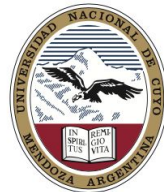


Dinámica de un gas de electrones en un recinto circular

Estudiante: Pablo N. Chegade



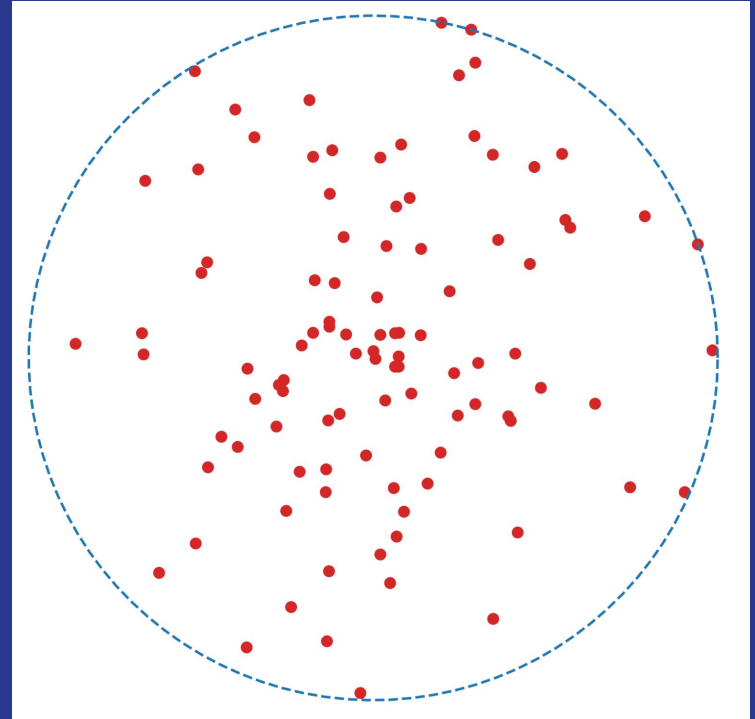
UNCUYO
UNIVERSIDAD
NACIONAL DE CUYO



Comisión Nacional
de Energía Atómica

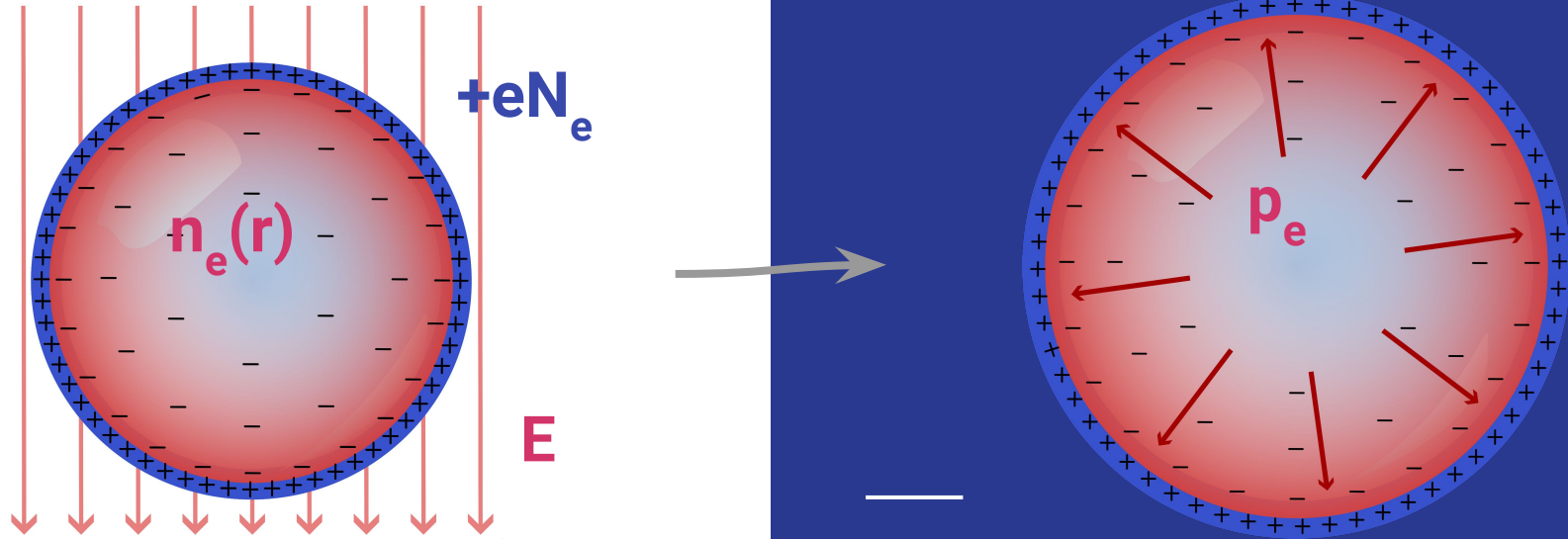
Introducción

El objetivo del proyecto es simular la dinámica de un gas de N electrones contenido en un recinto circular 2D de radio R_0 . En particular, nos interesa conocer las propiedades del gas en el equilibrio.



Motivación

El estudio de estas propiedades permite analizar el proceso de evolución de una avalancha de electrones generada en una cavidad rodeada por una fuente de electrones.



Definición del problema

Ecuaciones de movimiento adimensionalizadas

$$\frac{d\vec{r}_i}{dt} = \vec{v}_i$$

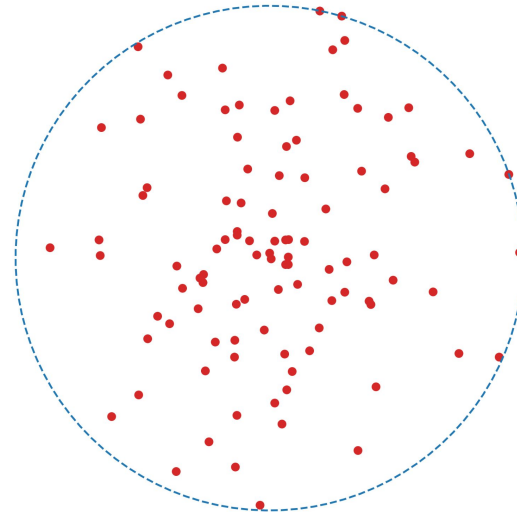
$$\frac{d\vec{v}_i}{dt} = \sum_{j=0}^{N-1} \alpha \frac{\vec{r}_i - \vec{r}_j}{|\vec{r}_i - \vec{r}_j|^3}$$

$$\alpha = \frac{e^2}{mR_0v_0^2} \quad v_0 = \sqrt{\frac{2KT}{m}}$$

Condiciones iniciales

$$r_{0,i,x}, r_{0,i,y} \sim U(0, 1) \quad \forall i$$

$$v_{0,i,x}, v_{0,i,y} \sim U(0, 1) \quad \forall i$$



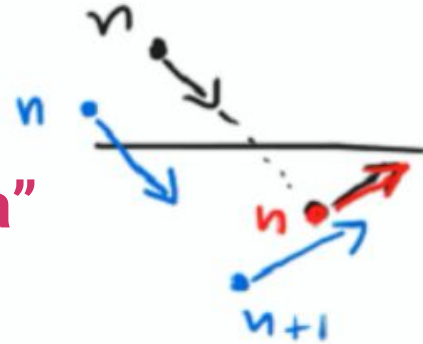
Corrección de Temperatura

$$\vec{v}_{new} = \lambda \vec{v}$$

$$\lambda = \sqrt{\frac{N}{\sum_i |\vec{v}_i|^2}}$$

Colisiones con la pared

pared “blanda”



Implementación

¿Qué cálculos hay que realizar?

