# 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 2D de radio $R_0$. En particular, nos interesa conocer las propiedades del gas en el equilibrio.

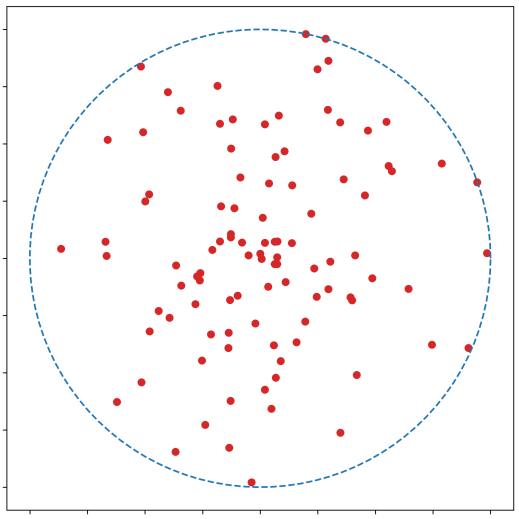

Figura 1: Figura representativa: algunos electrones adentro de un círculo de radio $R_0$. Hacerlo con python y con el mismo formato que los resultados.

## 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

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

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

$$\alpha = \frac{e^2}{mR_0 v_0^2}$$

$$v_0 = \sqrt{\frac{2KT}{m}}$$

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

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

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.

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

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

### 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.

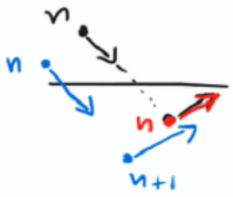


Figura 2: Figura del rebote blando

## 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.

Método de Verlet

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

$$\mathbf{v}_i^{n+1} = \mathbf{v}_i^n + \frac{1}{2}(\mathbf{F}_i^{n+1} + \mathbf{F}_i^n)$$

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
Si cada paso es independiente de los demás, entonces puedo hacer estadística del mismo N usando los pasos de tiempo De alguna manera debería "checkear" el resultado. Supongo que lo haré observando la conservación de la energía
Comentario: Tratar de entender por qué no anda el código de shared memory y ahora sí: antes estaba haciendo un for sobre partículas que no existían

## 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.
- CÓDIGO: 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
- Solo pudimos ejecutar el código hasta un máximo de $N = 1000$ porque para tamaños superiores le toma más de 20' en ejecutar.
- Vemos que a mayor tamaño, mayor tiempo de cómputo, lo cual es razonable porque estamos utilizando una CPU. Si vemos hacemos profiling del código, vemos que gran parte del tiempo se dedica al loop temporal.

La matriz de fuerzas es

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

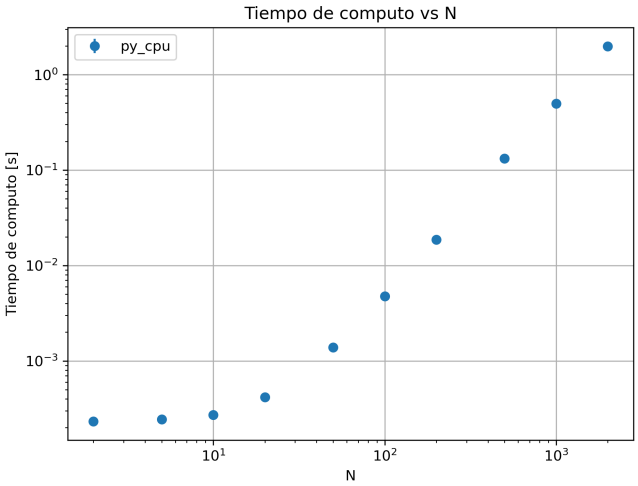


Figura 3: Gráfico de tiempo de cómputo

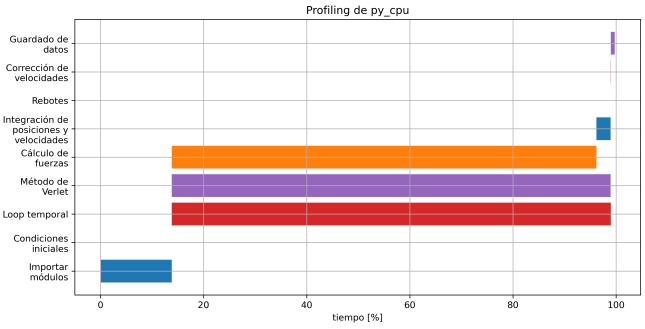


Figura 4

### 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"
- Si analizamos el tiempo de cómputo con el tamaño del sistema, vemos que inicialmente tarda más el código paralelizado, pero cuando el tamaño aumenta se mantiene constante el tiempo de cómputo. Cuando se consideran alrededor de 1000 partículas, la GPU serializa y el tiempo de cómputo comienza a crecer, pero aún así manteniéndose órdenes de magnitud por debajo del código en serie.
- Si analizamos el speed-up definido como el tiempo de cómputo del código en serie vs el del código en paralelo, observamos que para un tamaño de $N = 1000$ partículas, tenemos un seed-up del $x100$
- Si vemos el profiling realizado para $N = 100$, vemos que gran parte del tiempo está destinado a importar módulos y a la compilación.

