Computación de Altas Prestaciones (2025-2026)

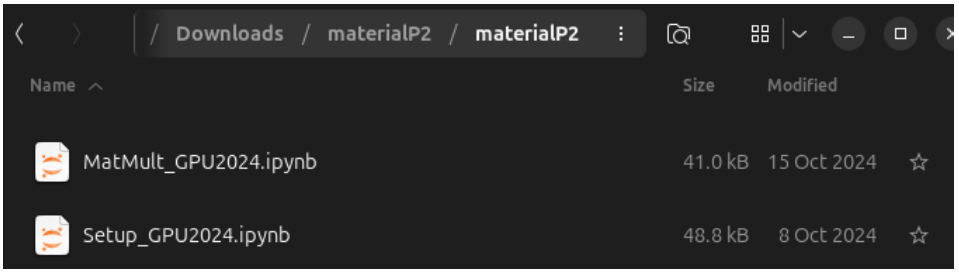
Realizado por: Javier Agüero y Marouane Chahbar

Fecha: 30/10/2025

Práctica 2: Programación de GPU

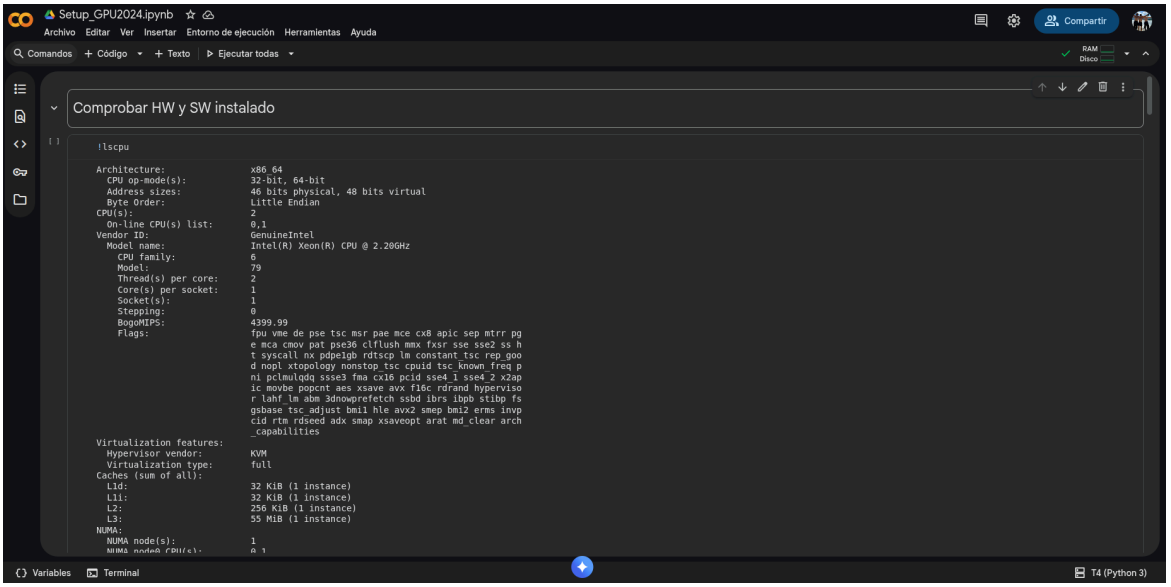
Ejercicio 0: Configuración

1. Primero obtenemos una copia del cuaderno *Setup_GPU.ypynb:*



2. Pasamos a crear la carpeta en Google Drive correspondiente con dicho archivo cargado:

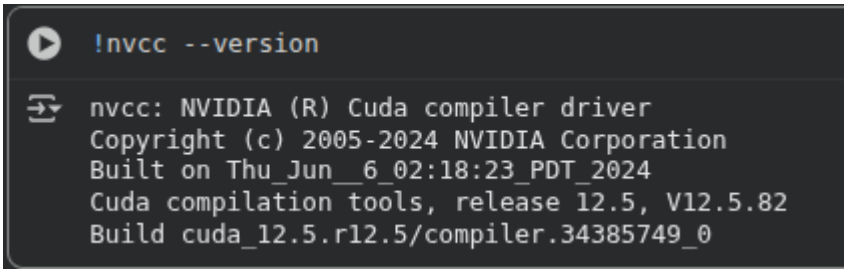
3.
4. Al hacer doble click nos manda a Google Collab, una vez dentro nos conectamos al entorno virtual (CPU Y GPU):



Donde podemos ver que estamos conectados a T4, la cual hace referencia al tipo de GPU que se nos ha asignado a nuestra sesión virtual. De forma que los comandos y/o programas que ejecutemos en nuestro collab se ejecutarán en dicha GPU.

5. Aquí tenemos un ejemplo de ejecución de comandos con "!". En este caso ejecutamos *lscpu* el cual nos muestra detalles del procesador del entorno al que estamos conectados.