1. **Loop sobre partículas:** asignar aleatoriamente posición y velocidades de las N partículas
2. **Loop temporal:**
 - a. Dadas \mathbf{r}_i^n y \mathbf{v}_i^n , calcular $\mathbf{r}_i^{(n+1)}$ y $\mathbf{v}_i^{(n+1)}$ mediante el método de Verlet. Esto implica:

$$\mathbf{r}_i^{n+1} = \mathbf{r}_i^n + \Delta t \mathbf{v}_i^n + \frac{1}{2} \mathbf{F}_i^n \Delta t^2, \quad \mathbf{v}_i^{n+1} = \mathbf{v}_i^n + \frac{1}{2} (\mathbf{F}_i^{n+1} + \mathbf{F}_i^n)$$

- i. **Loop sobre partículas:** calcular \mathbf{F}_i^n
 - ii. **Loop sobre partículas:** calcular las nuevas posiciones $\mathbf{r}_i^{(n+1)}$
 - iii. **Loop sobre partículas:** calcular $\mathbf{F}_i^{(n+1)}$
 - iv. **Loop sobre partículas:** calcular las nuevas velocidades $\mathbf{v}_i^{(n+1)}$
- b. **Loop sobre partículas:** verificar si alguna partícula "chocó" con la pared, es decir, ver si $|\mathbf{r}_o^{(n+1)}| > R_0$. En caso positivo, invertir la velocidad radial
 - c. **Loop sobre partículas:** corregir las velocidades para que la temperatura sea la deseada

En resumen, los procesos son:

1. **Condiciones iniciales**
2. **Loop temporal**
 - a. Método de Verlet
 - i. **Cálculo de fuerzas**
 - ii. **Integración de posiciones y velocidades**
 - b. **Rebotes**
 - c. **Corrección de velocidades**

Versiones en serie y en paralelo


Distintas versiones del código

Versión	Lenguaje	Modo de ejecución
Versión 1	Python	En serie (numpy)
Versión 2	Python	En paralelo (numpy -> cupy)
Versión 3	C++	En serie
Versión 4	CUDA C++	En paralelo (kernels)
Versión 5	CUDA C++	En paralelo (kernels + shared memory)

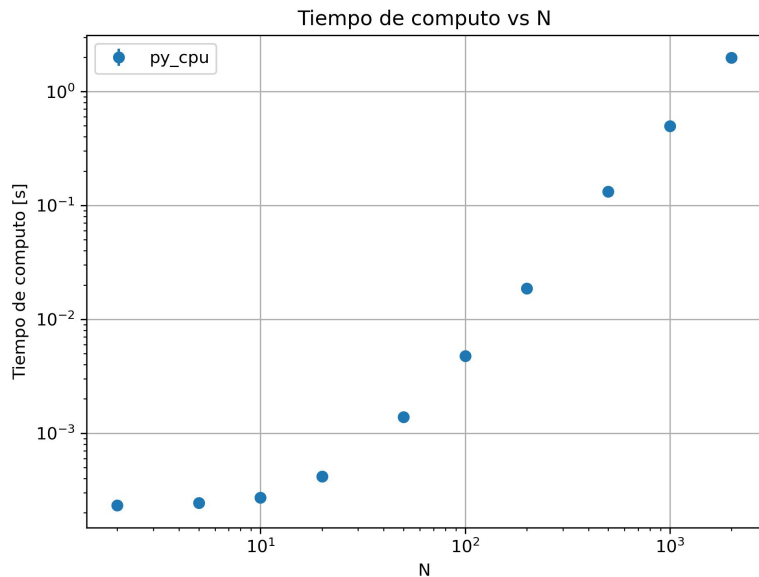
Versión 1: Python en serie

- Necesaria para hacer prototyping
- Cálculos eficientes empleando numpy

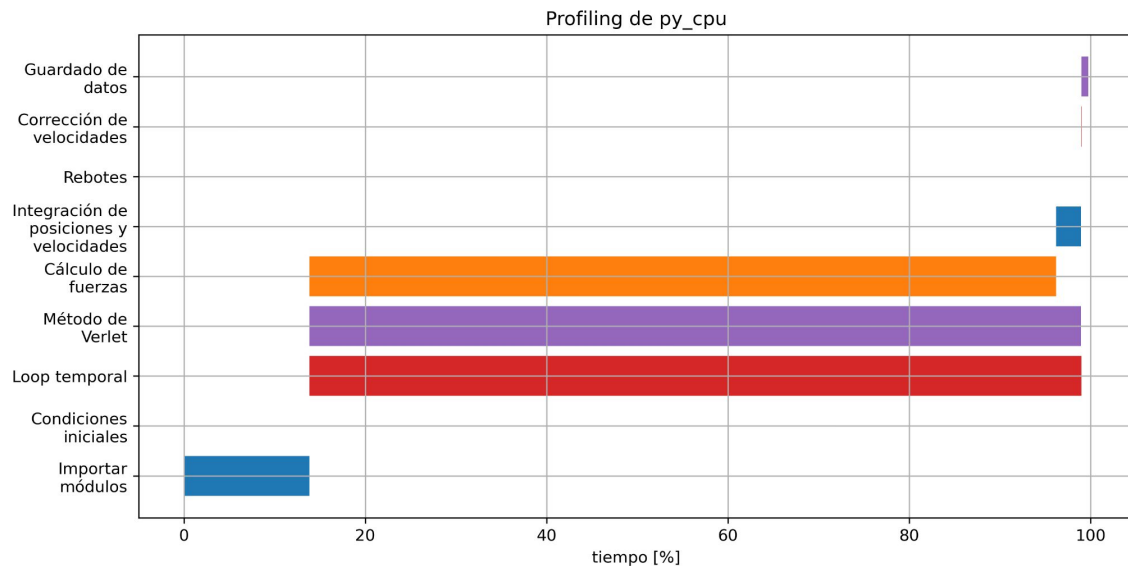
Cálculo	Método
Condiciones iniciales	numpy
Loop temporal	Loop de Python
Cálculo de fuerzas	numpy
Integración de posiciones y velocidades	numpy
Rebotes	Loop de Python
Corrección de velocidades	numpy


$$F = \begin{pmatrix} 0 & F_{1,2} & F_{1,3} & \cdots & F_{1,N} \\ F_{2,1} & 0 & F_{2,3} & \cdots & F_{2,N} \\ F_{3,1} & F_{3,2} & 0 & \cdots & F_{3,N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ F_{N,1} & F_{N,2} & F_{N,3} & \cdots & 0 \end{pmatrix}$$

Versión 1: Python en serie



- A mayor tamaño, mayor tiempo de cómputo



- Gran parte del tiempo se dedica al loop temporal.

