CAGE – Grupo de Investigación

SC3 – Super Cómputo y Cálculo Científico – UIS

@sc3uis

Programación de GPU con CUDA

Heterogeneous computing: de la CPU a la GPU

Módulo 6 – Semillero CAGE

Luis Alejandro Torres Niño | sc3.operaciones@uis.edu.co

Objetivos del módulo

- Entender por qué las GPU aceleran el cómputo paralelo: **throughput** frente a **latency**.
- Conocer el modelo **host/device** y la jerarquía **grid** → **block** → **thread**.
- Escribir, lanzar y verificar *kernels* en CUDA C++; manejar errores y medir con `cudaEvent`.
- Usar la jerarquía de memoria: **global**, **shared** y registros; **coalescing** y *tiling*.
- Entender **warps**, *divergence*, *occupancy*, *streams* y el costo de las transferencias.
- Ejecutar en la partición `gpu_titan` del cluster con SLURM.

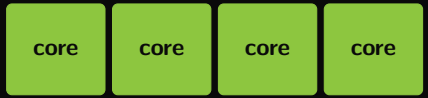
Agenda

1. Heterogeneous computing y arquitectura de la GPU
2. Primer programa y entorno en el cluster
3. Jerarquía de hilos: grid, block, thread, warp
4. Suma de vectores: el flujo completo y el manejo de errores
5. Jerarquía de memoria: coalescing y shared memory
6. Multiplicación de matrices: naive y tiled
7. Rendimiento: occupancy, streams y herramientas
8. CUDA en el cluster; MPI + CUDA
9. Retos individuales y reto grupal

1
Heterogeneous computing y arquitectura de la GPU

Latency vs throughput

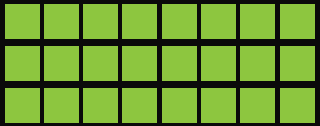
CPU: pocos núcleos potentes



caché grande (L1/L2/L3) + control complejo

optimiza el **tiempo** de una tarea
(baja latency)

GPU: miles de núcleos simples



caché pequeña + control simple

optimiza **cuántas** tareas por segundo
(alto throughput)

La GPU dedica casi todos sus transistores a **unidades aritméticas**; la CPU, a reducir la latencia de un solo hilo. Una aplicación usa ambas: la CPU para lo secuencial y la GPU para la parte **masivamente paralela**.

SIMD, SIMT y la taxonomía de Flynn

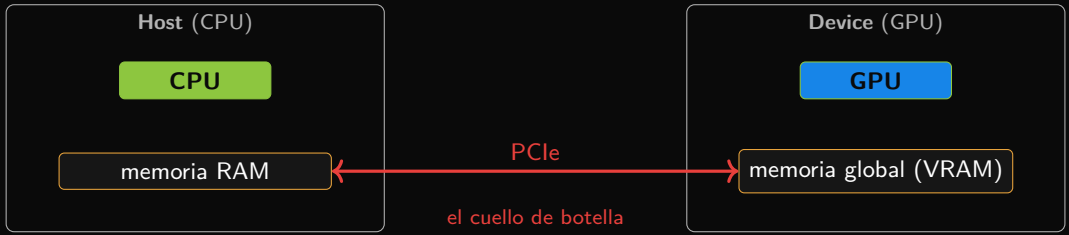
- La taxonomía de Flynn clasifica la arquitectura según *streams* de instrucciones y de datos: SISD, **SIMD**, MISD, MIMD.
- **SIMD**: una instrucción opera sobre varios datos en un registro vectorial.
- **SIMT** (*Single Instruction, Multiple Threads*): varios **hilos** ejecutan la **misma instrucción** sobre datos distintos, pero cada hilo tiene su propio flujo lógico (puede divergir).

★ Relación con lo visto

SPMD (MPI) es un modelo de programación; **SIMD** (`omp simd`) es paralelismo de hardware en un núcleo; **SIMT** es el modelo de ejecución de la GPU: escribes código para **un hilo** y se lanzan miles.

Referencia: M. Flynn, *Some computer organizations and their effectiveness*, IEEE Trans. Computers, 1972.

El modelo host / device



- CUDA es un modelo **heterogéneo**: la CPU (**host**) controla y la GPU (**device**) ejecuta los *kernels*.
- Cada uno tiene su **memoria propia**; los datos viajan por PCIe (o NVLink): **copiar es caro**.
- Los *kernels* se lanzan de forma **asíncrona**: el host sigue mientras la GPU trabaja.

Dentro de la GPU: streaming multiprocessors



- ➔ Una GPU se compone de varios **SM** (*streaming multiprocessors*); cada SM tiene muchos núcleos CUDA, registros y **shared memory**.
- ➔ Un *block* de hilos se ejecuta **completo en un solo SM**; un SM puede alojar varios blocks.
- ➔ El número de SM, la memoria y la *compute capability* dependen del modelo: consúltalos con `nvidia-smi` (modelo y memoria) y `cudaGetDeviceProperties` (SM).