6. En este caso vemos que nuestra versión del compilador CUDA es la v12.5.82.



```
!nvidia-smi

Thu Oct 30 18:09:02 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.54.15              Driver Version: 550.54.15      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  Tesla T4                       Off |   00000000:00:04.0 Off |                    0 |
| N/A   40C    P8             11W /   70W |       0MiB /  15360MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|  No running processes found                                                             |
+-----------------------------------------------------------------------------------------+
```

Con *nvidia-smi* podemos ver en detalle las características y procesos activos de la gpu que se nos ha asignado virtualmente.

7. Aquí tenemos un ejemplo de *Hello World* en CUDA, así como de una suma simple de números:

```
%%writefile cuda/holamundo.cu

#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>

__global__ void mykernel(void) {

}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello world\n");
    return 0;
}
```

```
%%writefile cuda/suma1d.cu
#include <cuda.h>
#include <cuda_runtime.h>
#include <stdio.h>

__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}

int main(void) {
    int a, b, c;            // host copies of a, b, c
    int *d_a, *d_b, *d_c;   // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Setup input values
    a = 2;
    b = 7;

    // Copy inputs to device
    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<1,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

    // Show result
    printf("%d + %d = %d", a, b, c);

    // Cleanup
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

8.
9. En estos apartados únicamente se nos muestra cómo compilar programas con *nvcc* y conectar el Google Drive a nuestro ordenador Colab para compartir archivos.

Ejercicio 1: Stencil 1D

1. A partir del Stencil 1D de las diapositivas hemos escrito el siguiente código:

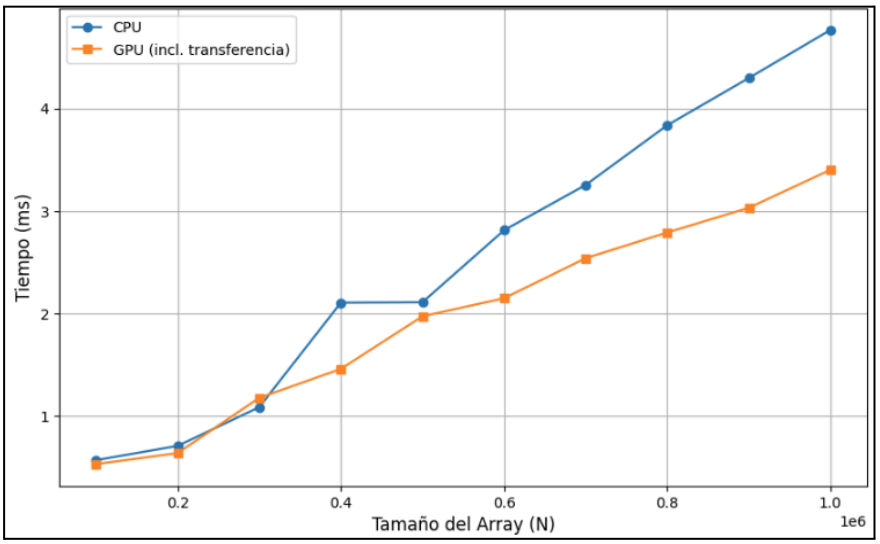
```
/**
* @brief Kernel CUDA para Stencil 1D optimizado.
*
* Utiliza memoria compartida para reducir accesos a memoria
global.
* Este kernel corrige los bugs críticos de las diapositivas.
*/
__global__ void stencil_1d_gpu(int *in, int *out, int N) {

   // Declaración de memoria compartida
   __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];

   // Índices global y local
   int gindex = threadIdx.x + blockIdx.x * blockDim.x;
   int lindex = threadIdx.x + RADIUS;

   // 1. Cargar datos de memoria global a compartida

   // Cargar elemento central (con chequeo de límites)
   if (gindex < N) {
       temp[lindex] = in[gindex];
   } else {
       temp[lindex] = 0; // Padding si gindex > N
   }