Profiling realizado con N = 100 y CPU AMD Ryzen 7 4700U with Radeon Graphics

Versión 2: Python en paralelo

- Estudiamos con rigurosidad cómo implementar la versión paralela en python
- Desarrollamos un código de gran complejidad pero elegante optimización que logra disminuir el tiempo de cómputo en órdenes de magnitud

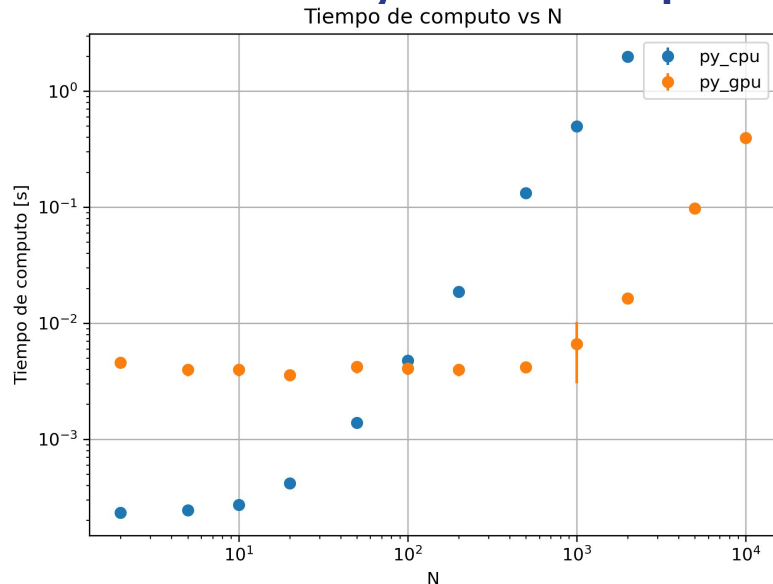
```
import numpy as np
```



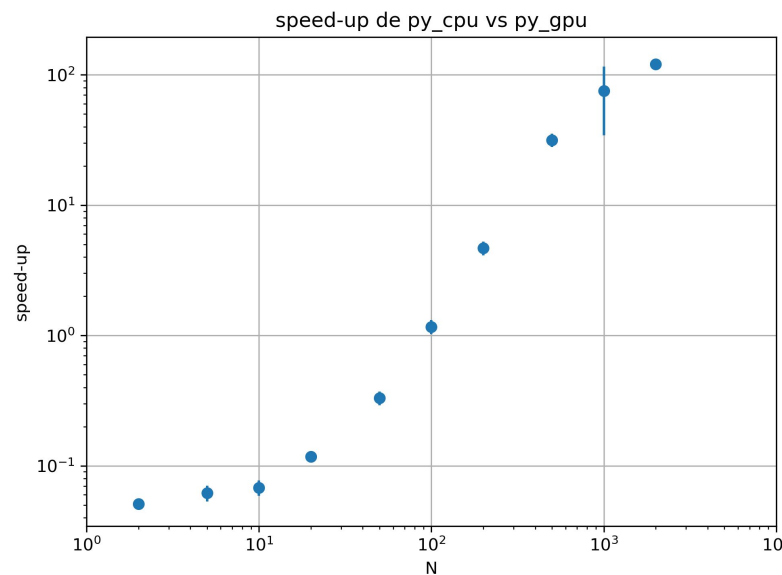
```
import cupy as np
```

- ¿Es posible optimizar el código con el menor esfuerzo posible?

Versión 2: Python en paralelo

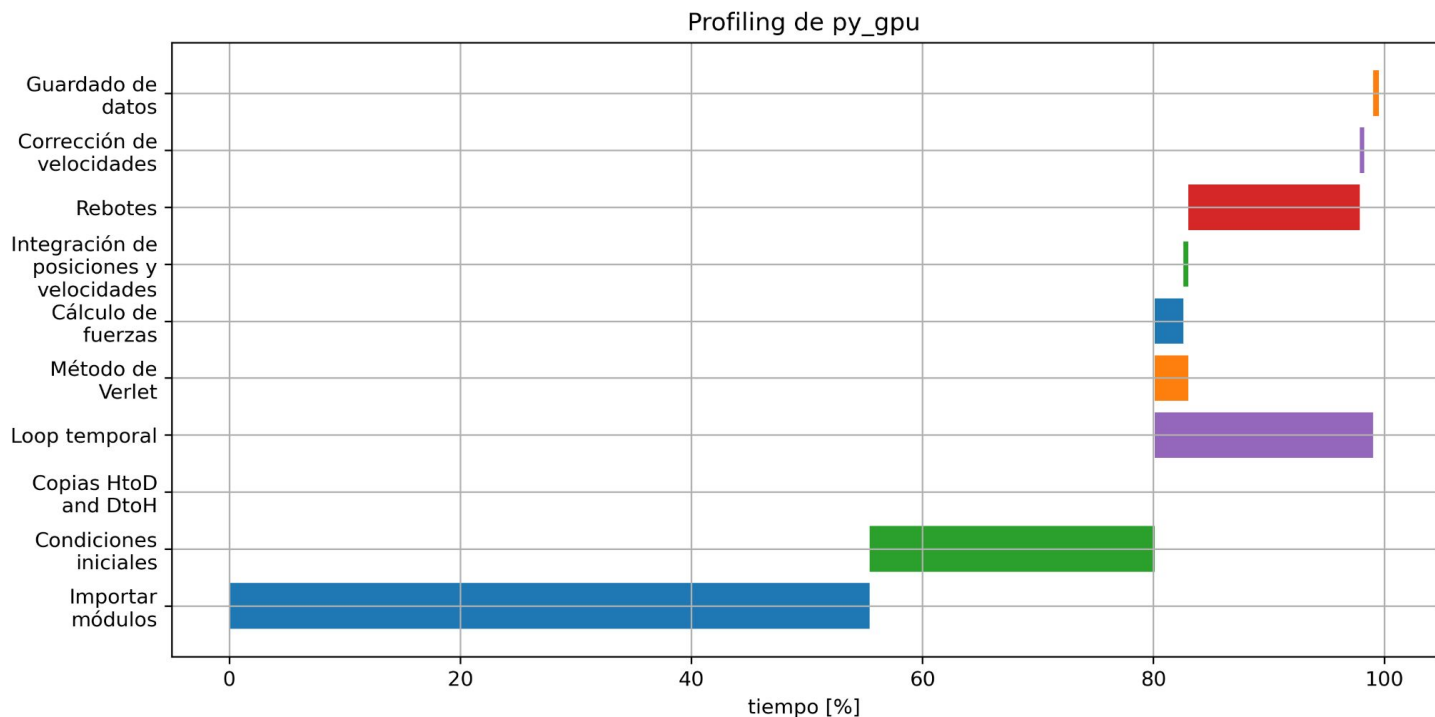


- Constante para N pequeño, pero mayor que la versión en serie
- Crece con N a N grande debido a que la GPU serializa



