Difusion de Calor 2D - CUDA

Bautista Garcia
bautistagarciace@gmail.com
Universidad Nacional de La Plata

20 de Noviembre, 2025

1 1. Introducción

El objetivo de este trabajo es resolver el problema de simulación de difusión de calor en una superficie 2D utilizando la arquitectura de GPU a través de CUDA. El modelo físico se basa en la Ley de Fourier, donde el calor se propaga desde zonas calientes hacia zonas frías buscando el equilibrio térmico.

La ecuación diferencial que rige este comportamiento es:

$$\frac{\partial T}{\partial t} = D\nabla^2 T \tag{1}$$

Donde D es el coeficiente de difusión y $\nabla^2 T$ es el Laplaciano de la temperatura. Para la simulación computacional, discretizamos el espacio en una grilla cuadrada de $N \times N$ y aproximamos la ecuación mediante diferencias finitas. La temperatura de una celda en el instante $t+1$ se calcula como:

$$T^{t+1}(x,y) = T^t(x,y) + D[T(x+1,y) + T(x-1,y) + T(x,y+1) + T(x,y-1) - 4T(x,y)] \tag{2}$$

Adicionalmente, se debe verificar la convergencia del sistema en cada paso de tiempo. El sistema alcanza el equilibrio cuando la diferencia absoluta entre el promedio de temperaturas de la grilla actual y la anterior es menor a un umbral ε:

$$| \overline{G^{t+1}} - \overline{G^t} | \leq \varepsilon \tag{3}$$

A continuación, se detallan los kernels desarrollados y las optimizaciones aplicadas.

2 2. Kernel de Difusión

1. **Memoria Compartida y Halo:** Cada bloque de hilos carga una porción de la grilla a la memoria compartida propia de cada SM (a la que solo pueden acceder los threads de un mismo bloque). En este caso como cada hilo requiere su posicion correspondiente y las de sus vecinos, debemos hacer cargas adicionales en los bordes, para que la memoria compartida cuente con todos los elementos.