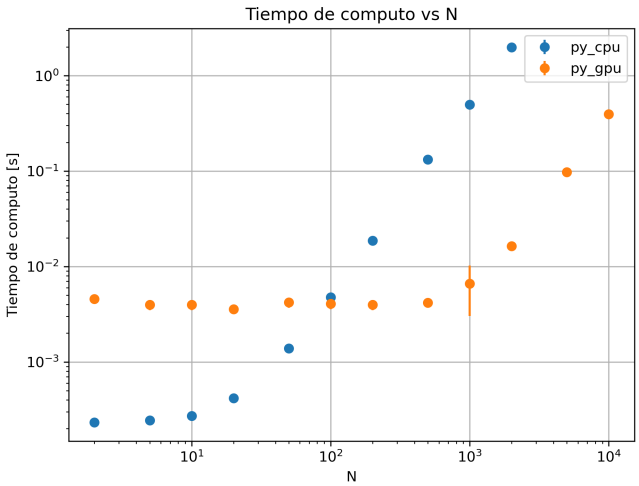


Figura 5: Gráfico de tiempo de cómputo ($\text{py}_c uy \text{py}_g pu$)

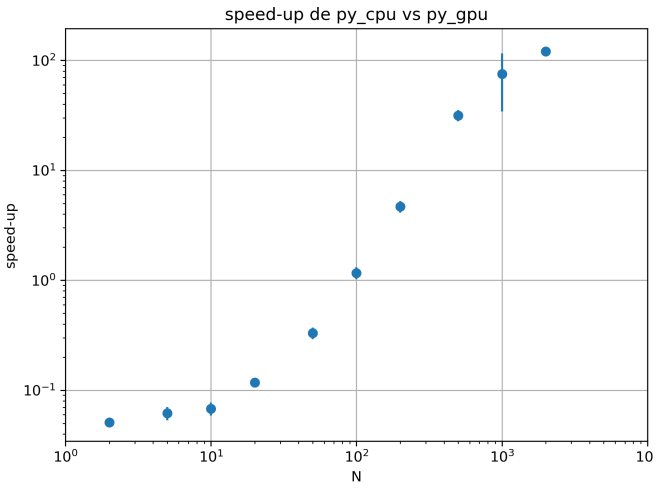


Figura 6: Gráfico de speed-up respecto a $\text{py}_c pu$

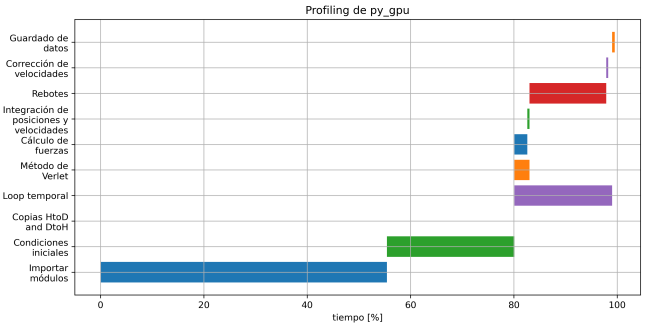


Figura 7

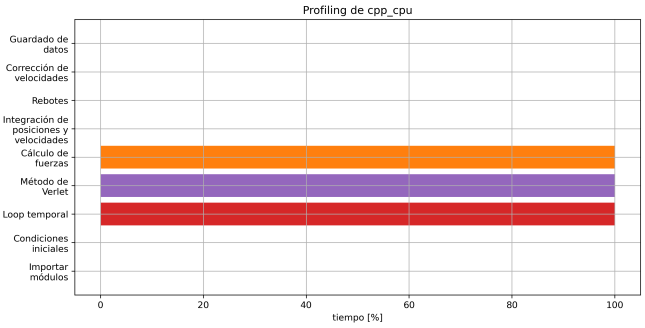


Figura 9

### C. Versión 3: C++ en serie

- Usé los loops de C++ para todos los cálculos. Las fuerzas ya no se calculan mediante una matriz, sino como un doble loop sobre pares de partículas.
- Poner la función de fuerzas
- Ver qué CPU estoy usando y análisis
- El tiempo de cómputo con C++ se reduce considerablemente respecto a Python. Pero al igual que este aumenta con el tamaño $N$ del sistema. Si vemos el profiling sin tener en cuenta los tiempos de compilación, vemos que casi el 100 % del tiempo de cómputo se debe al loop temporal y, en particular, al cálculo de fuerzas. Con el código de python paralelizado habíamos logrado disminuir este tiempo de cómputo, veamos si podemos hacerlo nuevamente pero en C++.

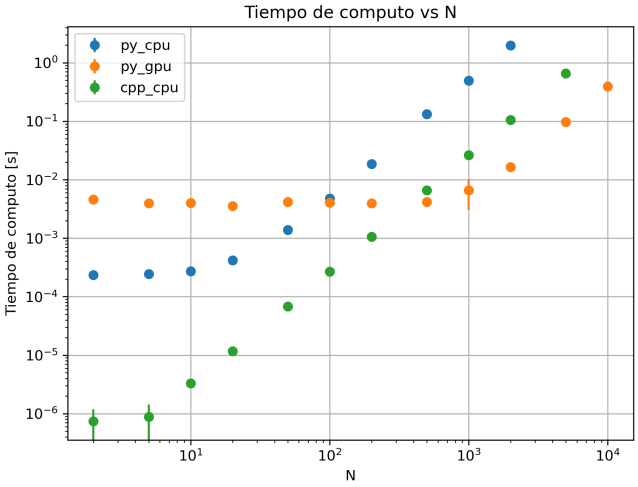


Figura 8: Gráfico de tiempo de cómputo ($\mathrm{py}_c u, py_g puycpp_c pu$)

### 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.
- Hice un kernel por cada sección de la evolución, es decir, un kernel para calcular las fuerzas, un par para aplicar el método de Verlet, otro para los rebotes y otro para corregir las velocidades. Todos son kernels simples que paralelizan un único loop, salvo el que calcula las fuerzas que paraleliza 2 loops. Veamos este en más detalle
