Programación en GPUs con CUDA
Laboratorio 1 — Caracterización experimental del dispositivo y primeros experimentos

Introducción

En esta primera sesión de laboratorio se pretende que el estudiante interactúe con una GPU real y adquiera una comprensión **experimental** del ecosistema CUDA: hardware disponible, compilación, medición rigurosa de tiempos, y el coste de mover datos entre CPU y GPU.

En una aplicación real, el rendimiento depende tanto del cómputo como del movimiento de datos. En este laboratorio, por tanto, distinguiremos explícitamente entre:

- tiempo de **cómputo** (kernel en GPU),
- tiempo de **transferencias** (CPU/GPU y GPU/GPU),
- tiempo **total** (transferencias + kernel + sincronizaciones necesarias).

Recursos oficiales (lectura y consulta durante la práctica):

- CUDA Programming Guide: https://docs.nvidia.com/cuda/cuda-c-programming-guide/
- CUDA Runtime API (referencia de funciones): https://docs.nvidia.com/cuda/cuda-runtime-api/
- CUDA Best Practices Guide: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/
- Nsight Systems (timeline): https://docs.nvidia.com/nsight-systems/
- Nsight Compute (métricas kernel): https://docs.nvidia.com/nsight-compute/

1. Ejercicio 1 — Exploración y caracterización del dispositivo

Objetivo

Comprender que la GPU es un dispositivo físico independiente, con memoria dedicada, consumo dinámico y límites arquitecturales que condicionan cómo se programa y cómo se mide el rendimiento.

Este ejercicio combina tres vistas:

A. inspección **estática** con `nvidia-smi`,

B. observación **dinámica** durante un kernel,

C. consulta **programática** de propiedades con la API CUDA.

1.1. Parte A — Inspección estática con `nvidia-smi`

Ejecuta en terminal:

```
nvidia-smi
```

y anota (al menos) la siguiente información, indicando claramente de dónde la extraes:

- Modelo exacto de GPU.
- Memoria total y memoria actualmente en uso.
- Versión del driver NVIDIA.
- Estado de utilización (*GPU utilization*) y consumo/potencia.
- Si aparece, información relacionada con PCIe (p. ej. generación/anchura) o con clocks.

Cuestiones

1. ¿La memoria que muestra `nvidia-smi` es la RAM del sistema o memoria dedicada de la GPU? (recuerda la orden `free` para observar la cantidad de memoria RAM en tu sistema).
2. ¿Qué significa exactamente *GPU utilization*? ¿Puede ser baja aunque haya memoria ocupada?
3. ¿Qué te sugiere el hecho de que se muestre potencia/consumo? ¿Por qué es relevante en sistemas equipados con GPU?

Parte A.2 — Inspección detallada con `nvidia-smi -q`

Ejecuta:

```
nvidia-smi -q
```

Busca y anota cualquier dato disponible sobre:

- capacidad de memoria y tipo (si aparece),
- clocks (gráficos/memoria),
- información del enlace PCIe (si aparece).

1.2. Parte B — Observación dinámica: ejecución de un kernel básico

Ahora vamos a observar cómo cambia `nvidia-smi` cuando realmente hay trabajo en GPU.

Paso 1: monitorización en tiempo real

En una terminal, ejecuta:

```
watch -n 0.5 nvidia-smi
```

Paso 2: copiar/pegar y compilar el siguiente programa CUDA

Crea un fichero `busy.cu` con este contenido (tal cual) y compílalo.

```
// busy.cu: kernel deliberadamente costoso para mantener la GPU ocupada
#include <cstdio>
#include <cmath>
#include <cuda_runtime.h>

__global__ void busy_kernel(float *x, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) {
        float v = x[i];
        // Carga computacional: muchas operaciones transcendentes
        // (suficiente para que se note en nvidia-smi)
        for (int k = 0; k < 20000; ++k) {
            v = sinf(v) + cosf(v);
        }
        x[i] = v;
    }
}

int main()
{
    const int N = 1 << 24;
    float *d = nullptr;

    cudaMalloc((void**)&d, N * sizeof(float));

    dim3 block(256);
    dim3 grid((N + block.x - 1) / block.x);

    busy_kernel<<<grid, block>>>(d, N);
    cudaDeviceSynchronize();

    cudaFree(d);
    return 0;
}
```

Compila con:

```
nvcc -O3 -lineinfo busy.cu -o busy
```

Ejecuta `./busy` mientras observas el `watch`.

Cuestiones

1. ¿Qué campos cambian (utilización, memoria, potencia, clocks)?
2. ¿Por qué un kernel "que tarda" ayuda a ver actividad en `nvidia-smi`?
3. ¿Qué esperas que ocurra si reduces N drásticamente? ¿Y si reduces el bucle interno?