El entorno de CUDA en el cluster

```
ssh usuario@toctoc.sc3.uis.edu.co          # bastion
ssh guane                                   # frontend (sin GPU)

sinfo -p gpu_titan                          # nodos con GPU
module avail cuda                           # cuda/11.8, ...
module load cuda/11.8
nvcc --version

srun -p gpu_titan --gres=gpu:1 --pty bash   # shell en un nodo con 1 GPU
nvidia-smi                                  # modelo de GPU, memoria, procesos
```

- El **frontend no tiene GPU**: compila allí si quieres, pero ejecuta en un nodo de `gpu_titan`.
- Confirma el nombre del recurso con `scontrol show node felix`.

★ Compute capability

`nvidia-smi --query-gpu=name,compute_cap --format=csv` te da el valor para `-arch=sm_XX` (el campo `compute_cap` solo existe en drivers recientes; si falla usa

Hola mundo en CUDA

```cpp
#include <cstdio>

__global__ void hola() {                        // kernel: corre en la GPU
    printf("Hola desde el bloque %d, hilo %d\n", blockIdx.x, threadIdx.x);
}

int main() {
    hola<<<2, 4>>>();          // <<<bloques, hilos>>>: 2 x 4 = 8 hilos
    cudaDeviceSynchronize();  // espera a que el kernel termine
    return 0;
}
```

```
nvcc -o hola hola.cu              # nvcc compila host y device
srun -p gpu_titan --gres=gpu:1 ./hola
```

`__global__` marca un *kernel*: se llama desde el host y corre en el device. Sin `cudaDeviceSynchronize()` el programa podría terminar antes de que la GPU imprima.

Experimentos con la configuración de lanzamiento

Experimento 1: más bloques

➔ Cambia `<<<2,4>>>` por `<<<4,4>>>` y luego `<<<8,4>>>`. ¿Cuántas líneas se imprimen? ¿Qué cambia en cada una?

Experimento 2: más hilos

➔ Cambia por `<<<1,8>>>` y `<<<2,64>>>`. ¿En qué se parece a los bloques?

★ Observa

El **orden** de impresión no está garantizado: bloques e hilos se ejecutan en paralelo. Piensa siempre en un hilo, y en que **cualquier orden es posible**.

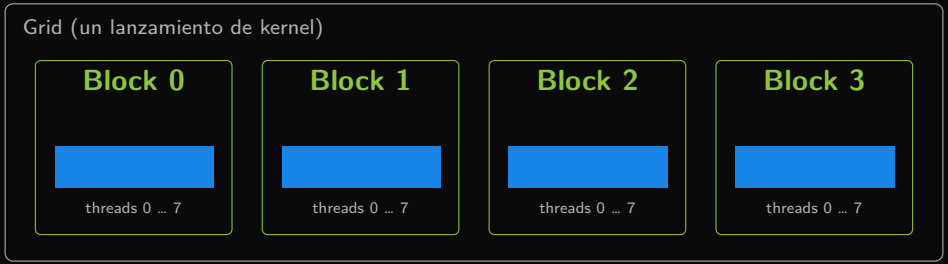
Reto individual 1: tu primera GPU

■ Reto 1 – Hola GPU

1. Compila `hola.cu` con `nvcc` y ejecútalo en un nodo de `gpu_titan`.
2. Ejecuta `nvidia-smi`: anota modelo de GPU y memoria total; obtén la *compute capability* con `nvidia-smi --query-gpu=compute_cap --format=csv` o con `cudaGetDeviceProperties`.
3. Prueba `<<<4,8>>>` y `<<<1,64>>>`; cuenta las líneas impresas y explica el total.
4. Escribe un programa que use `cudaGetDeviceProperties` e imprima: nombre, número de SM, memoria global, hilos máximos por bloque y *warp size*.

2
Jerarquía de hilos

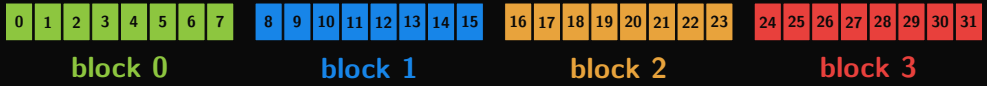
Grid, block y thread



`threadIdx.x`	índice del hilo **dentro** de su bloque.
`blockIdx.x`	índice del bloque dentro del grid.
`blockDim.x`	hilos por bloque.
`gridDim.x`	bloques en el grid.

