Desarrollo de dispositivos con redes percolativas de nanohilos de plata.

Pablo Chehade
pablo.chehade@ib.edu.ar
Introducción al Cálculo Numérico en Procesadores Gráficos, Instituto Balseiro, CNEA-UNCuyo, Bariloche, Argentina

Script de la presentación de CUDA

I. INTRODUCCIÓN

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

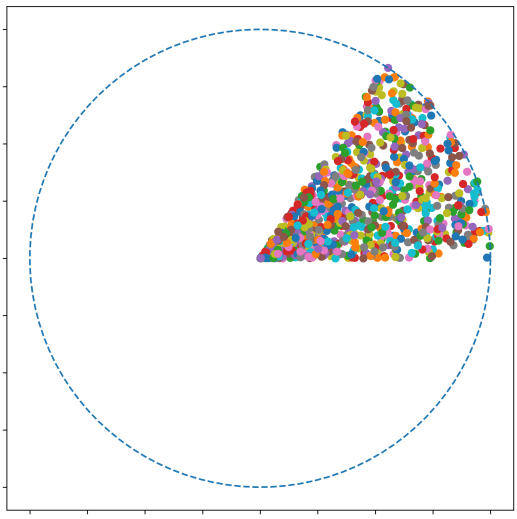

Figura 1: Figura representativa: algunos electrones adentro de un círculo de radio R_0. Hacerlo con python y con el mismo formato que los resultados.

II. 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.

- Explicar el proceso de avalancha de electrones de la tesis

III. DEFINICIÓN DEL PROBLEMA

1 **[Opciones de modelo y qué modelo elegimos]**

Para modelar el gas de electrones hay 2 opciones:
1. estadística
2. partículas

A. Ecuaciones de movimiento

2 **[Ecuaciones]** ecuaciones de movimiento dadas por la ley de Newton y la ley de Lorentz, adimensionalizadas

- Ecuaciones adimensionalizadas

3 **[Método de Verlet]** Ecuaciones del método (método simpléctico? conservativo)

B. Condiciones iniciales

Como condición inicial se parte de N electrones en posiciones y velocidades aleatorias con distribución uniforme, ambas entre 0 y 1 por la adimensionalización elegida.

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

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

C. Corección de Temperatura

El sistema es conservativo, de modo que dadas las condiciones iniciales la energía se conserva en el tiempo. La energía está constituida por energía cinética (temperatura) y energía potencial. Pero a nosotros no nos interesa el equilibrio para dada energía inicial total, sino que nos interesa el equiñlibrio a determinada temperatura. De este modo, se va a corregir

D. Colisiones con la pared

Se asume que las colisiones con la pared son elásticas, considerando la misma como una "pared blanda". Esto significa que durante la evolución la partícula invierte su velocidad si atraviesa la pared, pero no se refleja su posición. Esto es útil para evitar discontinuidades en la energía total del sistema.

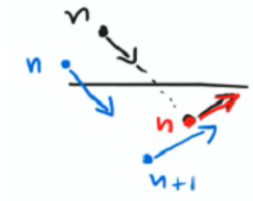

Figura 2: Figura del rebote blando

IV. MÉTODO NUMÉRICO

V. IMPLEMENTACIÓN

4 **[¿Qué pasos debe hacer nuestro código para calcular la evolución del gas de electrones?]**

1. Loop sobre partículas: asignar aleatoriamente posición y velocidades de las N partículas
2. Loop temporal:
   a) Dadas r_0^n y v_0^n calculas r_0^{n+1} y v_0^{n+1} mediante el método de Verlet. Esto implica
      1) Loop sobre partículas: calcular $F_{i,j}^n$
      2) Loop sobre partículas: calcular las nuevas posiciones r_i^{n+1}
      3) Loop sobre partículas: calcular $F_{i,j}^{n+1}$
      4) Loop sobre partículas: calcular las nuevas velocidades v_i^{n+1}
   b) Loop sobre partículas: verificar si alguna partícula çhocóçon la pared, es decir, ver si $|r_0^{n+1}| > R_0$. En caso positivo, invertir la velocidad radial
   c) Loop sobre partículas: corregir las velocidades para que la temperatura sea la deseada

Poner los items anteriores. Poner "loop sobre partículasz "loop temporal.en un color distinto.