   // Cargar halos (solo los primeros/últimos RADIUS hilos)
   if (threadIdx.x < RADIUS) {
       // Cargar halo izquierdo
       int left_gindex = gindex - RADIUS;
       // CORRECCIÓN BUG: Chequear 'gindex < RADIUS'
       if (left_gindex < 0) {
           temp[lindex - RADIUS] = 0;
       } else {
           temp[lindex - RADIUS] = in[left_gindex];
       }
```

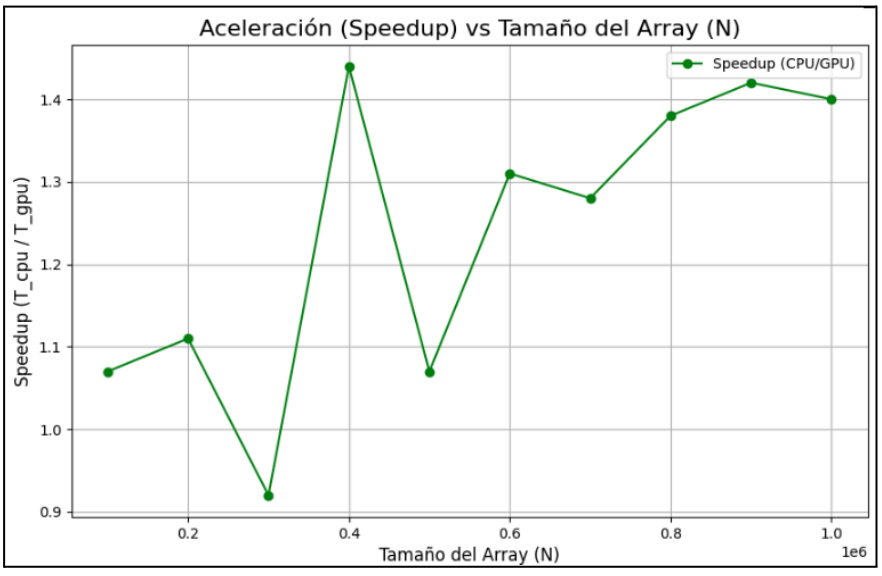
```
        // Cargar halo derecho
        int right_gindex = gindex + blockDim.x; // (blockDim.x es
el BLOCK_SIZE)
        // CORRECCIÓN BUG: Chequear 'gindex + BLOCK_SIZE > N-1'
        if (right_gindex >= N) {
            temp[lindex + blockDim.x] = 0;
        } else {
            temp[lindex + blockDim.x] = in[right_gindex];
        }
    }

    // Sincronizar hilos del bloque
    // Asegura que toda la memoria compartida esté cargada antes
de calcular.
    __syncthreads();

    // 2. Aplicar el stencil desde la memoria compartida
    if (gindex < N) {
        int result = 0;
        for (int offset = -RADIUS; offset <= RADIUS; offset++) {
            result += temp[lindex + offset];
        }
        out[gindex] = result;
    }
}
```

El código del stencil 1D optimizado utiliza la memoria compartida de la GPU para mejorar el rendimiento del cálculo. Cada bloque de hilos copia a una zona rápida de memoria (__shared__) los datos que necesita, incluyendo los elementos vecinos o “halos”, de forma que cada valor se lee de la memoria global solo una vez y luego se reutiliza dentro del bloque. Después de sincronizar los hilos para asegurar que todos los datos están cargados, cada hilo suma su valor y los de sus vecinos para obtener el resultado. Este enfoque reduce significativamente los accesos a memoria global, que son más lentos, y aprovecha mejor el paralelismo masivo de la GPU.

2. Aquí tenemos las gráficas resultados de la comparativa:

	N	CPU (ms)	GPU (ms)	Speedup
0	100000	0.569	0.530	1.07
1	200000	0.708	0.638	1.11
2	300000	1.086	1.177	0.92
3	400000	2.106	1.459	1.44
4	500000	2.110	1.973	1.07
5	600000	2.813	2.149	1.31
6	700000	3.255	2.540	1.28
7	800000	3.838	2.791	1.38
8	900000	4.299	3.032	1.42
9	1000000	4.768	3.403	1.40





Al comparar el rendimiento del stencil 1D en CPU y GPU se observa que, para tamaños pequeños de matriz, la GPU no ofrece gran ventaja porque el tiempo de copiar los datos hacia y desde la memoria de la GPU es comparable al tiempo de cálculo. Sin embargo, a medida que aumenta el tamaño de N, la GPU empieza a mostrar una clara mejora, ya que puede procesar miles de elementos en paralelo y aprovechar la memoria compartida para reducir accesos lentos a memoria global. Esto hace que el tiempo de ejecución en GPU crezca mucho más lento que en CPU, logrando aceleraciones de varias veces cuando el tamaño es grande. El uso de un tamaño de bloque de 256 hilos resulta adecuado, pues mantiene un buen equilibrio entre paralelismo y eficiencia de memoria, permitiendo aprovechar bien los recursos de la GPU sin desperdiciarlos.

3. En nuestra implementación utilizamos un tamaño de bloque de 256 hilos, que es un valor muy común y equilibrado en CUDA. Este número es múltiplo de 32, lo que permite aprovechar al máximo la estructura interna de la GPU (los warps). En las pruebas, este tamaño ofreció un buen rendimiento, ya que mantiene una alta ocupación de la GPU sin usar demasiada memoria compartida por bloque. Probé otros valores y las diferencias fueron pequeñas, por lo que 256 resulta un tamaño adecuado y cercano al óptimo para este tipo de cálculo. En resumen, este valor logra un buen equilibrio entre paralelismo, uso eficiente de recursos y velocidad de ejecución.

Ejercicio 2 : Tutorial multiplicación de matrices con GPU.

1. La versión 1D realizada es la siguiente:

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>

#define N 3

__global__ void matrix_mul_1d(const int *a, const int *b, int *c,
int n) {
   int id = blockIdx.x * blockDim.x + threadIdx.x;
   int total = n * n;
   if (id < total) {
       int row = id / n;
       int col = id % n;
       int sum = 0;
       for (int k = 0; k < n; k++) {
```