Índice global de un hilo

32 elementos = 4 bloques × 8 hilos; cada celda es el **índice global** `i`




```
int i = blockIdx.x * blockDim.x + threadIdx.x;    // bloque 2, hilo 3 -> 2*8+3 =
    19
```

- Escenario A: 32 elementos con 4 bloques de 8 hilos. Escenario B: 8 bloques de 4 hilos: **mismos índices**, otra configuración.
- Cada hilo calcula **su** índice y procesa **su** elemento.

Configurar el lanzamiento

```
int n = 1000000;
int hilos = 256;                          // multiplo de 32 (tipico 128 a 512)
int bloques = (n + hilos - 1) / hilos; // division techo: cubre n
kernel<<<bloques, hilos>>>(d_a, n);

__global__ void kernel(float* a, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) a[i] = 2.0f * a[i];      // guarda: sobran hilos
}
```

- ➔ Máximo **1024 hilos por bloque**; el número de bloques en el grid es mucho mayor.
- ➔ Sin la guarda `if (i < n)`, los hilos sobrantes escribirían **fuera del arreglo**.
- ➔ `dim3` permite grids/bloques en 2D y 3D: `dim3 bloque(16,16);`

Grid-stride loop

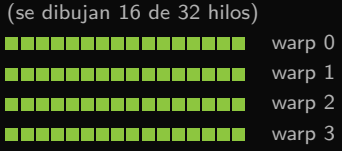
```cuda
__global__ void saxpy(int n, float a, const float* x, float* y) {
    // cada hilo procesa varios elementos
    int paso = gridDim.x * blockDim.x;
    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < n; i += paso)
        y[i] = a * x[i] + y[i];
}

saxpy<<<128, 256>>>(n, 3.0f, x, y);   // 32768 hilos, n = 1M
```

- Funciona para **cualquier** *n* y cualquier tamaño de grid; el kernel es más reutilizable y flexible.
- Útil cuando *n* supera el máximo de bloques o quieres limitar el grid al tamaño de la GPU.

Warps: la unidad real de ejecución

- El SM agrupa los hilos de un bloque en **warps** de **32 hilos consecutivos**.
- Los 32 hilos de un warp ejecutan la **misma instrucción** a la vez (SIMT).
- Por eso conviene que los bloques sean **múltiplos de 32**.



Divergence dentro de un warp

```
// SIN divergence: la rama la decide el
    warp entero
y[i] = (i / 32) % 2 ? x[i] * 2.0f : x[i] +
    1.0f;

// CON divergence: pares e impares en el
    mismo warp
y[i] = (i % 2)      ? x[i] * 2.0f : x[i] +
    1.0f;
```



Cuando los hilos de un warp toman **ramas distintas**, el hardware ejecuta **ambas en serie**, con los hilos inactivos enmascarados (en gris): se pierde paralelismo. La *divergence* entre warps distintos **no** cuesta.

Reto individual 2: índices y geometría

■ Reto 2 – Dibuja los índices

1. Escribe un kernel que escriba en un arreglo $a[i] = i$ usando `<<<4,8>>>` y verifícalo en la CPU.
2. Cámbialo a `<<<8,4>>>` y a `<<<2,16>>>`: ¿cambia el resultado?
3. Para $n = 1000$ y bloques de 256 hilos, ¿cuántos bloques necesitas? ¿Cuántos hilos **sobran** y cómo los proteges?
4. Escribe un kernel **2D** (`dim3`) que llene una matriz $A[i][j] = i \cdot N + j$ y verifícalo.

3
Suma de vectores: el flujo completo

Los seis pasos de un programa CUDA




```
float *d_a, *d_b, *d_c;
cudaMalloc(&d_a, bytes);  cudaMalloc(&d_b, bytes);  cudaMalloc(&d_c, bytes);
    // 1
cudaMemcpy(d_a, h_a, bytes, cudaMemcpyHostToDevice);
                                  // 2
cudaMemcpy(d_b, h_b, bytes, cudaMemcpyHostToDevice);
suma<<<bloques, hilos>>>(d_a, d_b, d_c, n);
                                            // 3
cudaDeviceSynchronize();
                                                                  // 4
cudaMemcpy(h_c, d_c, bytes, cudaMemcpyDeviceToHost);
                                  // 5
cudaFree(d_a);  cudaFree(d_b);  cudaFree(d_c);
                                        // 6
```

Suma de vectores: programa completo (1/2)

```
#define CUDA_CHECK(llamada)                                                \
    do { cudaError_t e = (llamada);                                        \
         if (e != cudaSuccess) {                                           \
             std::fprintf(stderr, "CUDA error %s en %s:%d\n",              \
                          cudaGetErrorString(e), __FILE__, __LINE__);      \
             std::exit(1); } } while (0)

__global__ void suma(const float* a, const float* b, float* c, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) c[i] = a[i] + b[i];
}
```

❄ Siempre comprueba errores

Las llamadas CUDA **no abortan** si fallan: devuelven un código. Envuelve cada llamada en `CUDA_CHECK`. Tras un *kernel*, usa `cudaGetLastError()` (error de lanzamiento) y `cudaDeviceSynchronize()` (error de ejecución).