1.3. Parte C — Consulta programática de propiedades (API CUDA)

En esta parte, vas a obtener las características del dispositivo **desde código**, lo cual es fundamental en aplicaciones reales (selección de dispositivo, parametrización, logging reproducible).

Enunciado

Implementa un programa `query.cu` que:

- enumere todos los dispositivos CUDA disponibles,
- imprima, para cada dispositivo, al menos:
  - nombre, compute capability,
  - número de SMs,
  - `warpSize`,
  - memoria global total,
  - `sharedMemPerBlock`,
  - `regsPerBlock`,
  - `maxThreadsPerBlock`,
  - `maxThreadsPerMultiProcessor`.

Pistas (fragmento de solución)

El patrón mínimo a utilizar es:

```
int ndev = 0;
cudaGetDeviceCount(&ndev);

for (int d = 0; d < ndev; ++d) {
  cudaDeviceProp p;
  cudaGetDeviceProperties(&p, d);
  // printf("Device %d: %s\n", d, p.name);
  // printf("Compute capability: %d.%d\n", p.major, p.minor);
  // ...
}
```

Compila con:

```
nvcc -O2 query.cu -o query
```

Invitación a ampliar

Además de los campos anteriores, el struct `cudaDeviceProp` contiene muchos más. Se pide que el estudiante **busque en la documentación oficial** al menos **tres campos adicionales** relevantes para rendimiento/limitaciones, y los añada a la salida (por ejemplo: límites de grid, tamaño de L2 si aparece, propiedades de la memoria, etc.).

Cuestiones

1. ¿Qué significa `warpSize` y por qué aparece como propiedad explícita?
2. ¿Qué recursos (a nivel de SM) sugieren `sharedMemPerBlock` y `regsPerBlock`?
3. ¿Qué información de `query.cu` te será útil para razonar sobre ocupación en sesiones posteriores?
4. Cita (con enlace) la sección de la documentación donde se describe `cudaGetDeviceProperties` o `cudaDeviceProp`.

2. Ejercicio 2 — Caracterización de transferencias CPU/GPU y GPU/GPU

Objetivo

Medir experimentalmente la latencia y el ancho de banda efectivo de transferencias:

- Host → Device (H→D),
- Device → Host (D→H),
- Device → Device (D→D),

y estudiar dos fenómenos fundamentales:

- dependencia con el tamaño (latencia vs saturación de BW),
- asimetría práctica del bus PCIe (H→D vs D→H).

Metodología experimental (obligatoria)

Para que los resultados tengan sentido:

1. **Warm-up**: antes de medir una copia, ejecútala una vez sin medir (calienta el path de DMA, asignaciones internas, etc.).
2. **Repeticiones**: mide al menos 10 veces y usa la **mediana** (hay ruido del sistema).
3. **Eventos CUDA**: mide con `cudaEventRecord`/`cudaEventElapsedTime` (no con `std::chrono` para GPU).
4. **Separación de casos**: mide por separado pageable vs pinned, y D→D.

Tamaños

Evalúa una lista de tamaños (bytes) que cubra desde latencia hasta saturación, por ejemplo:

$$1\,\text{KB},\ 4\,\text{KB},\ 16\,\text{KB},\ 64\,\text{KB},\ 1\,\text{MB},\ 16\,\text{MB},\ 64\,\text{MB},\ 256\,\text{MB}$$

Enunciado paso a paso (con rutinas a usar)

1. Reserva dos buffers en host (pageable) con `malloc` y dos buffers en device con `cudaMalloc`. (`cudaMalloc`, `cudaFree`)
2. Implementa la medición con eventos CUDA. (`cudaEventCreate`, `cudaEventRecord`, `cudaEventSynchronize`, `cudaEventElapsedTime`)
3. Para cada tamaño, mide H→D y D→H con `cudaMemcpy`. (`cudaMemcpy`)
4. Repite el experimento usando host pinned con `cudaMallocHost` / `cudaFreeHost`. (`cudaMallocHost`, `cudaFreeHost`)
5. Mide D→D copiando entre dos buffers de device. (`cudaMemcpyDeviceToDevice`)

Ejemplo breve de temporización (plantilla)

Se recomienda encapsular la zona a medir. Ejemplo conceptual:

```
// Medir una "zona" concreta: aqui una copia cudaMemcpy
cudaEventRecord(start);
cudaMemcpy(d, h, bytes, cudaMemcpyHostToDevice);
cudaEventRecord(stop);

cudaEventSynchronize(stop);
cudaEventElapsedTime(&ms, start, stop);
```

Resultados y análisis obligatorios

- Completa una tabla con BW (GB/s) para cada tamaño y cada caso: pageable H→D, pageable D→H, pinned H→D, pinned D→H, D→D.
- Representa gráficamente:
  - BW vs tamaño (curvas separadas),
  - comparación directa de H→D vs D→H (asimetría),
  - pageable vs pinned.
- Identifica en qué tamaño aparece la saturación (BW aproximadamente constante).