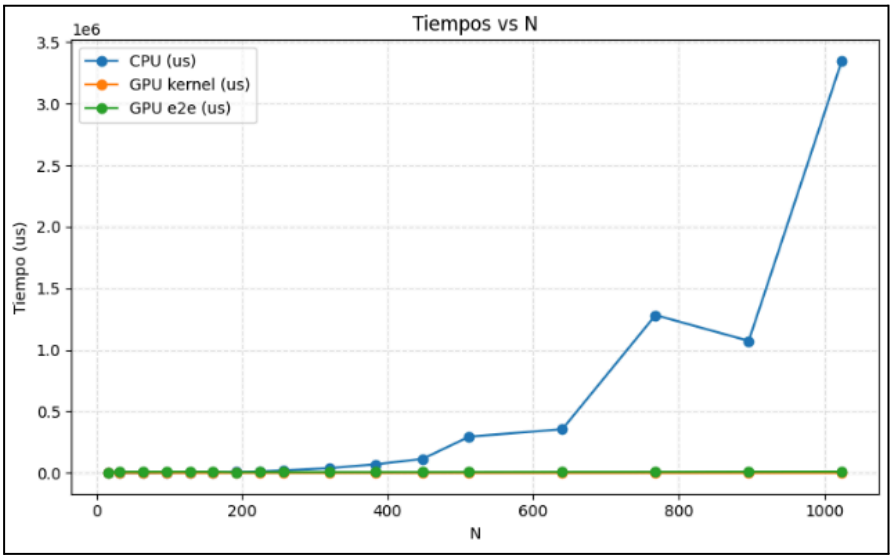
```
            sum += a[row * n + k] * b[k * n + col];
        }
        c[row * n + col] = sum;
    }
}

void printMat(const int *mm, const char *nombre, int n){
  for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            printf(" %c[%d][%d] = %d ",nombre[0] ,i, j, mm[i * n +
j]);
        printf("\n");
    }
}

int main() {
    int n = N;
    int *a, *b, *c;
    int *d_a, *d_b, *d_c;
    int size = n * n * sizeof(int);
    char matA[]= "A";
    char matB[]= "B";
    char matC[]= "C";

    a = (int *)malloc(size);
    b = (int *)malloc(size);
    c = (int *)malloc(size);

    // Inicializacion de valores
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            a[i * n + j] = i + j;
            b[i * n + j] = i * j;
        }

    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    int total = n*n;
```

```
    int threadsPerBlock = 128;
    int blocksPerGrid = (total + threadsPerBlock - 1) /
threadsPerBlock;
    matrix_mul_1d<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b,
d_c, n);

    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    printMat( a, matA, n); printf("\n");
    printMat( b, matB, n); printf("\n");
    printMat( c, matC, n);

    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    free(a);
    free(b);
    free(c);

    return 0;
}
```

Para ello, usamos grid 1D y bloques 1D; cada hilo calcula un elemento C[row,col] a partir de un índice lineal id → (row=id/N, col=id%N).

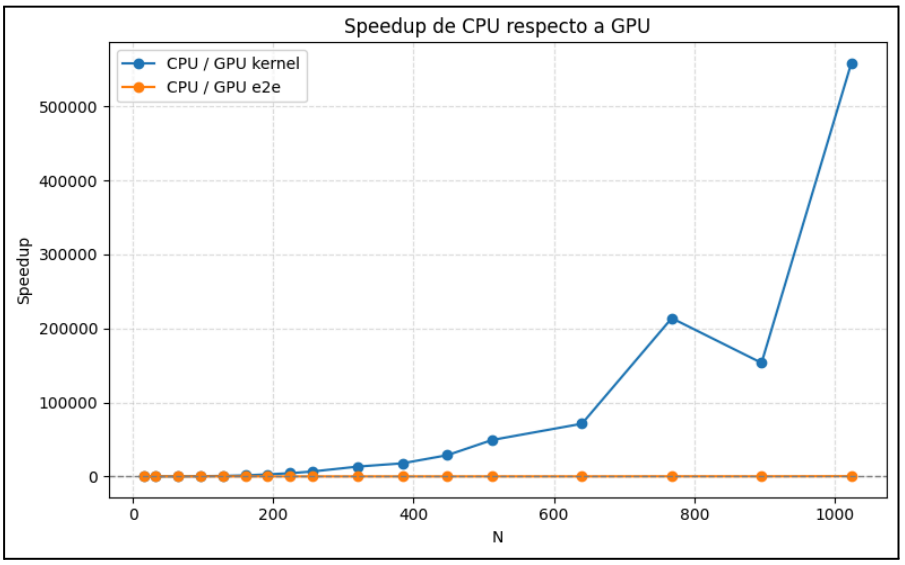
La versión 2D (grid y bloques bidimensionales) resulta más eficiente que la versión 1D, porque se adapta mejor a la estructura natural del problema de multiplicación de matrices. En una matriz, los datos están organizados por filas y columnas, y al usar una malla 2D de hilos, cada hilo puede calcular directamente un elemento (i, j) del resultado, accediendo de forma más ordenada y unificada a la memoria global. En cambio, la versión 1D necesita hacer conversiones de índices y sus accesos a memoria son menos regulares, lo que provoca más latencia y menor rendimiento. Por eso, aunque ambas versiones son funcionales, la versión 2D aprovecha mejor la jerarquía de memoria y el paralelismo de la GPU, siendo claramente más eficiente.