Suma de vectores: programa completo (2/2)

```
const int n = 1 << 24;  const size_t bytes = n * sizeof(float);
// ... malloc de h_a, h_b, h_c y relleno ...
CUDA_CHECK(cudaMalloc(&d_a, bytes));  CUDA_CHECK(cudaMalloc(&d_b, bytes));
    CUDA_CHECK(cudaMalloc(&d_c, bytes));
CUDA_CHECK(cudaMemcpy(d_a, h_a, bytes, cudaMemcpyHostToDevice));
CUDA_CHECK(cudaMemcpy(d_b, h_b, bytes, cudaMemcpyHostToDevice));

int hilos = 256, bloques = (n + hilos - 1) / hilos;
cudaEvent_t t0, t1;  cudaEventCreate(&t0);  cudaEventCreate(&t1);
cudaEventRecord(t0);
suma<<<bloques, hilos>>>(d_a, d_b, d_c, n);
cudaEventRecord(t1);
CUDA_CHECK(cudaGetLastError());
CUDA_CHECK(cudaEventSynchronize(t1));
float ms = 0;  cudaEventElapsedTime(&ms, t0, t1);                  // tiempo del
    kernel en ms

CUDA_CHECK(cudaMemcpy(h_c, d_c, bytes, cudaMemcpyDeviceToHost));
// verificar en la CPU, cudaFree(...), free(...)
```

Unified Memory: menos código

```cpp
float *a, *b, *c;
cudaMallocManaged(&a, n * sizeof(float));   // visible en CPU y GPU
cudaMallocManaged(&b, n * sizeof(float));
cudaMallocManaged(&c, n * sizeof(float));
for (int i = 0; i < n; ++i) { a[i] = 1.0f; b[i] = 2.0f; }   // la CPU escribe

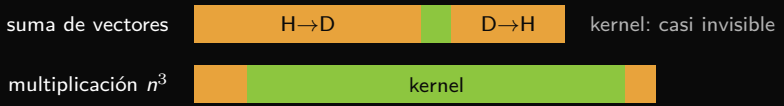
suma<<<(n + 255) / 256, 256>>>(a, b, c, n);
cudaDeviceSynchronize();                    // obligatorio antes de leer

std::printf("c[0] = %.1f\n", c[0]);
cudaFree(a); cudaFree(b); cudaFree(c);
```

- El driver migra las páginas entre host y device **bajo demanda**: sin `cudaMemcpy` explícitos.
- Muy cómodo para empezar; en rendimiento puro, copiar explícitamente (y con *streams*) suele dar más control.

Referencia: M. Harris, *Unified Memory for CUDA Beginners*, NVIDIA Technical Blog.

Las transferencias pueden dominar



- Una operación como $c = a + b$ hace **1 suma por 12 bytes** movidos: **limitada por memoria y por PCIe**.
- Una multiplicación de matrices hace $O(n^3)$ operaciones sobre $O(n^2)$ datos: la **intensidad aritmética** compensa la copia.
- **Regla**: mueve los datos a la GPU **una vez**, haz mucho cómputo allí y trae solo el resultado.

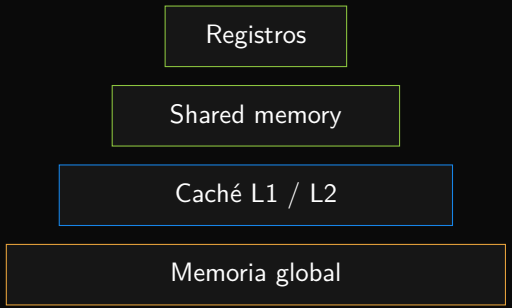
Reto individual 3: SAXPY y errores

■ Reto 3 – SAXPY con comprobación de errores

1. Implementa $y = a\,x + y$ con `cudaMalloc`/`cudaMemcpy` y la macro `CUDA_CHECK`.
2. Verifica el resultado en la CPU con tolerancia y mide el kernel con `cudaEvent`.
3. Provoca **tres errores**: pedir más de 1024 hilos por bloque, olvidar `cudaMemcpy` de vuelta, y acceder fuera de rango. ¿Qué reporta CUDA en cada caso?
4. Mide el tiempo **total** (con copias) y el del **kernel**: ¿qué porcentaje es transferencia?
5. Repite con `cudaMallocManaged` y compara.