- speed-up del orden de x100

Versión 2: Python en paralelo



Gran parte del tiempo está destinado a importar módulos y a la compilación

Profiling para N = 100
con CPU Intel(R) Xeon(R)
2.00GHz y GPU Nvidia
Tesla T4

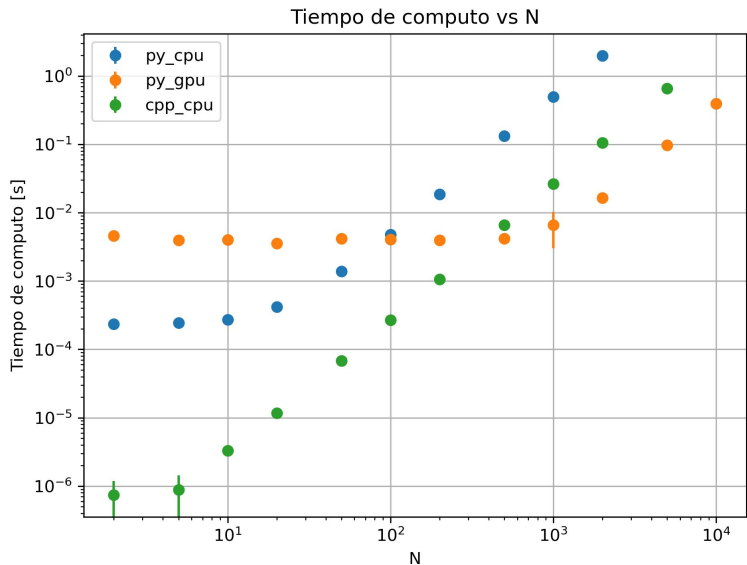
Versión 3: C++ en serie

- Se emplearon loops de C++ para todos los cálculos, inclusive para las fuerzas

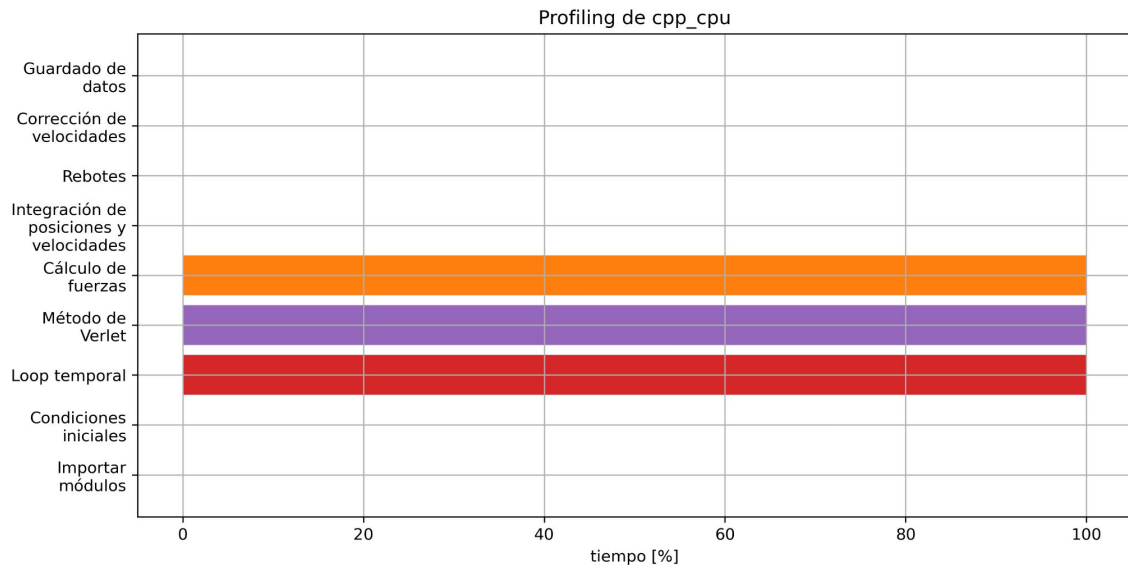
Cálculo	Método
Condiciones iniciales	Loop de C++
Loop temporal	Loop de C++
Cálculo de fuerzas	Loop de C++
Integración de posiciones y velocidades	Loop de C++
Rebotes	Loop de C++
Corrección de velocidades	Loop de C++

```
for(int i = 0; i < N; ++i){  
    // Calculo drdt  
    dydt[i] = vx_vec[i]; //drx_vec  
    dydt[N + i] = vy_vec[i]; //dry_vec  
  
    // Calculo dvdt  
    //dvx_vec = dydt[2 * N + i]  
    //dvy_vec = dydt[3 * N + i]  
    dydt[2 * N + i] = 0;  
    dydt[3 * N + i] = 0;  
    for(int j = 0; j < N; ++j){  
        if (i != j){  
            float dx = rx_vec[i] - rx_vec[j];  
            float dy = ry_vec[i] - ry_vec[j];  
            float r = sqrt(dx*dx + dy*dy);  
            float r3 = r*r*r;  
            dydt[2 * N + i] += alpha*dx/r3;  
            dydt[3 * N + i] += alpha*dy/r3;  
        }  
    }  
}
```

Versión 3: C++ en serie



- El tiempo de cómputo se reduce considerablemente respecto a Python.
- Pero al igual que este aumenta con N