Los procesos son

- Condiciones iniciales
- Loop temporal
  - Método de Verlet
    - Cálculo de fuerzas
    - Integración de posiciones y velocidades
  - Rebotes
  - Corrección de velocidades

itemize
Condiciones iniciales Loop temporal Método de Verlet Cálculo de fuerzas Integración de posiciones y velocidades Rebotes Corrección de velocidades
Tenemos muchos loops que podrían ser paralelizables

VI. VERSIONES DEL CÓDIGO EN SERIE Y EN PARALELO

*Contar cada versión por separado. Hacer una tabla traspuesta en la que diga
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)
Me gustaría hacer una "historia"de cómo fui cambiando de una versión a la otra y en el camino menciono cómo fui haciendo las cuentas. Esto debería intercalarse con el profiling y los gráficos de speed-up

A. Versión 1: Python en serie

- Esta versión fue implementada en un Notebook de Python. Se empleó este lenguaje porque inicialmente el problema no estaba bien definido y tuve que hacer mucho prototyping
- Para hacer los cálculos de forma eficiente, decidí usar numpy en lugar de los loops de python.
- Ejemplo de código en el que usé numpy: asignación de las condiciones iniciales
- Pude hacerlo en todos los pasos salvo en el cálculo de los rebotes, ahí usé loops de python
- En particular, la consecuencia de lo anterior fue tener que calcular una matriz de fuerzas
- Gráfico de la matriz de fuerzas
- Análisis del tiempo de cómputo
- Análisis de profiling

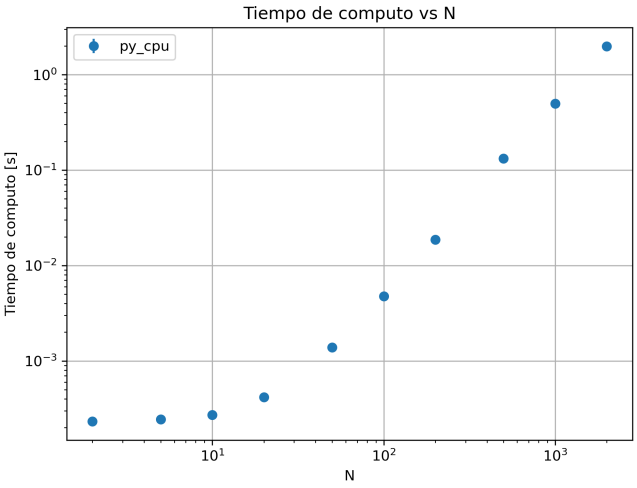

Figura 3: Gráfico de tiempo de cómputo

B. Versión 2: Python en paralelo

- Sólo intercambié numpy por cupy como para demostrar qué speed-up uno podría obtener con un código
- Escribir "import numpy as np" → "import cupy as np"
- Profiling. Ver qué CPU y GPU estoy usando y análisis

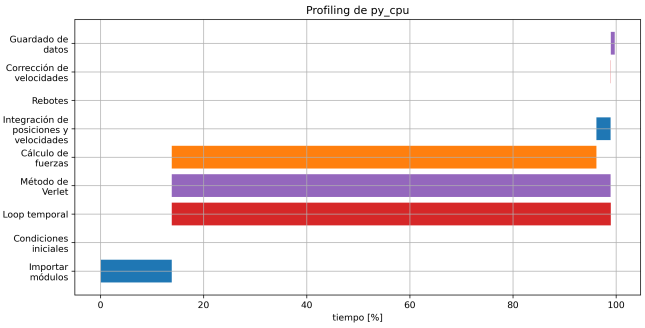

Figura 4

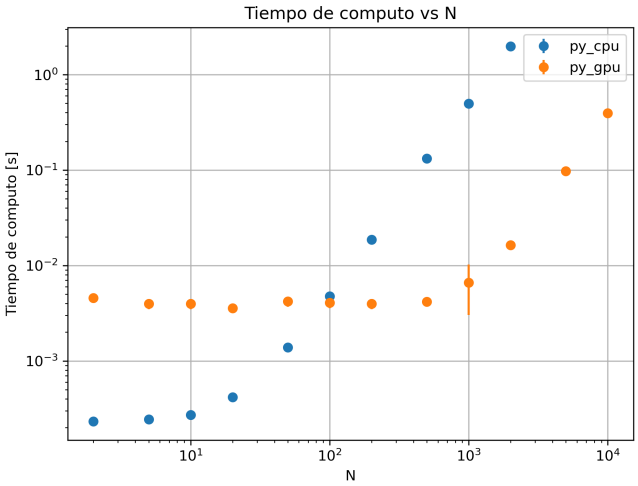

Figura 5: Gráfico de tiempo de cómputo ($\mathrm{py}_c uy \mathrm{py}_g pu$)