4
Jerarquía de memoria

Los tipos de memoria de la GPU

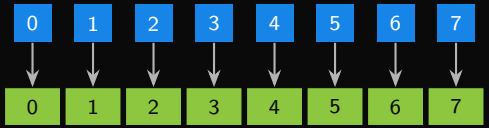


	Alcance	Velocidad
Registros	un hilo	más rápida
Shared memory	un bloque	muy rápida (en el SM)
Caché L1 / L2	automática	rápida
Memoria global	todos los hilos + host	lenta (cientos de ciclos)

Además: **constant memory** (solo lectura, con caché) y **local memory** (desbordes de registros, vive en la global). La **global** es grande pero lenta: gran parte de la optimización es **evitar accesos innecesarios** a ella.

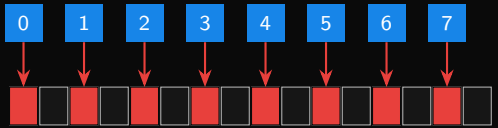
Coalescing: cómo se accede a la memoria global

Coalesced: hilos consecutivos → direcciones consecutivas



1 transacción

Strided: hilos consecutivos → direcciones separadas



varias transacciones

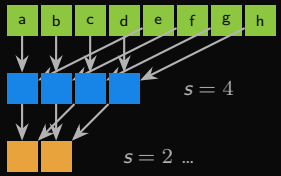
```
out[i] = in[i];                    // coalesced
out[i * salto] = in[i * salto];    // strided: ancho de banda desperdiciado
```

Los 32 hilos de un warp acceden juntos: si las direcciones son **contiguas** se atienden con pocas transacciones; si están separadas, el ancho de banda útil cae.

Shared memory: *reduction* por bloque

```cpp
__global__ void reduce_suma(const float* in, float* out,
      int n) {
    __shared__ float tile[256];            // memoria del
          bloque
    int tid = threadIdx.x;
    int i = blockIdx.x * blockDim.x + tid;
    tile[tid] = (i < n) ? in[i] : 0.0f;
    __syncthreads();

    for (int s = blockDim.x / 2; s > 0; s >>= 1) {
        if (tid < s) tile[tid] += tile[tid + s];
        __syncthreads();                   // todos
              terminan el paso
    }
    if (tid == 0) atomicAdd(out, tile[0]); // suma parcial
          del bloque
}
```



Cada paso suma la mitad superior sobre la inferior: $\log_2(256) = 8$ pasos.

`__syncthreads()` sincroniza **solo dentro del bloque**. Los resultados de bloques distintos se combinan con `atomicAdd` o con un segundo kernel.

Reto individual 4: *reduction*

Reto 4 – Suma de 2^{24} elementos

1. Implementa la *reduction* con `shared memory` y `atomicAdd` (usa el kernel de la slide; inicializa antes `*out` en 0 con `cudaMemset`) y verifica contra la CPU.
2. Mide el tiempo de la GPU y compara con `std::accumulate` y con `reduction` de OpenMP del módulo 4.
3. Cambia el tamaño de bloque (64, 128, 256, 512). ¿Cuál es mejor? Recuerda ajustar el tamaño de `tile`.
4. *Extra:* evita el `atomicAdd`: escribe una suma parcial por bloque y reduce el arreglo de parciales con un segundo kernel o en la CPU.

5

Multiplicación de matrices: naive y tiled

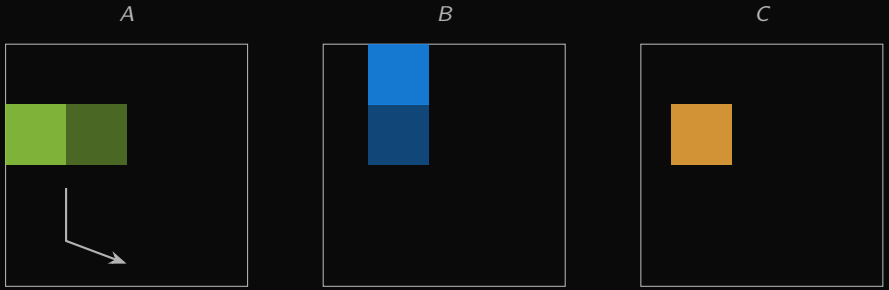
Kernel 2D: matmul naive

```cpp
__global__ void matmul_naive(const float* A, const float* B
    ,
                             float* C, int n) {
    int fila = blockIdx.y * blockDim.y + threadIdx.y;
    int col  = blockIdx.x * blockDim.x + threadIdx.x;
    if (fila < n && col < n) {
        float s = 0.0f;
        for (int k = 0; k < n; ++k)
            s += A[fila * n + k] * B[k * n + col];
        C[fila * n + col] = s;
    }
}

dim3 bloque(16, 16);
dim3 grid((n + 15) / 16, (n + 15) / 16);
matmul_naive<<<grid, bloque>>>(dA, dB, dC, n);
```

- **Un hilo por elemento** de C.
- Cada hilo lee una fila de A y una columna de B: $2n$ lecturas de memoria **global** por elemento.
- Hilos vecinos (`col` consecutivo) leen `B[k*n+col]` contiguo: coalesced. Pero A y B se **releen** n veces desde memoria global.