2. Primeramente, hemos obtenidos las siguientes gráficas comparativas:



Siendo e2e "end to end" y us "microsegundos".

Con la siguiente aceleración,



Como podemos ver, existe un punto, generalmente con valores de N pequeños o medios (por ejemplo en este caso, alrededor de N ≈ 256 o 512), donde el tiempo de ejecución de la GPU y la CPU se igualan. Antes de ese valor, la CPU puede ser incluso más rápida porque el coste de transferir datos a la GPU y lanzar el kernel supera el tiempo de cálculo. A partir de ese N, la GPU empieza a ser mucho más eficiente, ya que puede explotar su paralelismo y reducir drásticamente el tiempo de ejecución.

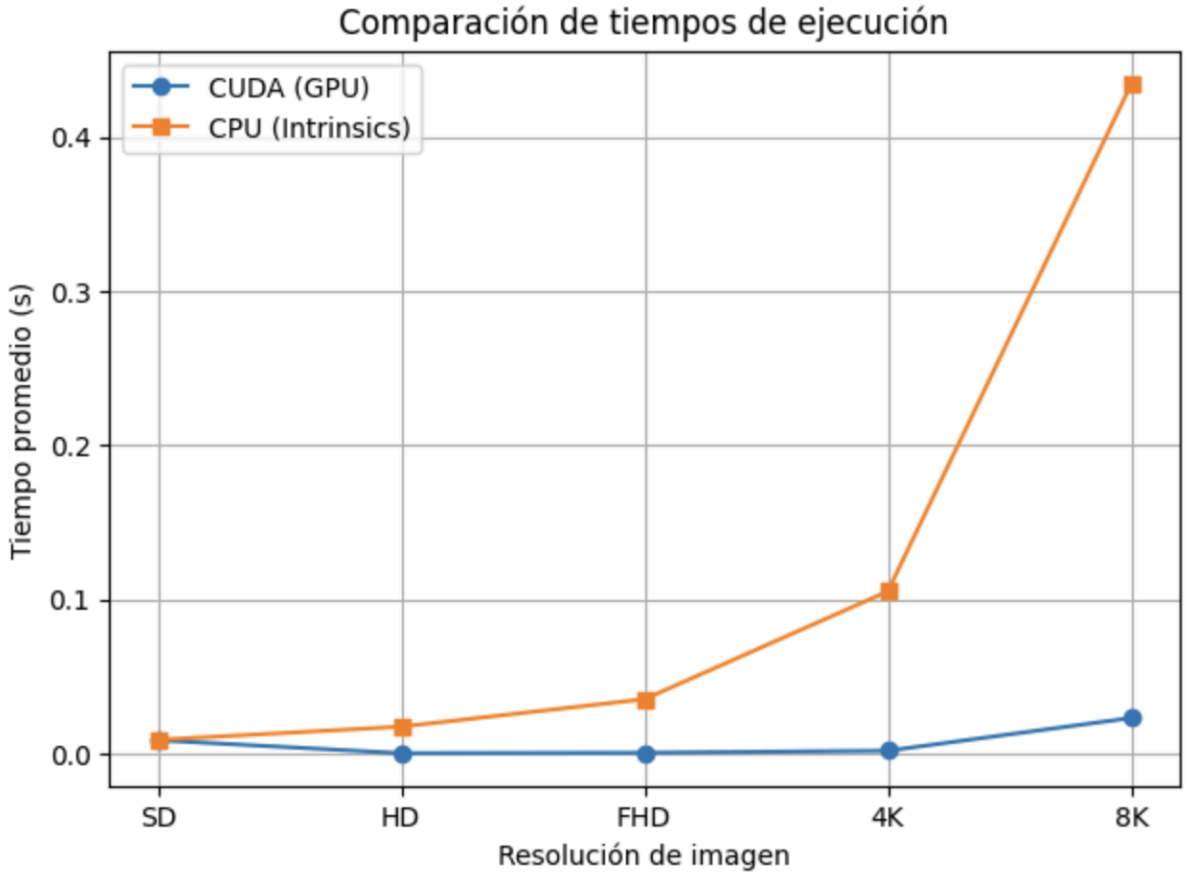
El tamaño máximo depende de la memoria disponible en la GPU. Si N es demasiado grande, las matrices A, B y C no caben en la memoria global del dispositivo. En

Colab, con una GPU T4 de 16 GB, normalmente el límite está entre N ≈ 6000 y 8000. Superar ese tamaño genera errores de asignación de memoria (cudaMalloc).