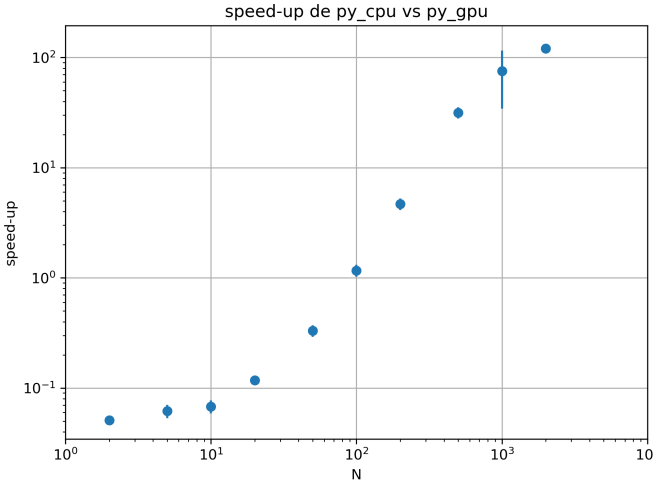

Figura 6: Gráfico de speed-up respecto a $\mathrm{py}_c pu$

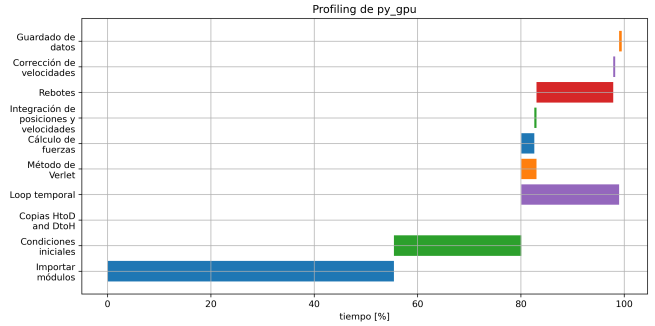

Figura 7

C. Versión 3: C++ en serie

- Usé los loops de C++ para todos los cálculos. Las fuerzas ya no se calculan mediante una matriz.
- Poner la función de fuerzas
- Ver qué CPU estoy usando y análisis

D. Versión 4: C++ en paralelo v1

- Usé kernels para hacer cada una de las cuentas, un blocksize de 256? y solo hice copias al inicio y al final de la evolución
- Volver a poner los items al inicio de ïmplementación¿ poner al lado de cada item una flecha y la declaración de los kernels
-
-
- . Ver qué CPU y GPU estoy usando y análisis

E. Versión 5: C++ en paralelo v2

- usé shared-memory
- Poner la sección en la que usé shared memory
- . Ver qué CPU y GPU estoy usando y análisis

VII. RESULTADOS

-Animación o gráfico de los resultados Hacer una animación de la densidad radial en el tiempo comparándola con la solución de Bunkin para N grande
Todo profiling hacerlo con N = 2000
En una CPU con Información de la CPU: Nombre: AMD Ryzen 7 4700U with Radeon Graphics Arquitectura: $X86_64 Frecuencia$: $[1400000000, 0] Núcleos físicos$: 8
$\mathrm{py}_c pu$:
Para hacer profiling: jupyter nbconvert $\mathrm{py}_c pu_s imulacion_d e_p articulas.ipynb$ – $-topythonpython$ – $mcProfile$ – $opy_c pu_s imulacion_d e_p articulas.profilepy_c pu_s imulacion_d e_p articul$
Del total 85.15 para hacer el loop temporal 13.83 lo usa el compilador para importar módulos 0.77 para guardar los archivos txt
Del 85.15: 85.10 se usa para el método de Verlet, el restante para los rebotes