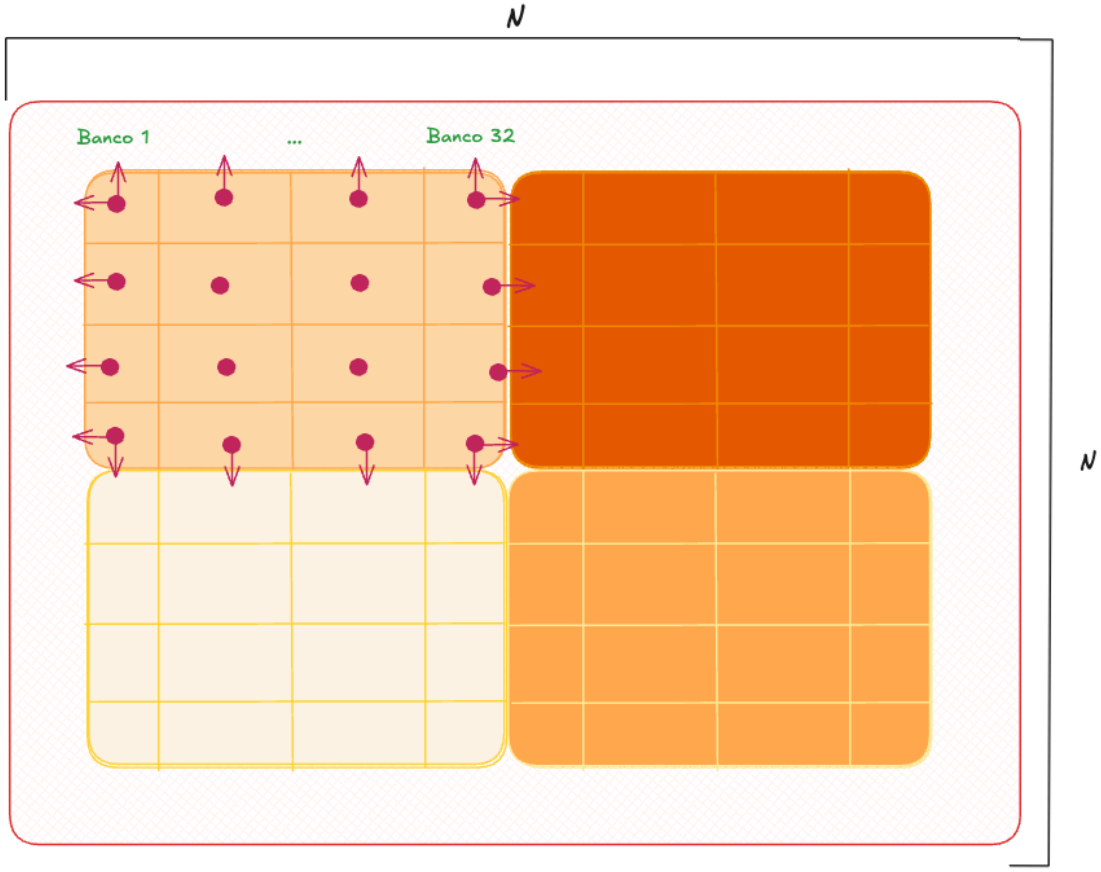

Figure 1: Patron de acceso

Cada color representa un bloque distinto.

En la **Figura 1** se puede observar el patron de acceso y la carga de los bordes, ademas se puede observar que los bordes (en blanco) no se procesan (condiciones de borde de Dirichlet).

En las cargas a memoria compartida no existen conflictos de banco ya que los bloques se lanzan con una configuracion `dim3 block(32, CEILDIV(threadsperblock)//32)`. Por lo que cada warp estara compuesto de 32 threads con la misma coordenada y y distinta coordenada x, ademas `dtype=float` por lo que cada warp solo accede a 32 bancos.

```
// En cada warp solo cambia lx
smem[ly * stride + lx] = _grid[i];
```

Adicionalmente se comprobo la ausencia de conflictos de banco de forma practica usando **Nsight Compute** (herramienta de profiling de NVIDIA) [ver archivo `.ncu-rep`].

	Instructions	Requests	Wavefronts	% Peak	Bank Conflicts
Shared Load	620	620	620	1.44	0
Shared Load Matrix	0	0			
Shared Store	316	316	316	0.74	0
Shared Atomic	0	0	0	0	0
Other	-	-	936	2.18	0
Total	936	936	1872	4.35	0

Figure 2: Conflictos de Banco

2. **Acceso Coalescente:** Se maximizo la carga coalescente desde memoria global en los casos donde fue posible. En cada carga de **center, top y bottom** esto fue posible ya que la grilla esta ordenada por filas. El unico caso de acceso no coalescente fue en las cargas de los bordes **right y left**. Ver Figura 2:

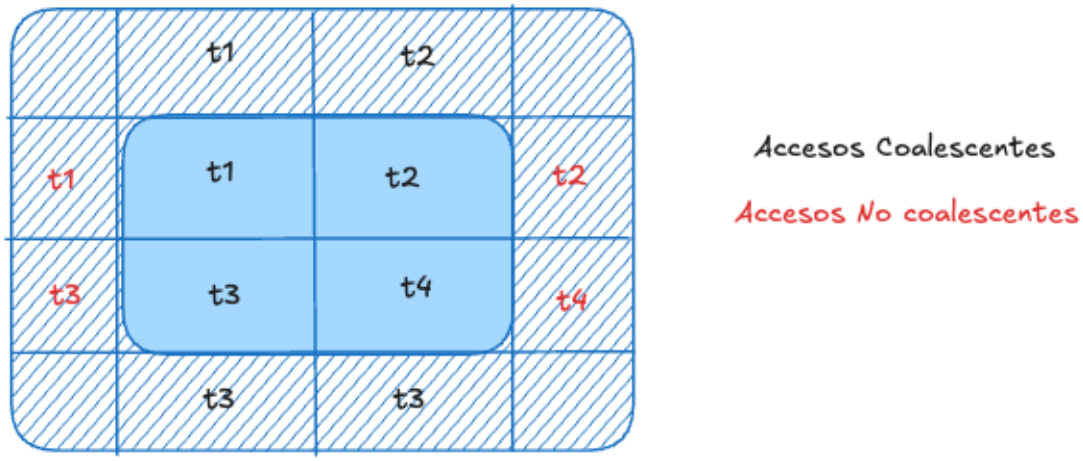

Figure 3: Accesos

En el codigo siguiente se puede observar que para las cargas **top y bottom** unicamente varia la coordenada x (acceso coalescente). Mientras que para los bordes izquierdo y derecho varia la coordenada y (acceso no coalescente).

```
if ((threadIdx.y == 0) && (y > 0)) { // Top Halo
    smem[0 * stride + lx] = _grid[(y - 1) * grid_size + x];
}
if ((threadIdx.y == blockDim.y - 1) && (y < grid_size - 1)) { // Bottom Halo
    smem[(blockDim.y + 1) * stride + lx] = _grid[(y + 1) * grid_size + x];
}

if ((threadIdx.x == 0) && (x > 0)) { // Left Halo
    smem[ly * stride + 0] = _grid[y * grid_size + x - 1];
}
if ((threadIdx.x == blockDim.x - 1) && (x < grid_size - 1)) { // Right Halo
    smem[ly * stride + (blockDim.x + 1)] = _grid[y * grid_size + x + 1];
}
```

3 3. Kernel de Reducción (Convergencia)

Para verificar la convergencia del algoritmo, es necesario calcular el promedio de temperatura de toda la grilla en cada paso. Esto implica sumar los $N \times N$ elementos de la matriz, mediante un kernel de **reduccion**.

Debido a que unicamente podemos sincronizar y comunicar los threads a nivel de bloques, debemos lanzar el kernel de reduccion mas de una vez para reducir iterativamente los resultados parciales calculados por cada bloque.

La estrategia utilizada se basa en la presentacion [**Optimizing Parallel Reduction in CUDA - Mark Harris**]:

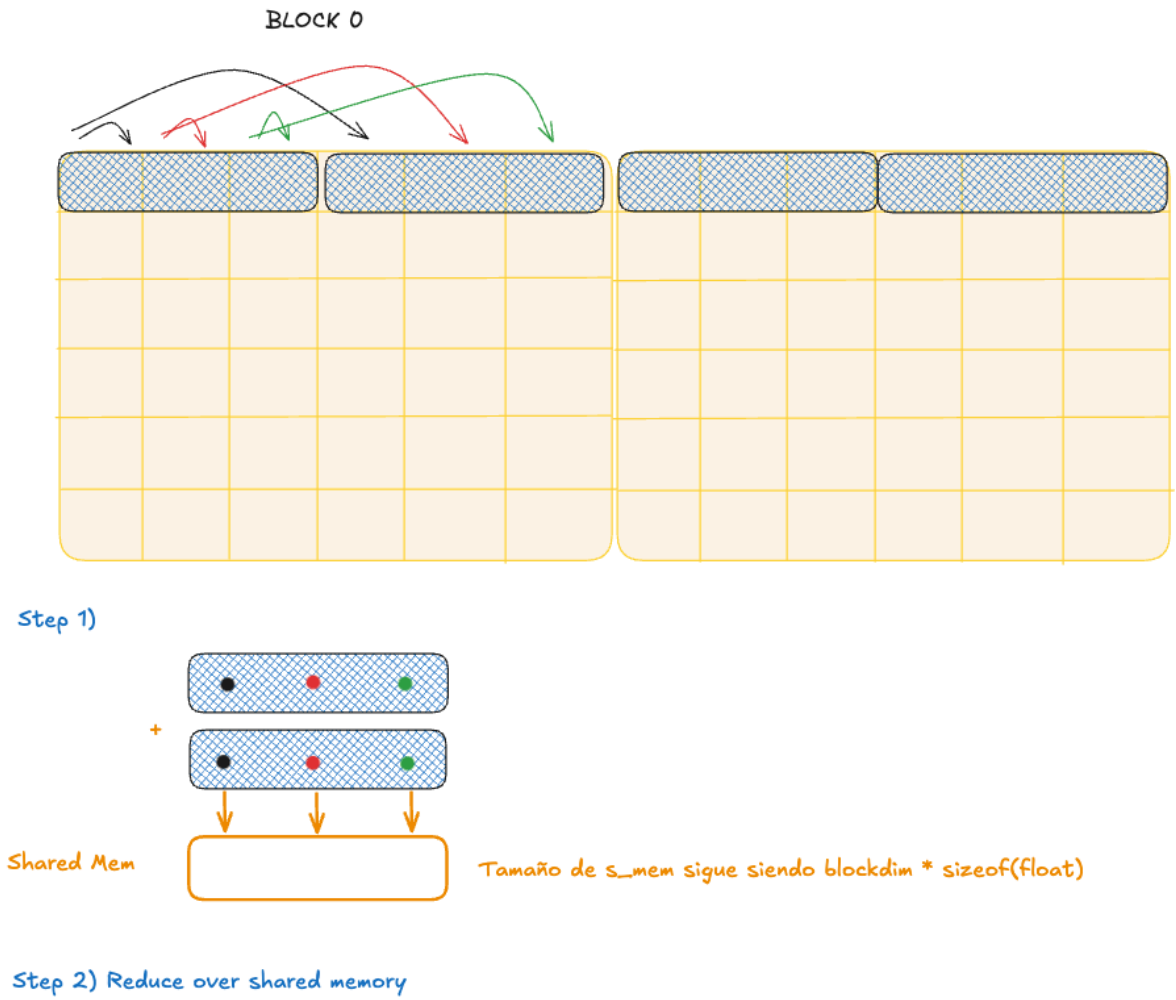

Figure 4: Estrategia de Reduccion

La estrategia consiste en cargar a memoria compartida la primer suma parcial correspondiente al tamaño de dos bloques:

```
unsigned int i = blockIdx.x * (blockDim.x*2) + threadIdx.x;
sdata[tid] = input[i] + input[i+blockDim.x];
```

De esta forma reducimos en un factor de 2 la cantidad de bloques necesarios para la reduccion, en comparacion a cargar a memoria compartida unicamente la posicion correspondiente al propio thread (`blockIdx.x * blockDim.x+ threadIdx.x`).

Una vez cargada la primer suma parcial a la memoria compartida, aplicamos la reduccion sobre esta. En esta reduccion indexamos a la memoria compartida con el siguiente patron para evitar los conflictos de banco:

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
        if (tid < s) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
}
```

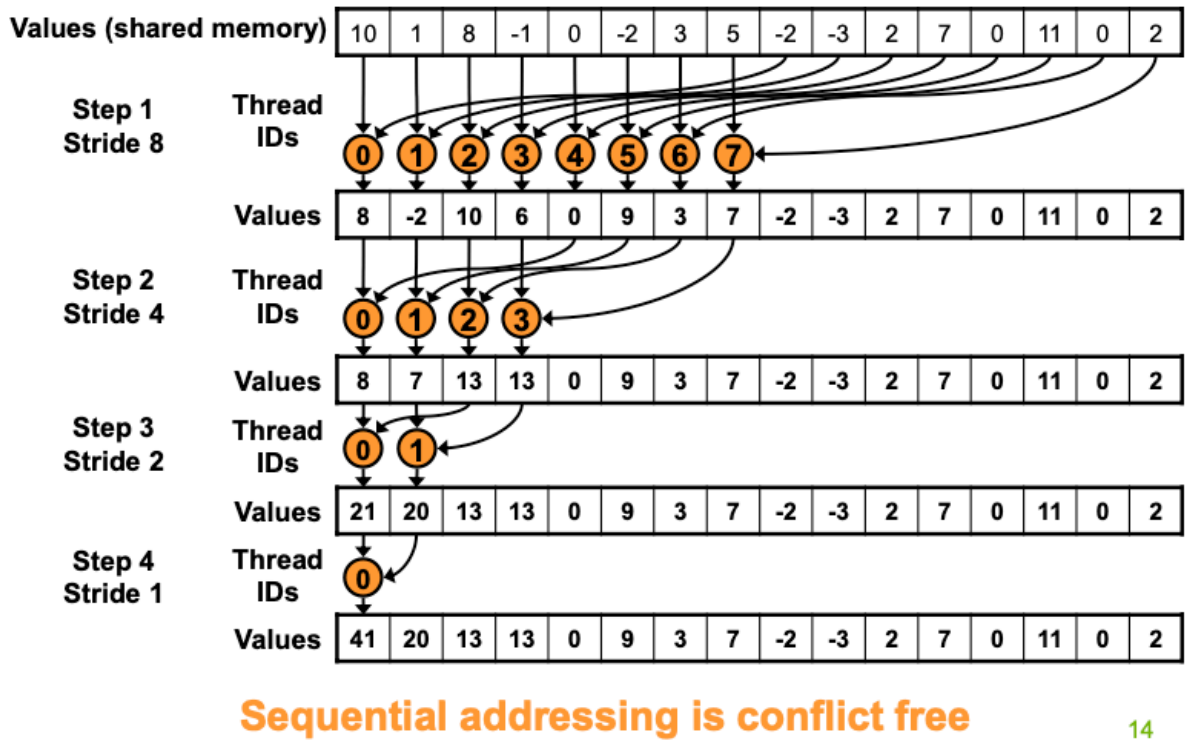

Figure 5: Indexado en memoria compartida

NOTA: *Se podrian haber utilizado optimizaciones adicionales como* ***loop unrolling*** *o el* ***uso de templates****, pero estas no fueron vistas en profundidad y se pierde cierta legibilidad en el codigo.*

Utilizando la herramienta de **profiling** se verifico la ausencia de conflictos de banco:

	Instructions	Requests	Wavefronts	% Peak	Bank Conflicts
Shared Load	272	272	272	0.57	0
Shared Load Matrix	0	0			
Shared Store	192	192	192	0.40	0
Shared Atomic	0	0	0	0	0
Other	-	-	1152	2.40	0
Total	464	464	1616	3.36	0

Figure 6: Conflictos de banco

4 4. Parametros de Kernels

La correcta configuración de las dimensiones de la grilla (`gridDim`) y de los bloques (`blockDim`), así como el cálculo de la memoria compartida dinámica, son fundamentales para la corrección y eficiencia del algoritmo. A continuación se detalla la estrategia utilizada para cada etapa.

4.1 Configuración de Difusión (Stencil 2D)

Dado que la simulación física ocurre en un espacio bidimensional, mapeamos la topología de los hilos para que coincida con la geometría del problema.