- Casi el 100% del tiempo de cómputo se destina al cálculo de fuerzas

Profiling realizado con N = 100 y CPU AMD Phenom II X4 955 3.21 GHz

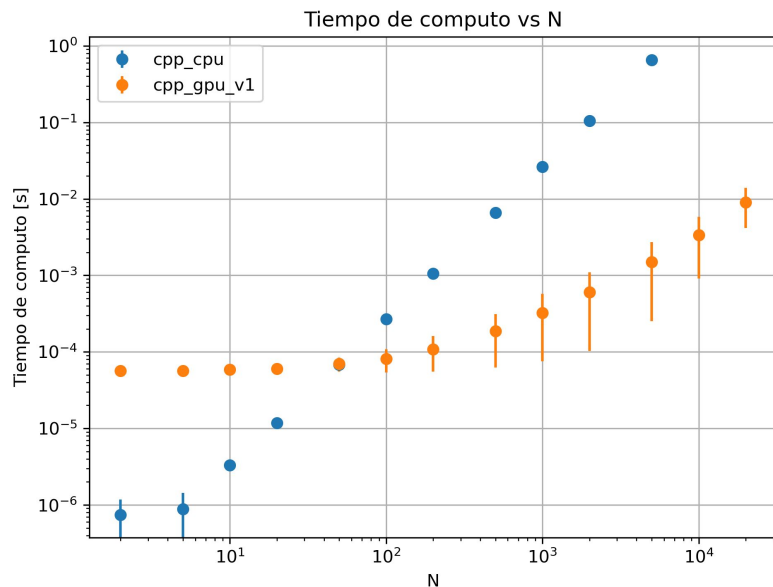
Versión 4: CUDA C++ en paralelo

- Uso kernels para cada una de las cuentas
- BLOCK_SIZE de 256
- Copias HtoD y DtoH solo al inicio y al final de la evolución

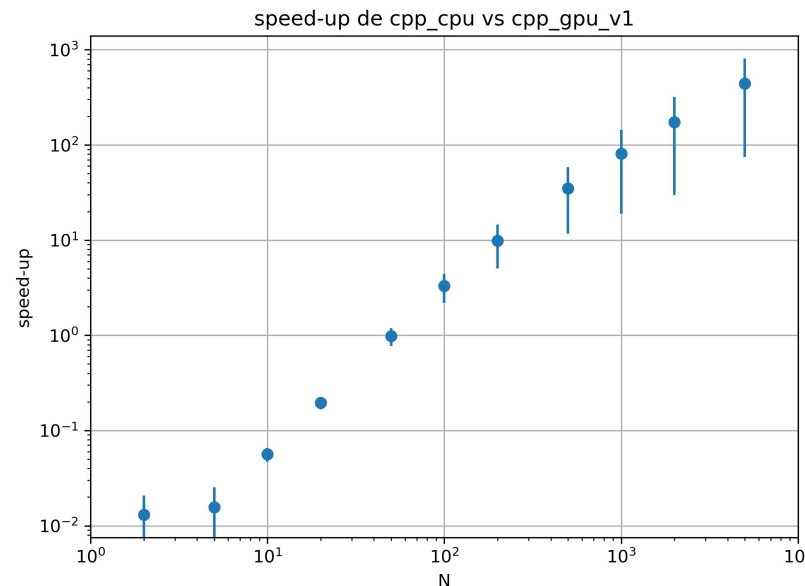
Cálculo	Método
Condiciones iniciales	Loop de C++
Loop temporal	Loop de C++
Cálculo de fuerzas	kernel
Integración de posiciones y velocidades	kernel
Rebotes	kernel
Corrección de velocidades	kernel

```
__global__
void bodyForce(Particula *p, Particula *dpdt, float dt,
int N, float alpha) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N) {
        float Fx = 0.0f; float Fy = 0.0f;
        for (int j = 0; j < N; j++) {
            float dx = p[i].x - p[j].x;
            float dy = p[i].y - p[j].y;
            float r2 = dx*dx + dy*dy + SOFTENING;
            float inv_r = rsqrtf(r2);
            float inv_r3 = inv_r * inv_r * inv_r;
            Fx += alpha * dx * inv_r3; Fy += alpha * dy * inv_r3;
        }
        //Asigno las derivadas
        dpdt[i].x = p[i].vx; dpdt[i].y = p[i].vy;
        dpdt[i].vx = Fx; dpdt[i].vy = Fy;
    }
}
```

Versión 4: CUDA C++ en paralelo

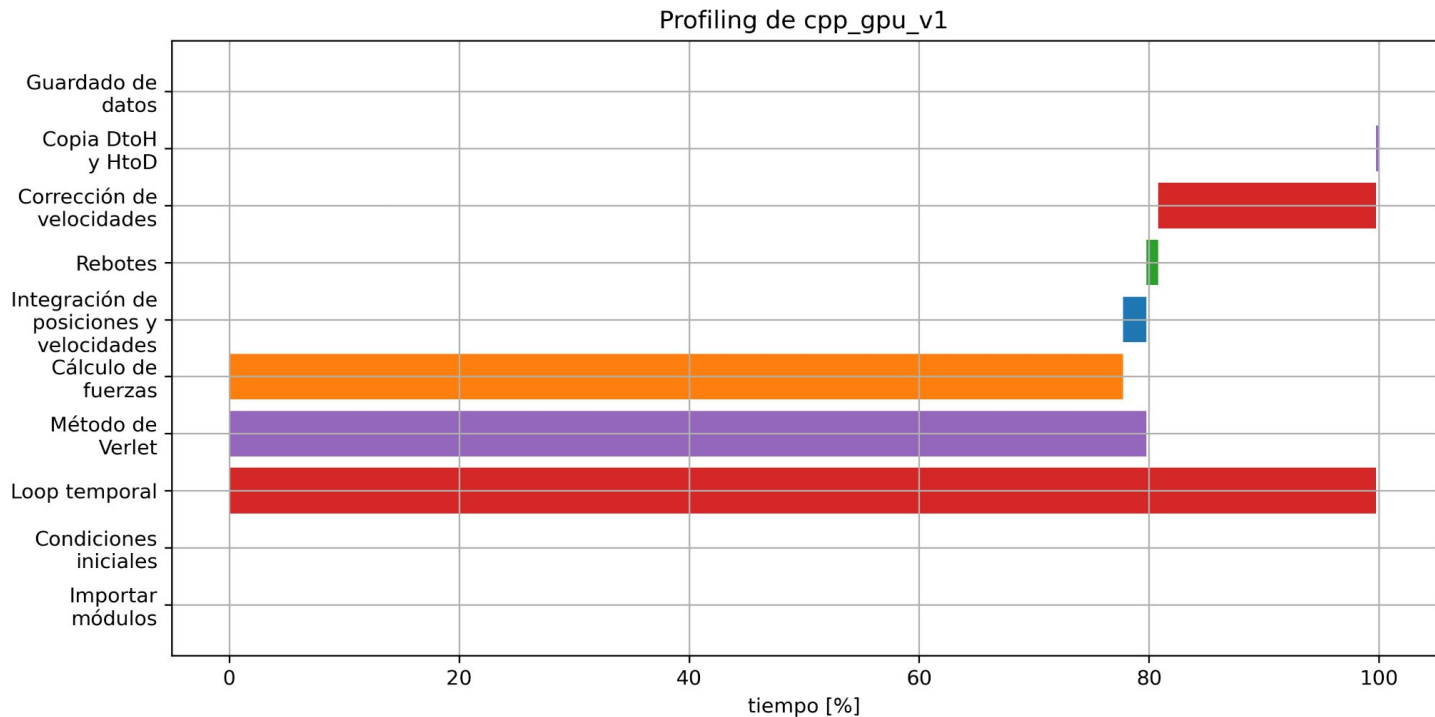


- Comportamiento análogo a lo visto en Python



- Speed-up de hasta x500

Versión 4: CUDA C++ en paralelo



- No se consideran las copias HtoD y DtoH
- Se logró disminuir el tiempo de cómputo del cálculo de fuerzas (de forma relativa)