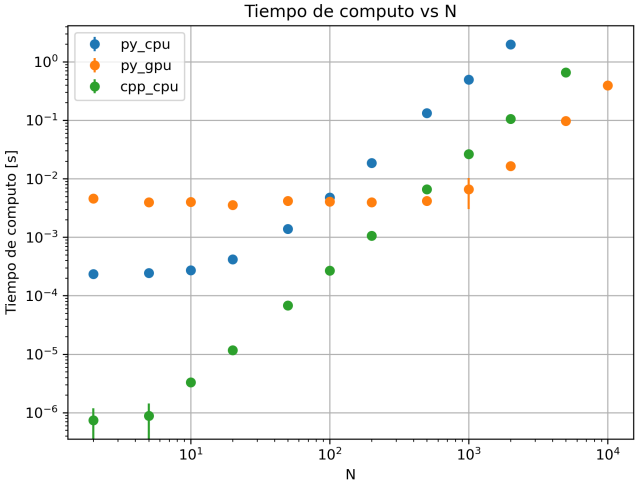

Figura 8: Gráfico de tiempo de cómputo ($\text{py}_c u, py_g puy cpp_c pu$)

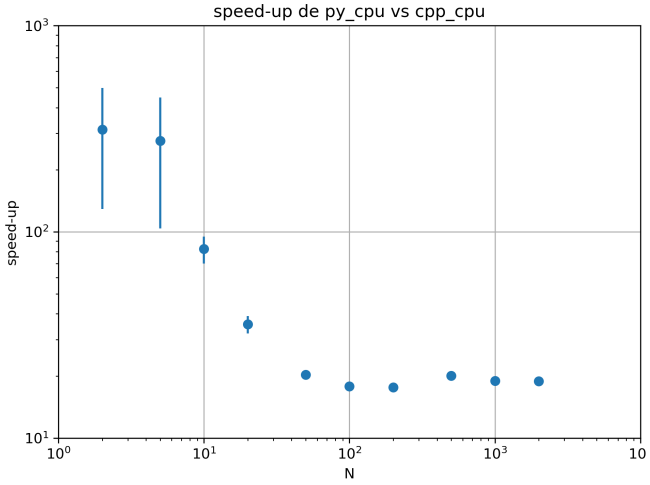

Figura 9: Gráfico de speed-up respecto a $\text{py}_c pu$

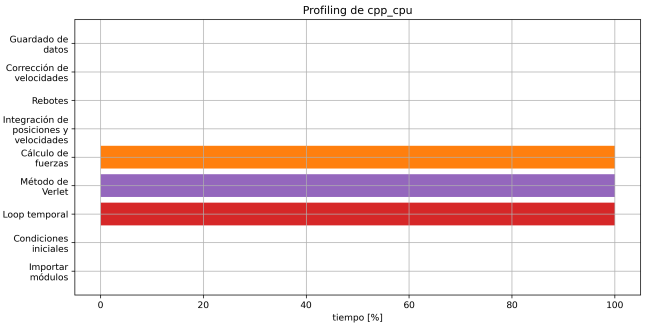

Figura 10

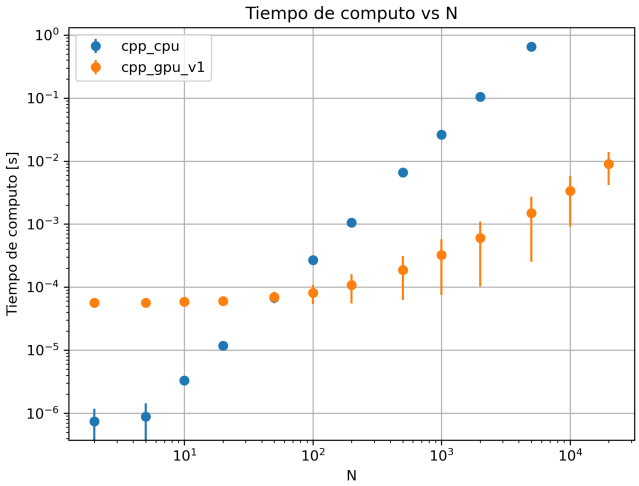

Figura 11: Gráfico de tiempo de cómputo ($\text{cpp}_c pu, cpp_g pu_v 1$)