```
dim3 block_diffusion(TILE_X, CEIL_DIV(threads_per_block, TILE_X));
dim3 grid_diffusion(CEIL_DIV(N, block_diffusion.x), CEIL_DIV(N, block_diffusion.y));
size_t shm_bytes_diffusion = (size_t)(block_diffusion.y + HALO_SM) *
(block_diffusion.x + HALO_SM) * sizeof(float);
```

- **Dimensiones del Bloque:** Fijamos `TILE_X`= 32 para que coincida con el tamaño de un **Warp**. Esto maximiza el acceso coalescente a memoria, ya que hilos contiguos en x accederán a direcciones contiguas en memoria global. La dimensión y se calcula dividiendo el total de hilos por bloque ingresados por el usuario sobre `TILE_X`.
- **Dimensiones de la Grilla:** Se utiliza una división entera con techo (`CEIL_DIV`) para asegurar que se cubra la totalidad de la grilla de tamaño $N \times N$, lanzando suficientes bloques para procesar incluso los bordes si N no es múltiplo del tamaño del bloque.
- **Memoria Compartida:** El tamaño solicitado no es solo el del bloque ($\text{Block}_x * \text{Block}_y$), sino que incluye el **Halo** (`HALO_SM`). Esto reserva espacio extra para cargar los vecinos necesarios para el cálculo de la difusion sin tener que volver a acceder a memoria global.

4.2 Reducción Iterativa (Cálculo de Convergencia)

Configuración Inicial:

```
dim3 block_reduce(threads_per_block, 1);
dim3 grid_reduce(CEIL_DIV(N * N, threads_per_block * 2), 1);
```

Se utiliza una grilla 1D linealizada. Un punto clave es el divisor `threads_per_block * 2`. Esto se debe a que el kernel de reducción realiza una **primera suma durante la carga**: cada hilo lee dos valores de la memoria global y los suma antes de empezar la reducción en memoria compartida. Esto permite procesar $2 \times M$ elementos utilizando solo M hilos, incrementando la eficiencia de la instrucción de carga.

Reconfiguracion iterativa:

```
while (grid_reduce.x > 1) {
    reduction_kernel<<<grid_reduce, ...>>>(...);
    cudaDeviceSynchronize();

    // Actualización de parámetros para el siguiente paso
    len_reduce = grid_reduce.x;
    grid_reduce.x = CEIL_DIV(len_reduce, threads_per_block * 2);
}
```

1. **Reducción Parcial:** En la primera iteración, `grid_reduce.x` bloques generan `grid_reduce.x` resultados parciales.
2. **Redimensionamiento:** Para la siguiente iteración, la entrada ya no es N^2, sino el número de bloques de la iteración anterior (`len_reduce`). Recalculamos `grid_reduce.x` para cubrir esta nueva cantidad de elementos.

5 Simulacion grafica con OpenGL

Para verificar de forma grafica, la unica diferencia con las funciones de `heat_simulacion.c` es que `update_simulation()` retorna un booleano que indica cuando la simulacion converge.

La version actual del `.cu` imprime en un archivo de texto los resultados finales, los cuales fueron visualizados con un script de Python para verificar que los resultados tengan sentido. Esto se debe a que el acceso a mi GPU es mediante ssh (por lo que no se puede utilizar OpenGL).

6 Resultados y Profiling

La informacion del `device` en el que se realizaron las simulaciones:

Figure 7: GPU utilizada

6.1 Tiempos

Dado que los lanzamientos de kernels en CUDA son asíncronos, se incluyó una barrera de sincronización explícita (`cudaDeviceSynchronize()`) inmediatamente después de cada llamada al kernel, asegurando así que el tiempo registrado contemple la ejecución completa en la GPU y no solo la latencia de lanzamiento. Los valores presentados corresponden al tiempo promedio por paso, calculado acumulando la duración de cada operación a lo largo de toda la simulación y dividiéndolo por el número total de pasos ejecutados. El error utilizado para el calculo de convergencia fue:

$$\varepsilon = 0.00001 \tag{4}$$

N	Pasos Ejecutados	Tiempo Total (s)	Difusión Avg (μs)	Reducción Avg (μs)	D2D Avg (μs)	D2H Avg (μs)
512	10000	2.570	21.97	38.27	2.77	171.09
1024	10000	6.760	73.38	107.77	2.92	468.97
2048	10000	23.470	271.80	384.46	3.04	1666.97
4096	508	4.763	1072.20	1497.54	3.10	6490.15