Profiling realizado con N = 100, CPU AMD Phenom II X4 955 3.21 GHz y GPU NVIDIA GeForce GTX TITAN X

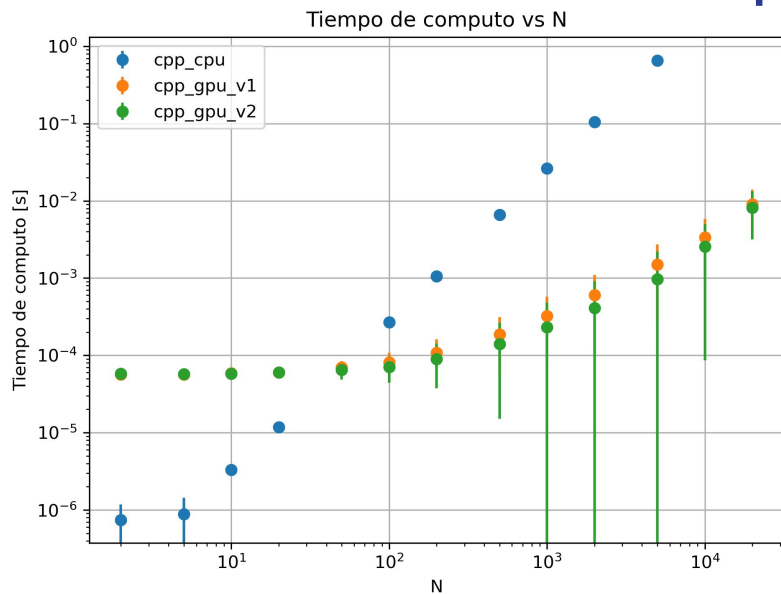
Versión 5: CUDA C++ en paralelo optimizado

- Idéntica a la versión anterior salvo por el cálculo de fuerzas, en el cual se emplea memoria compartida

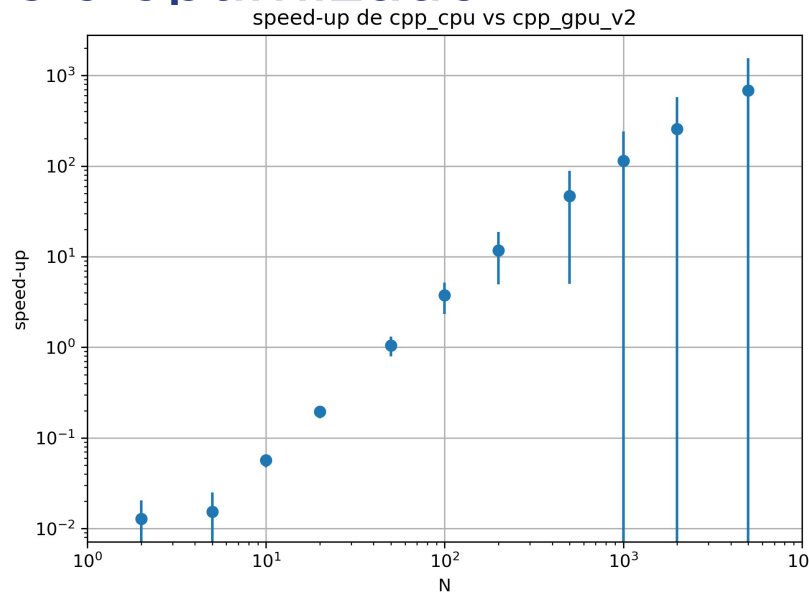
```
__global__
void bodyForce(Particula *p, Particula *dpdt, float dt, int N, float alpha) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N) {
        float Fx = 0.0f; float Fy = 0.0f;
        for (int tile = 0; tile < gridDim.x; tile++) {
            __shared__ Particula shared_p[BLOCK_SIZE];
            shared_p[threadIdx.x] = p[tile * blockDim.x + threadIdx.x];
            __syncthreads();

            int n = min(BLOCK_SIZE, N - tile * BLOCK_SIZE); // número de partículas
            en este bloque
            for (int j = 0; j < n; j++) {
                float dx = p[i].x - shared_p[j].x;
                float dy = p[i].y - shared_p[j].y;
                float r2 = dx*dx + dy*dy + SOFTENING;
                float inv_r = rsqrtf(r2);
                float inv_r3 = inv_r * inv_r * inv_r;
                Fx += alpha * dx * inv_r3; Fy += alpha * dy * inv_r3;
            }
            __syncthreads();
        }
        dpdt[i].x = p[i].vx; dpdt[i].y = p[i].vy;
        dpdt[i].vx = Fx; dpdt[i].vy = Fy;
    }
}
```