Tiling: reutilizar datos con shared memory



1) el bloque carga un tile de A y uno de B a **shared memory**

2) `__syncthreads`; los hilos usan el tile T veces

3) acumulan en un registro y avanzan al siguiente tile

Cada elemento de A y B se carga desde memoria global n/T veces (en lugar de n) y se reutiliza T veces desde shared memory (T = tamaño del tile).

matmul con tiles

```cuda
#define TILE 16
__global__ void matmul_tiled(const float* A, const float* B, float* C, int n) {
    __shared__ float As[TILE][TILE], Bs[TILE][TILE];
    int fila = blockIdx.y * TILE + threadIdx.y;
    int col  = blockIdx.x * TILE + threadIdx.x;
    float s = 0.0f;
    for (int t = 0; t < (n + TILE - 1) / TILE; ++t) {
        int a_col = t * TILE + threadIdx.x, b_fila = t * TILE + threadIdx.y;
        As[threadIdx.y][threadIdx.x] = (fila < n && a_col < n) ? A[fila*n + a_col] : 0.0f;
        Bs[threadIdx.y][threadIdx.x] = (b_fila < n && col < n) ? B[b_fila*n + col] : 0.0f;
        __syncthreads();                          // tile completo
        for (int k = 0; k < TILE; ++k) s += As[threadIdx.y][k] * Bs[k][threadIdx.x];
        __syncthreads();                          // nadie sobrescribe antes
    }
    if (fila < n && col < n) C[fila*n + col] = s;
}
```

Lanzamiento: `dim3 bloque(TILE,TILE); dim3 grid((n+TILE-1)/TILE, (n+TILE-1)/TILE);`. Los dos `__syncthreads()` son **imprescindibles**.

Reto individual 5: matmul en GPU

■ Reto 5 – De naive a tiled

1. Implementa `matmul_naive` y verifícalo contra la CPU ($n = 512$, tolerancia 10^{-3}).
2. Implementa `matmul_tiled` con `TILE` 8, 16 y 32 (cuidado: $32 \times 32 = 1024$ hilos).
3. Mide GFLOP/s ($2n^3/t$) para $n = 512, 1024, 2048$ y compáralos con la versión de CPU y con la de OpenMP del módulo 4.
4. Explica con un dibujo por qué `tiled` lee menos de memoria global.
5. *Extra:* compara con `cublasSgemm` (cuBLAS): ¿qué tan cerca estás?

6

Rendimiento: occupancy, streams y herramientas

Occupancy: cuántos warps viven en un SM

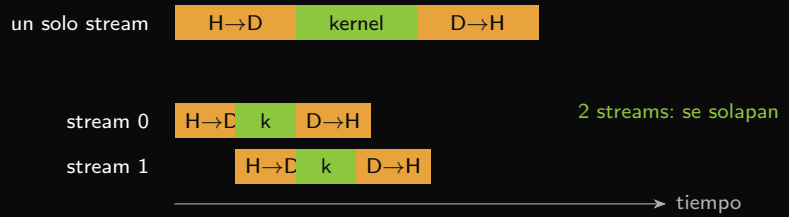
- **Occupancy** = warps activos / máximo de warps por SM. Más warps residentes permiten **esconder la latencia**: mientras uno espera memoria, otro calcula.
- La limitan los **recursos por bloque**: registros por hilo, shared memory por bloque, hilos por bloque.
- Más occupancy **no siempre** es mejor, pero muy poca suele ser un problema.

```
int minGrid, mejorBloque;
cudaOccupancyMaxPotentialBlockSize(&minGrid, &mejorBloque, suma, 0, 0);
std::printf("bloque sugerido: %d\n", mejorBloque);   // sugerencia de la API
```

★ Reglas prácticas

Bloques múltiplos de 32 (128–256 como punto de partida), muchos bloques por grid (varios por SM) y **mide**: el óptimo depende del kernel y de la GPU.

Streams: solapar copia y cómputo




```
cudaMallocHost(&h, bytes);                          // memoria pinned
cudaStream_t s[2];  cudaStreamCreate(&s[0]);  cudaStreamCreate(&s[1]);
for (int k = 0; k < 2; ++k) {                       // cada stream procesa la mitad de los datos
    cudaMemcpyAsync(d + k*mitad, h + k*mitad, mitad*sizeof(float), cudaMemcpyHostToDevice, s[k]);
    escala<<<(mitad + 255) / 256, 256, 0, s[k]>>>(d + k*mitad, mitad, 2.0f);
    cudaMemcpyAsync(h + k*mitad, d + k*mitad, mitad*sizeof(float), cudaMemcpyDeviceToHost, s[k]);
}
cudaDeviceSynchronize();
```