La GPU necesita copiar los datos desde la memoria del host (CPU) a la del dispositivo (GPU) y luego devolver los resultados, lo que añade un tiempo de transferencia que la CPU no tiene. Por eso, al comparar rendimientos, es justo incluir el tiempo de estas transferencias dentro del tiempo total de GPU, ya que forman parte del coste real del procesamiento en este entorno.

Ejercicio 3 : Procesamiento de imagen.

1. El código se encuentra en greyScaleGPU.cu
2.

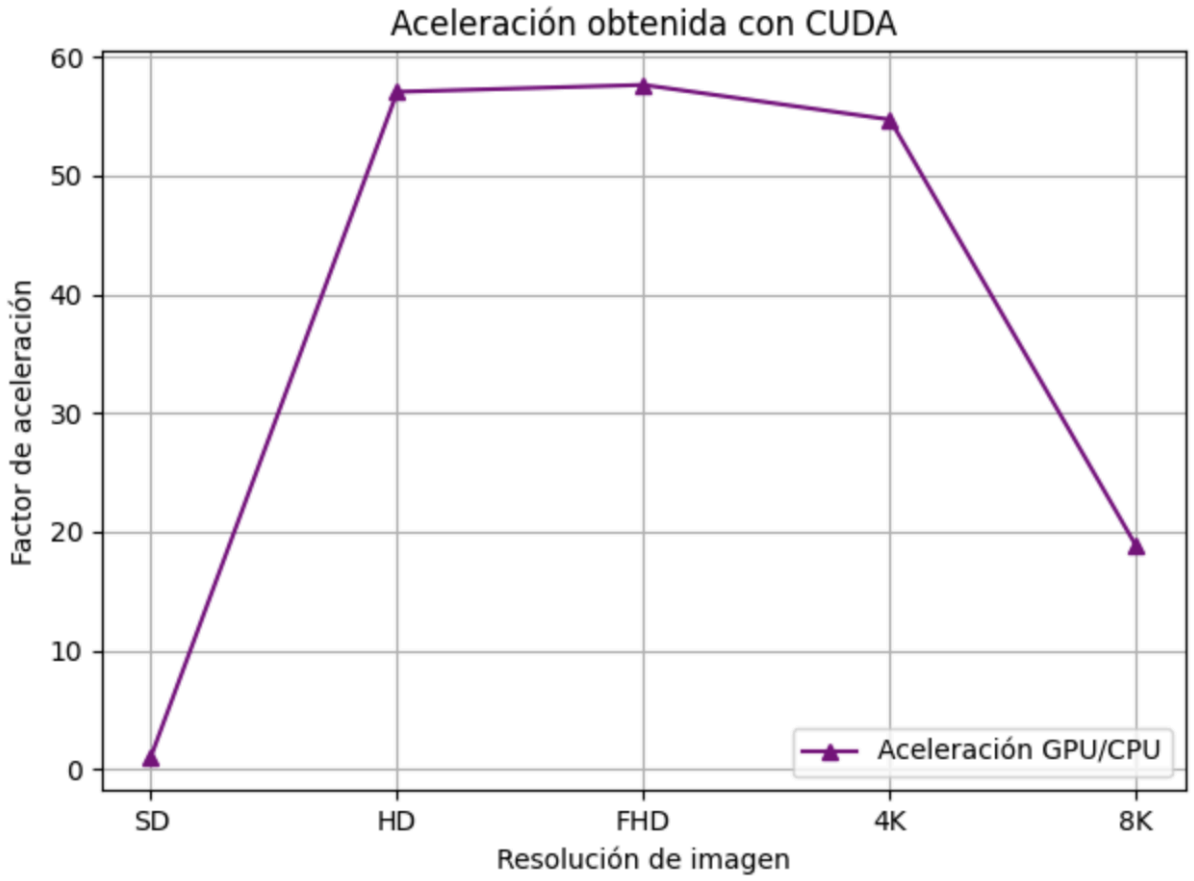


Los resultados obtenidos en los tiempos de ejecución evidencian de forma clara la superioridad de la implementación en GPU mediante CUDA frente a la versión optimizada manualmente con intrinsics en CPU para el procesamiento del algoritmo de conversión de imágenes RGB a escala de grises.

En la gráfica comparativa se aprecia que los tiempos de ejecución en GPU se mantienen muy bajos y prácticamente constantes, incluso al trabajar con resoluciones elevadas como 4K y 8K, mientras que los tiempos en CPU aumentan de manera exponencial a medida que crece el tamaño de la imagen.