Versión 5: CUDA C++ en paralelo optimizado

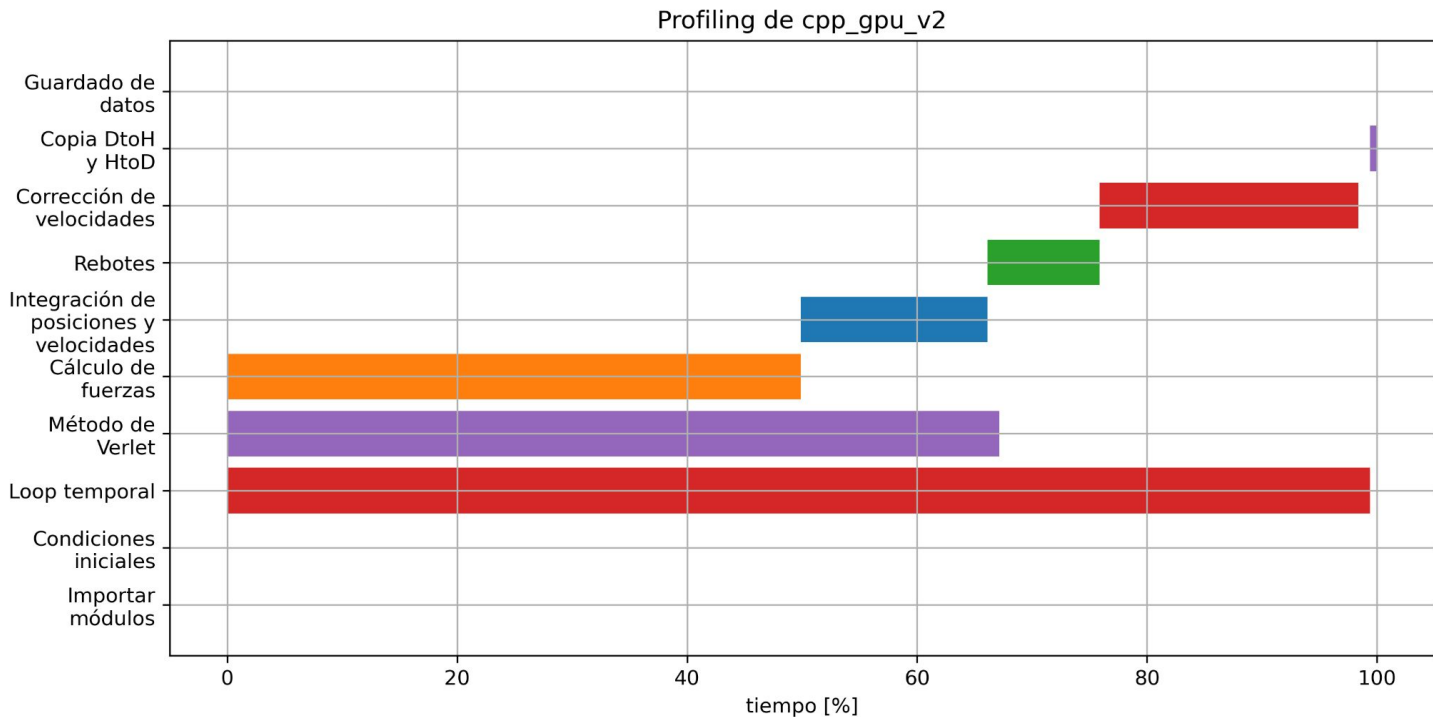


- Tiempo de cómputo consistentemente menor que la última versión



- Se obtiene un speed-up ligeramente mayor

Versión 5: CUDA C++ en paralelo optimizado



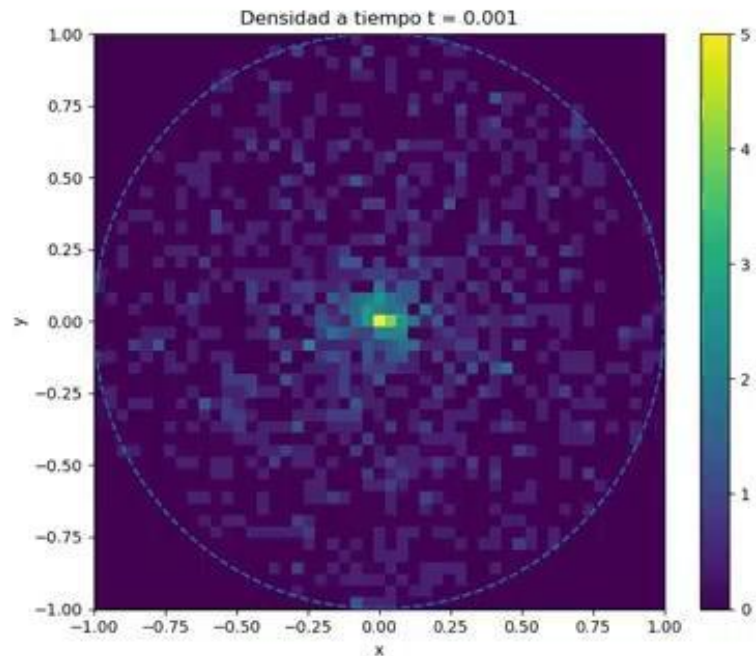
- No se consideran las copias HtoD y DtoH
- Se logró disminuir aún más el tiempo de cómputo del cálculo de fuerzas (de forma relativa)

Profiling realizado con N = 100, CPU AMD Phenom II X4 955 3.21 GHz y GPU NVIDIA GeForce GTX TITAN X