Herramientas de diagnóstico

- ➔ `nvidia-smi`: estado de la GPU (uso, memoria, procesos, temperatura).
- ➔ `compute-sanitizer` (incluido desde CUDA 11): detecta accesos fuera de rango y *race conditions* en *shared memory* (sucesor de `cuda-memcheck`).
- ➔ `nsys` (*Nsight Systems*): línea de tiempo de CPU, GPU, copias y streams.
- ➔ `ncu` (*Nsight Compute*): métricas detalladas de un kernel (occupancy, ancho de banda).
- ➔ `cudaEvent` y `cudaEventElapsedTime`: mediciones dentro del código.

```
module load cuda/11.8
compute-sanitizer ./vecadd                    # verifica memoria
nsys profile -o perfil ./vecadd               # genera perfil.nsys-rep
```

Comprueba con `module avail` y `which nsys ncu` qué herramientas hay en tu entorno antes de planear un análisis con ellas.

No reinventes: bibliotecas

- **cuBLAS**: álgebra lineal densa (`cublasSgemm`).
- **cuFFT**, **cuSPARSE**, **cuRAND**.
- **Thrust**: algoritmos tipo STL en GPU.
- **cuDNN**: redes neuronales.

```
#include <thrust/device_vector.h>
#include <thrust/reduce.h>

thrust::device_vector<float> d(1 <<
    24, 1.0f);
float s = thrust::reduce(d.begin(), d.
    end());
```

★ Criterio

Usa una biblioteca cuando exista: está optimizada por NVIDIA para cada arquitectura. Escribe tu propio kernel para entender el modelo, o cuando tu operación no esté cubierta.

Errores frecuentes en CUDA

- **Olvidar la guarda** `if (i < n)`: escrituras fuera de rango silenciosas.
- **Leer en la CPU sin sincronizar**: el kernel es asíncrono; falta `cudaDeviceSynchronize()`.
- **Usar punteros del host en el device** (o al revés): `d_a` no se desreferencia en la CPU.
- `__syncthreads()` **dentro de un** `if` **divergente**: deadlock o comportamiento indefinido; todos los hilos del bloque deben llegar.
- **Bloques de más de 1024 hilos** o shared memory excesiva: el lanzamiento falla (revisa `cudaGetLastError()`).
- **Mover datos en cada iteración**: copias dentro de un bucle de cómputo.
- **Medir sin sincronizar**: cronometrar solo el lanzamiento, no la ejecución.

Reto individual 6: ¿qué tan eficiente es tu kernel?

■ Reto 6 – Medir y optimizar

1. Escribe los kernels `copia_coalesced` y `copia_strided` (salto 2, 4, 8, 32) y mide el **ancho de banda efectivo** (GB/s) de cada uno.
2. Mide los kernels `sin_divergencia` y `con_divergencia` con 2^{24} elementos. ¿Cuál es más rápido y por qué?
3. Varía el tamaño de bloque (32, 64, 128, 256, 512, 1024) en tu suma de vectores y grafica el tiempo.
4. Corre `compute-sanitizer` sobre un kernel sin la guarda `if (i<n)` y registra el reporte.

7
CUDA en el cluster y MPI + CUDA

Script SLURM para CUDA

```
#!/bin/bash
#SBATCH --job-name=cuda_vecadd
#SBATCH --partition=gpu_titan
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --gres=gpu:1                    # 1 GPU
#SBATCH --time=00:10:00
#SBATCH --output=cuda_%j.out

module load cuda/11.8
nvidia-smi                              # que GPU nos toco
./vecadd
```

- `--gres=gpu:N` reserva *N* GPU; SLURM fija `CUDA_VISIBLE_DEVICES` para tu trabajo.
- Compila con `nvcc -O2 -arch=sm_XX` usando la compute capability de la GPU (`nvidia-smi --query-gpu=compute_cap --format=csv`; en drivers antiguos usa `cudaGetDeviceProperties`).

Compatibilidad de compiladores

❄ CUDA 11.8 y el compilador del host

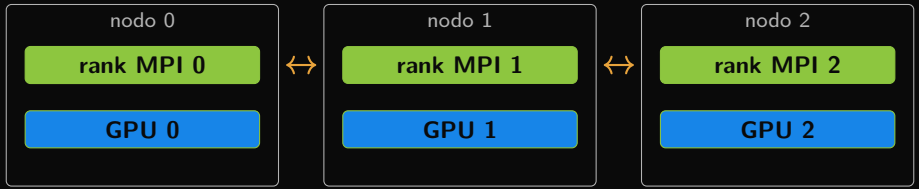
`nvcc` usa un compilador C++ del host (`g++`) y cada versión de CUDA soporta solo **ciertas versiones de GCC**. CUDA 11.8 **no soporta** GCC 14 ni 15 (módulos `gnu14`/`gnu15`): usa un GCC de la serie 11 o anterior para el host, o indícalo con `-ccbin`.

```
module avail gnu                                  # versiones de GCC disponibles
module load cuda/11.8                             # y un GCC compatible con CUDA 11.8
nvcc -O2 -std=c++17 -arch=sm_XX -ccbin g++ -o programa programa.cu
```

➔ Con CUDA 11.8 el estándar máximo para el código del device es **C++17**.