Cuestiones

1. ¿Por qué BW es bajo en tamaños pequeños? Relaciónalo con latencia y overhead fijo.
2. ¿A partir de qué tamaño se estabiliza BW en tu sistema?
3. ¿Hay asimetría (H→D $\neq$ D→H)? ¿Cuánto difieren aproximadamente en el régimen de saturación?
4. ¿Por qué pinned suele mejorar BW?
5. ¿Por qué D→D es muy superior a CPU/GPU en general?

3. Ejercicio 3 — Suma de vectores con tamaños crecientes (CPU vs GPU)

Objetivo

Evaluar empíricamente cuándo la GPU proporciona aceleración respecto a CPU, y separar cuidadosamente el coste de transferencias del coste del kernel.

Enunciado

Implementa la operación:

$$C[i] = A[i] + B[i]$$

en CPU (secuencial) y en GPU (kernel paralelo).

No se evaluará un único tamaño, sino una familia de tamaños crecientes. Por ejemplo:

$$N \in \{2^{10}, 2^{12}, 2^{14}, \ldots, 2^{26}\}$$

Mediciones obligatorias

Para cada tamaño N:

- Tiempo CPU del bucle secuencial.
- Tiempo GPU del kernel (solo kernel).
- Tiempo GPU total (H→D + kernel + D→H).

Se recomienda medir kernel con eventos CUDA, y CPU con `std::chrono` en C++ o cualquier rutina de tipo `time` en C.

Representación obligatoria

Construye al menos estas gráficas (una por figura):

- Tiempo vs N (CPU y GPU total).
- Tiempo kernel vs N (solo GPU kernel).
- Throughput efectivo (GB/s) vs N para CPU y GPU (total y/o kernel).
- Speedup vs N (CPU / GPU total).

Cuestiones

1. ¿Existe un tamaño mínimo a partir del cual la GPU gana claramente? ¿Por qué?
2. ¿Qué domina el tiempo total GPU: transferencias o kernel? ¿Cómo cambia con N?
3. Compara tus conclusiones con los resultados del Ejercicio 2 (PCIe). ¿Son consistentes?

A. Apéndice A — APIs CUDA usadas (prototipos y descripción breve)

Gestión de dispositivo

```
cudaError_t cudaGetDeviceCount(int *count);
```

Devuelve el número de dispositivos CUDA visibles en el sistema.

```
cudaError_t cudaGetDeviceProperties(cudaDeviceProp *prop, int device);
```

Rellena la estructura `cudaDeviceProp` con características del dispositivo (`SMs`, `warpSize`, límites, memoria, etc.).

```
cudaError_t cudaGetDevice(int *device);
```

Devuelve el dispositivo actualmente seleccionado por el proceso.

Gestión de memoria

```
cudaError_t cudaMalloc(void **devPtr, size_t size);
```

Reserva memoria en el dispositivo (memoria global).

```
cudaError_t cudaFree(void *devPtr);
```

Libera memoria previamente reservada con `cudaMalloc`.

```
cudaError_t cudaMallocHost(void **ptr, size_t size);
```

Reserva memoria host **pinned** (page-locked), útil para mejorar transferencias CPU/GPU por DMA.

```
cudaError_t cudaFreeHost(void *ptr);
```

Libera memoria host reservada con `cudaMallocHost`.

Transferencias

```
cudaError_t cudaMemcpy(void *dst,
                       const void *src,
                       size_t count,
                       cudaMemcpyKind kind);
```

Copia memoria entre espacios host/device según `kind`: `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`.

Eventos y temporización

```
cudaError_t cudaEventCreate(cudaEvent_t *event);
```

Crea un evento CUDA (marca temporal en un stream).

```
cudaError_t cudaEventRecord(cudaEvent_t event, cudaStream_t stream);
```

Inserta el evento en un stream; el evento se "dispara" cuando el stream alcanza ese punto.

```
cudaError_t cudaEventSynchronize(cudaEvent_t event);
```

Bloquea hasta que el evento se haya completado (el stream haya alcanzado el evento).

```
cudaError_t cudaEventElapsedTime(float *ms,
                                 cudaEvent_t start,
                                 cudaEvent_t end);
```

Devuelve el tiempo transcurrido (en ms) entre dos eventos.

Sincronización global

```
cudaError_t cudaDeviceSynchronize(void);
```

Bloquea el host hasta que todas las operaciones pendientes en el dispositivo hayan finalizado.

B. Apéndice B — Enlaces a documentación oficial

- CUDA Programming Guide: https://docs.nvidia.com/cuda/cuda-c-programming-guide/
- CUDA Runtime API: https://docs.nvidia.com/cuda/cuda-runtime-api/
- CUDA Best Practices Guide: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/
- Nsight Systems: https://docs.nvidia.com/nsight-systems/
- Nsight Compute: https://docs.nvidia.com/nsight-compute/