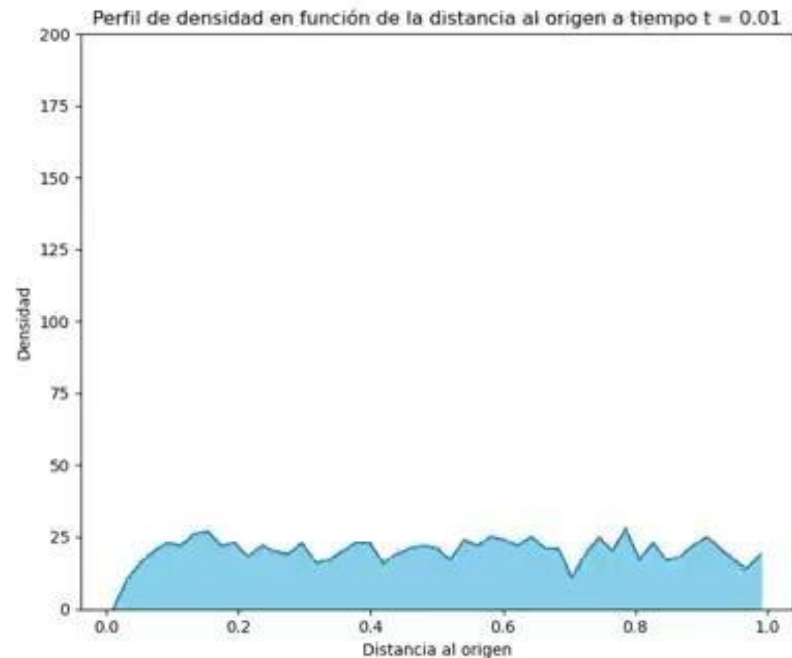
Resultados

Resultados

Densidad espacial en función del tiempo



Densidad radial en función del tiempo



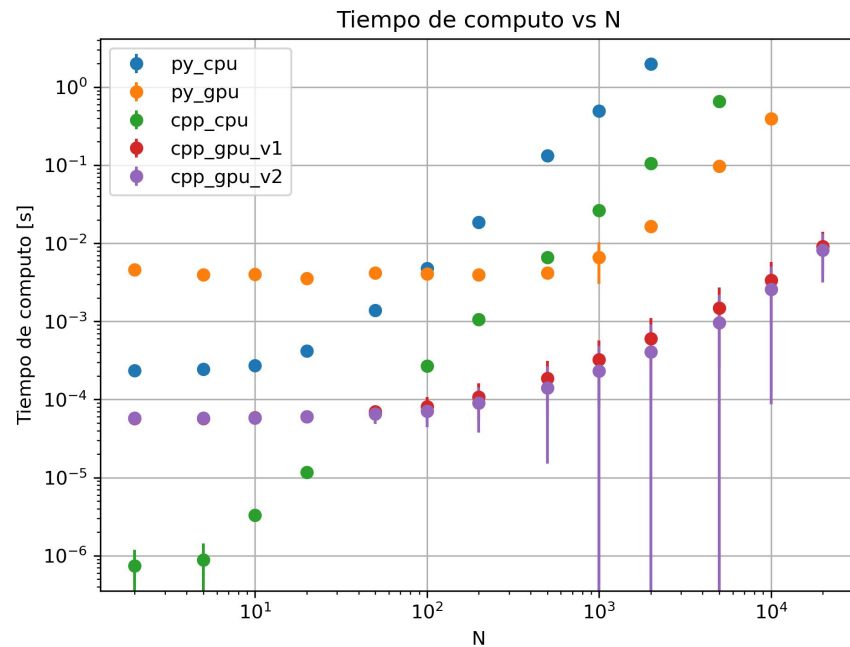
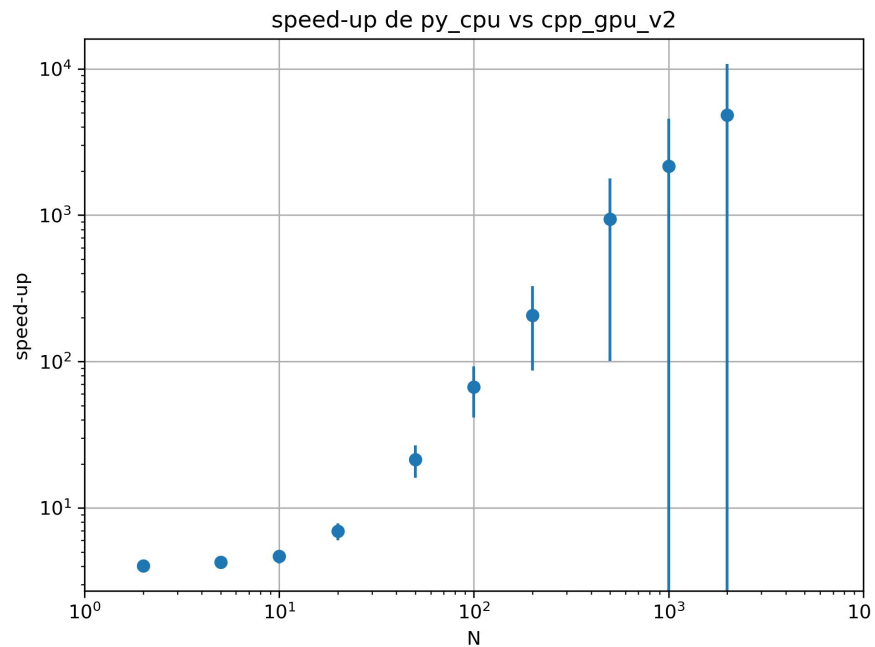
Conclusión

Conclusión

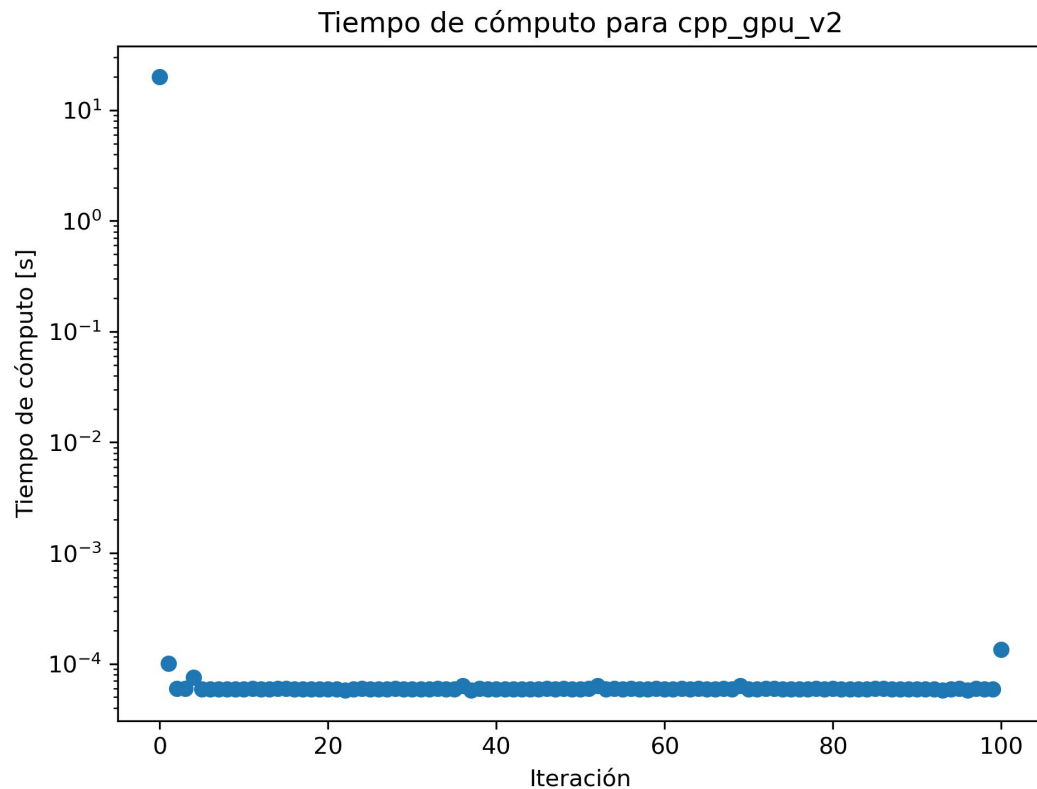
- Logramos determinar la dinámica de un gas de N electrones dentro de un recinto circular
- Desarrollamos un código en serie en Python y C++
- Desarrollamos un código en paralelo en Python y 2 en CUDA C++ con distinta optimización
- Estudiamos el tiempo de cómputo asociado a cada una de ellas logrando consistentemente un mejor rendimiento de la versión paralela respecto a la versión en serie para N grande
- Determinamos cualitativamente las propiedades en el equilibrio

Muchas gracias!
Preguntas?

Anexo 1 de 100



Anexo 2 de 100



Anexo 3 de 100

- kernel de rebotes

```
__global__
void rebote_blando(Particula *p, int N, float R0) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N) {
        //Opero sobre las partículas que chocaron
        if (sqrt(p[i].x * p[i].x + p[i].y * p[i].y) > R0) {

            // Obtengo las variables correspondientes
            float vx = p[i].vx;
            float vy = p[i].vy;

            float tita = atan2(p[i].y, p[i].x);
            p[i].vx = -vx * cos(2 * tita) - vy * sin(2 * tita);
            p[i].vy = -vx * sin(2 * tita) + vy * cos(2 * tita);
        }
    }
}
```