➔ Si ves `unsupported GNU version`, es este problema: cambia el módulo del compilador, no el código.

MPI + CUDA: un rank por GPU




```
int rank, ngpu;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
cudaGetDeviceCount(&ngpu);
cudaSetDevice(rank % ngpu);        // una GPU por rank (rank local del nodo en
    multi-nodo)
// cada rank copia su subdominio, calcula e intercambia halos por MPI
```

Con `--gres=gpu:N` y `--ntasks-per-node=N` cada rank del nodo ve las GPU asignadas. Si MPI soporta *CUDA-aware* puedes pasar punteros de device a `MPI_Send`; si no, copia primero al host.

Reto grupal: difusión de calor 2D en GPU

★ Reto grupal – Equipos de 3 o 4 personas

Lleven la difusión de calor 2D (Jacobi) de los módulos 4 y 5 a la GPU:

1. **Kernel 2D** (`dim3` de 16×16) con dos buffers y *ping-pong* de punteros, sin copiar entre iteraciones.
2. Criterio de convergencia: error máximo por ***reduction*** en GPU (`atomicMax` sobre enteros o *reduction* por bloque) o cada k iteraciones.
3. Mover datos **solo al inicio y al final**; verificar contra la versión de CPU.
4. Barrido de tamaños $N = 512, 1024, 2048, 4096$ y bloques 8×8, 16×16, 32×8; GFLOP/s y ancho de banda efectivo.
5. Comparar CPU secuencial, OpenMP, MPI y GPU en una sola gráfica.
6. *Extra:* versión con *shared memory* (tile + halo) y/o **MPI + CUDA** con dos GPU.

Esquema del reto grupal

```
__global__ void jacobi(const float* T, float* Tn, int N)
      {
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    if (i > 0 && i < N - 1 && j > 0 && j < N - 1)
        Tn[i*N + j] = 0.25f * (T[(i-1)*N + j] + T[(i+1)*N
              + j] +
                               T[i*N + j - 1] + T[i*N + j
                                     + 1]);
}

dim3 bloque(16, 16), grid((N + 15) / 16, (N + 15) / 16);
for (int it = 0; it < iter; ++it) {
    jacobi<<<grid, bloque>>>(T, Tn, N);
    std::swap(T, Tn);                  // intercambio de
          punteros
}
cudaDeviceSynchronize();
```

- ➔ Misma idea de *stencil* del módulo 4: cada celda es el **promedio de 4 vecinas**.
- ➔ Un hilo por celda: `i` (fila) y `j` (columna).
- ➔ Se lee `T` y se escribe `Tn`: sin *race conditions*.
- ➔ Los bordes (`i=0`, `j=0`, …) **no se actualizan**: guardan las condiciones de frontera en ambos buffers.

Entregables y criterios

Entregables

- ➔ Código `.cu` con `Makefile` y `CUDA_CHECK`.
- ➔ Script `sbatch` (`gpu_titan`).
- ➔ Tabla y gráficas CPU vs OpenMP vs MPI vs GPU.
- ➔ Informe corto: qué limita el rendimiento.

Criterios

- ➔ Resultado igual al de la CPU (con tolerancia).
- ➔ Sin errores en `compute-sanitizer`.
- ➔ Mediciones con `cudaEvent`, repetidas.
- ➔ Análisis con ancho de banda y jerarquía de memoria.

Referencias

- NVIDIA, *CUDA C++ Programming Guide* y *CUDA C++ Best Practices Guide*: docs.nvidia.com/cuda.
- M. Harris, *Unified Memory for CUDA Beginners*, NVIDIA Technical Blog.
- J. Sanders, E. Kandrot, *CUDA by Example*, Addison-Wesley, 2010.
- D. Kirk, W. Hwu, *Programming Massively Parallel Processors*, Morgan Kaufmann.
- J. Han, B. Sharma, *Learn CUDA Programming*, Packt, 2019.
- M. Flynn, *Some computer organizations and their effectiveness*, IEEE Trans. Computers, 1972.
- Material del semillero: *Introduction to CUDA Programming* (L. A. Torres Niño, SC3-UIS) y módulos 3, 4 y 5.

Cierre

★ CUDA en una línea

Miles de hilos organizados en *grid* y *blocks*; el rendimiento lo decide la **jerarquía de memoria**: *coalescing*, *shared memory* y pocas transferencias host–device.

¡Gracias!

sc3.operaciones@uis.edu.co