Del 85.10: 82.38 se usa para calcular las fuerzas, el restante para integrar posiciones y velocidades
————- CPU: AMD Ryzen 7 4700U with Radeon Graphics Tiempo total: 100
P1 Importar módulos: 13.83 P2 Condiciones iniciales: 0. P3 Loop temporal: 85.15 S1 Método de Verlet: 85.10 SS1 Cálculo de fuerzas: 82.38 SS2 Integración de posiciones y velocidades: 85.10-82.38 S2 Rebotes: 0.025 S3 Corrección de velocidades: 0.025 P4 Guardado de datos: 0.77 ————
$\text{py}_g pu$: $Parahacerprofiling$: $jupyternbconvertpy_g pu_s imulacion_d e_p articulas.ipynb--topythonpython - mcProfile - opy_g pu_s imulacion_d e_p articulas.profilepy_g pu_s imulacion_d e_p articulas.py$
Información de la CPU: Nombre: Intel(R) Xeon(R) CPU @ 2.00GHz Arquitectura: $X86_64 Frecuencia$: $[2000144000, 0] Núcleos físicos : 2 Device 0 Name : Tesla T4$
cargar librerías: 55.45 condiciones iniciales: 24.68avanzo$_d t$: 18,87$save_t xt$: 0,52
dentro de avanzo$_d t metodo_V erlet$: 2,89$where$: 14,84$compilaciódecupy$: 12,44$correccion_T emperatura$: 0,41$restanteotrosprocesos$
————- CPU: Intel(R) Xeon(R) CPU @ 2.00GHz GPU: Nvidia Tesla T4
P1 Importar módulos: 55.45 P2 Condiciones iniciales: 24.68 P3 Copias HtoD and DtoH: 0. P4 Loop temporal: 18.87 S1 Método de Verlet: 2.89 SS1 Cálculo de fuerzas: 2.47 SS2 Integración de posiciones y velocidades: 2.89 - 2.47 S2 Rebotes: 14.84 S3 Corrección de velocidades: 0.41 P5 Guardado de datos: 0.52 ————-
¿Cómo hago profiling de un código escrito en C++? Quiero obtener un archivo del tipo .profile
$\text{cpp}_c pu$
Para hacer profiling:
1. Compilar usando la opción -pg 2. Ejecutarlo g++ -O3 -pg $\text{cpp}_c pu_s imulacion_d e_p articulas.cpp$ $ocpp_c pu_s imulacion_d e_p articulas$
$\text{cpp}_c pu_f unciones.exe$
gprof $\text{cpp}_c pu_s imulacion_d e_p articulasgmon.out > cpp_c pu_s imulacion_d e_p articulas_p rofiling.txt$

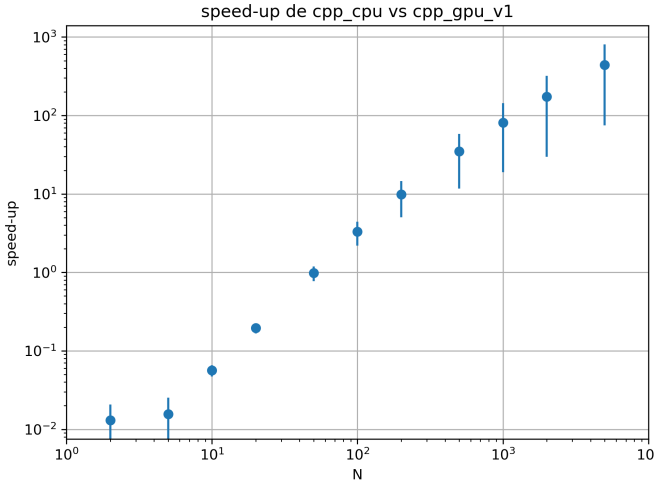

Figura 12: Gráfico de speed-up respecto a cpp$_cpu$

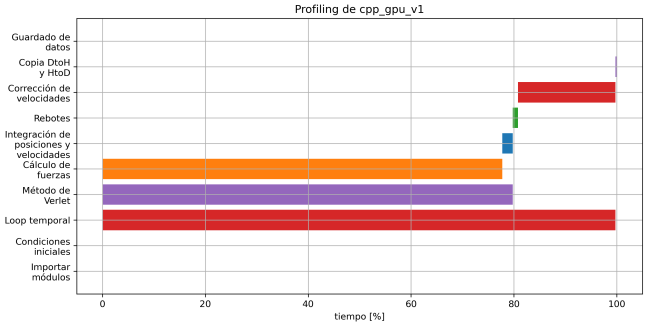

Figura 13: Profiling

Las pruebas se realizaron en una CPU AMD Phenom II X4 955 con una arquitectura de 64 bits. Esta CPU tiene 4 núcleos, con una frecuencia de reloj de 3.21 GHz.

