 13(2), 22–30.