Esta diferencia de rendimiento se explica porque las imágenes son estructuras de datos altamente paralelizables: cada píxel puede procesarse de manera independiente. En este contexto, la GPU aprovecha su arquitectura masivamente paralela, compuesta por miles de hilos distribuidos en múltiples multiprocesadores, lo que permite ejecutar operaciones simultáneas sobre grandes bloques de datos. Por el contrario, la CPU, aunque más versátil para tareas secuenciales o de control, dispone de un número mucho menor de núcleos e hilos, lo que limita su capacidad para abordar este tipo de procesamiento intensivo.

En consecuencia, la GPU resulta significativamente más eficiente para este tipo de operaciones gráficas y de cálculo paralelo, logrando una reducción drástica en los tiempos de ejecución y un mejor aprovechamiento del hardware disponible.

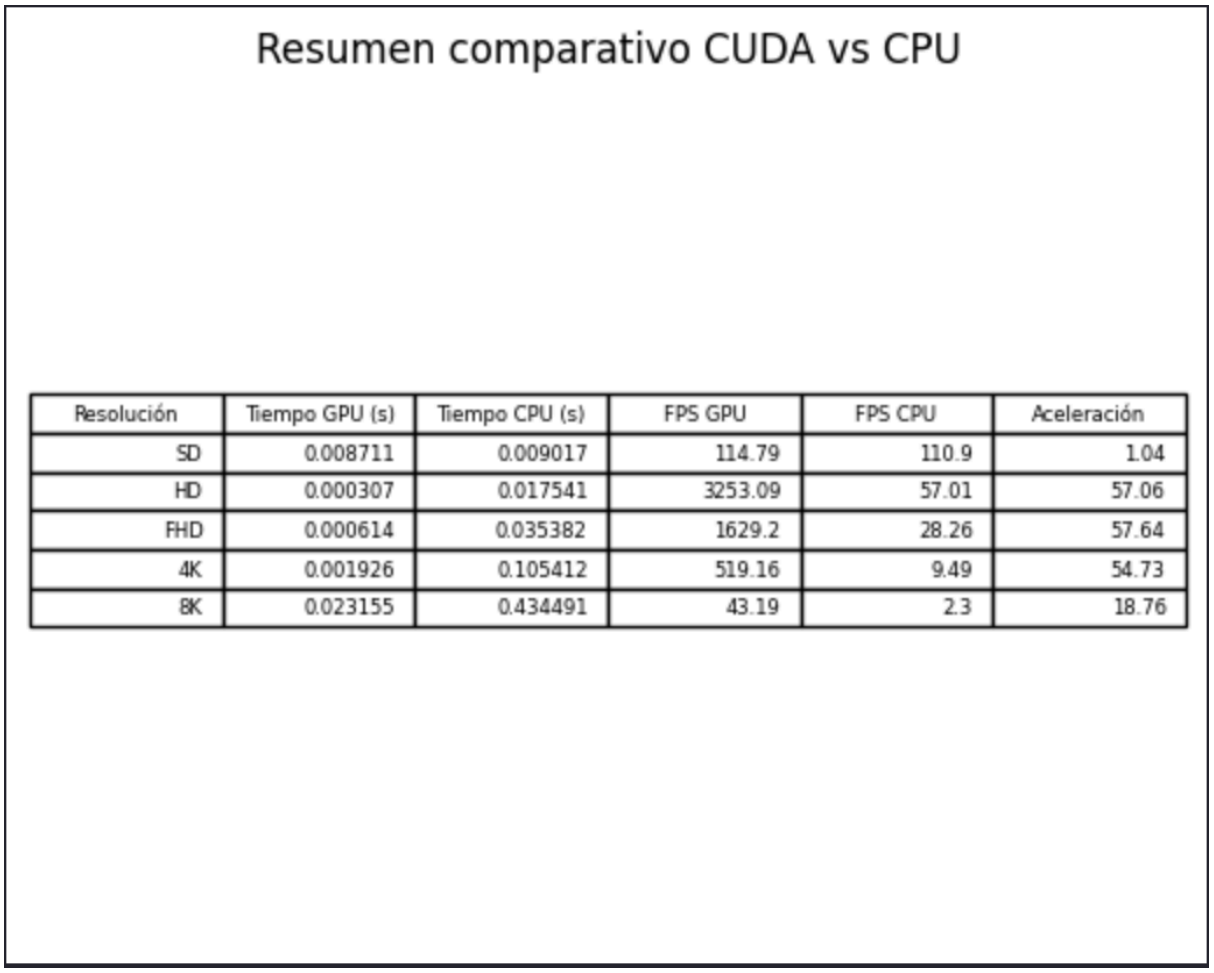


El factor de aceleración obtenido con la implementación en CUDA varía de forma significativa según la resolución de la imagen. Tal como se observa en la gráfica, la aceleración comienza siendo muy baja en resoluciones SD, pero aumenta bruscamente en HD

y FHD, donde alcanza su valor máximo, con un factor cercano a 58–59 veces más rápido que la ejecución en CPU.