cpp$_gpu_v$1

Las pruebas se realizaron en una CPU AMD Phenom II X4 955 con una arquitectura de 64 bits. Esta CPU tiene 4 núcleos, con una frecuencia de reloj de 3.21 GHz.

Además, se empleó una GPU NVIDIA GeForce GTX TITAN X

cpp$_cpu_v$2

Las pruebas se realizaron en una CPU AMD Phenom II X4 955 con una arquitectura de 64 bits. Esta CPU tiene 4 núcleos, con una frecuencia de reloj de 3.21 GHz.

Además, se empleó una GPU NVIDIA GeForce GTX TITAN X

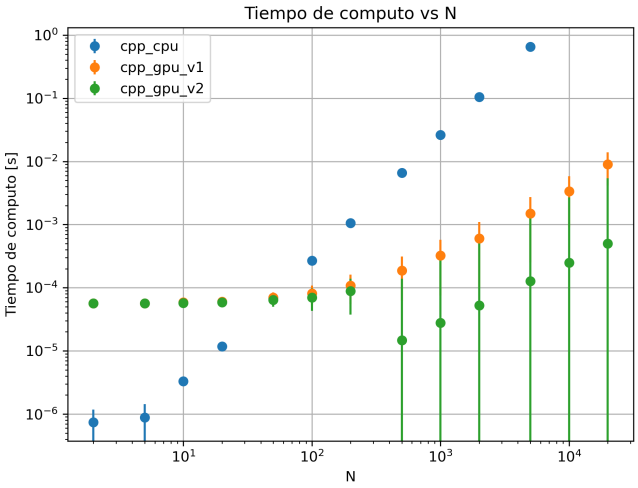

Figura 14: Gráfico de tiempo de cómputo ($\text{cpp}_c u, cpp_g pu_v 1 y cpp_g pu_v 2$)

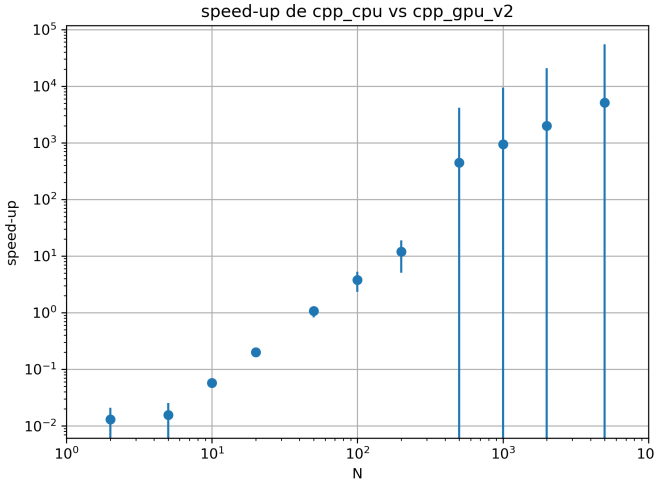

Figura 15: Gráfico de speed-up respecto a $\text{cpp}_c pu$

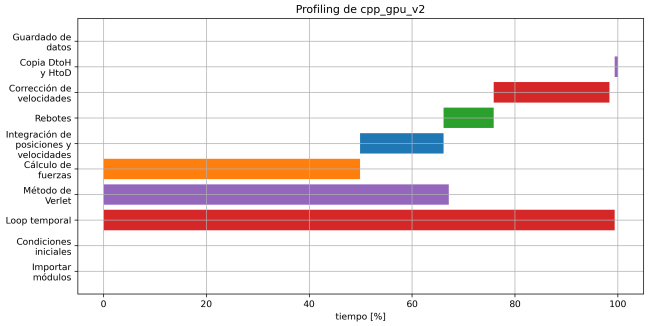

Figura 16: Profiling