A partir de esa resolución óptima, el rendimiento relativo de la GPU disminuye ligeramente en 4K y de forma más pronunciada en 8K, donde el factor de aceleración cae hasta alrededor de 20. Esta caída se debe a que el procesamiento de imágenes de muy alta resolución demanda una cantidad mucho mayor de memoria, lo que incrementa la latencia de transferencia de datos entre la memoria global y los multiprocesadores, además de provocar mayor presión sobre las cachés y los registros internos. Estos factores generan un cuello de botella que limita el aprovechamiento pleno del paralelismo de la GPU.

Pese a esta reducción en las resoluciones más altas, la aceleración sigue siendo muy notable en todos los casos, demostrando la eficiencia y escalabilidad de CUDA frente al procesamiento secuencial o parcialmente vectorizado de la CPU, especialmente en tareas masivamente paralelizables como la conversión de imágenes a escala de grises.



Resolución	Tiempo GPU (s)	Tiempo CPU (s)	FPS GPU	FPS CPU	Aceleración
SD	0.008711	0.009017	114.79	110.9	1.04
HD	0.000307	0.017541	3253.09	57.01	57.06
FHD	0.000614	0.035382	1629.2	28.26	57.64
4K	0.001926	0.105412	519.16	9.49	54.73
8K	0.023155	0.434491	43.19	2.3	18.76

En la tabla se observa claramente que la GPU supera ampliamente a la CPU en el número de imágenes procesadas por segundo (FPS), mostrando una diferencia abismal especialmente en las resoluciones HD, FHD y 4K. Por ejemplo, en FHD, la GPU alcanza 1629 FPS, mientras

que la CPU apenas llega a 28 FPS. Del mismo modo, en 4K, la GPU mantiene un rendimiento de 519 FPS, frente a los 9,49 FPS de la CPU.

Esta diferencia refleja la enorme capacidad de paralelización de la GPU, que le permite ejecutar miles de operaciones simultáneamente sobre los píxeles de la imagen, mientras que la CPU incluso con el uso de instrucciones vectoriales intrinsics se ve limitada por su menor número de núcleos y hilos concurrentes.

Aunque en resoluciones muy bajas (como SD) el rendimiento es similar, en cuanto aumenta la carga de trabajo, la eficiencia de la GPU crece exponencialmente, logrando factores de aceleración superiores a 50 veces respecto a la CPU. Esto evidencia que las GPU son la opción óptima para el procesamiento masivo de imágenes, donde las operaciones son altamente paralelizables y el rendimiento vectorial de la CPU resulta marginal en comparación.

3.

Para conseguir que el algoritmo funcione de forma óptima en la GPU, lo más importante es pensar la implementación desde la lógica de CUDA y aprovechar al máximo el paralelismo que ofrece. En este caso, lo ideal es asignar un hilo a cada píxel de la imagen, de manera que todos los píxeles se procesen al mismo tiempo. La imagen se guarda en memoria como un array lineal (ordenada por filas), lo que permite que los hilos accedan a posiciones contiguas de memoria global. Esto hace que las lecturas y escrituras sean más rápidas y eficientes, ya que los accesos están “coalescidos”, algo fundamental para aprovechar bien el ancho de banda de la GPU.

El cálculo de la escala de grises se hace usando la fórmula estándar que combina los valores RGB con los coeficientes 0.299, 0.587 y 0.114. Estos coeficientes se pueden guardar en memoria constante para que todos los hilos los consulten rápidamente sin recargar la memoria global. Además, se evita usar ifs dentro del kernel para que todos los hilos ejecuten el mismo flujo de instrucciones y no haya divergencias que ralenticen la ejecución. En cuanto al tamaño de los bloques, se suelen usar configuraciones como 16x16 o 32x8 hilos, que suelen dar buena ocupación de los multiprocesadores, y se calcula el grid de forma que cubra toda la imagen sin dejar píxeles fuera.

Otro aspecto importante es minimizar las transferencias entre la CPU y la GPU, ya que son una de las partes más lentas del proceso. Lo mejor es copiar la imagen completa a la GPU una sola vez, ejecutar el kernel y luego traer de vuelta solo el resultado final en escala de grises. Si se quieren procesar muchas imágenes seguidas, se pueden usar streams y memoria pinned para solapar las transferencias de datos con la ejecución del kernel y ahorrar tiempo. En resumen, este enfoque aprovecha la

capacidad de la GPU para ejecutar miles de hilos en paralelo, accediendo a la memoria de forma eficiente y reduciendo las transferencias innecesarias, lo que permite alcanzar un rendimiento muy superior al de la CPU.