- CÓDIGO: kernel bodyForce
- Volver a poner los items al inicio de ïmplementaciónz poner al lado de cada item una flecha y la declaración de los kernels
- En cuanto al tiempo de cómputo vemos el mismo comportamiento que en python: en la versión en serie el tiempo de cómputo aumenta con el tamaño del sistema, mientras que en la versión en paralelo, al inicio se mantiene constante y mayor a la versión en serie y luego comienza a serializar.
- Analizando el speed-up, vemos que con un tamaño del orden de 2000 partículas, tenemos un speed-up de $x500$
- Si hacemos profiling del código y sin considerar el tiempo de compilación, vemos que al igual que antes toda la ejecución radica en el loop temporal, pero a diferencia de antes, se logró disminuir el tiempo de cómputo relativo del cálculo de fuerzas
- . Ver qué CPU y GPU estoy usando y análisis
-

### E. Versión 5: C++ en paralelo v2

- Esta versión en paralelo admite una última optimización, que se basa en usar memoria compartida.
- CÓDIGO: bodyForce
- Vemos que el tiempo de cómputo es consistentemente menor que la última versión, obteniendo un speed-up ligeramente mayor. Además, vemos que el error asociado al tiempo de cómputo aumenta, Cómo explico

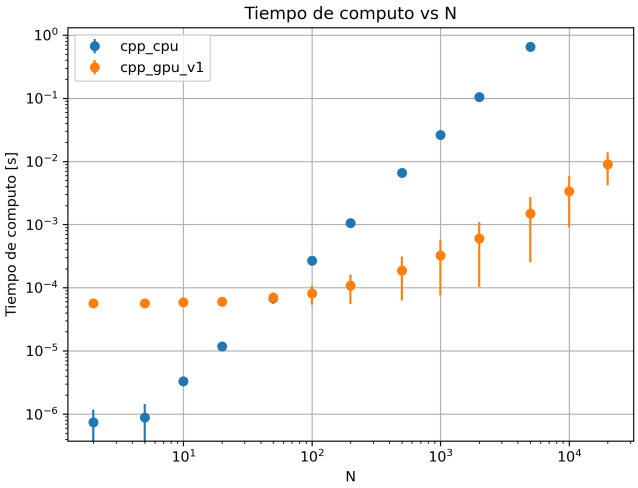


Figura 10: Gráfico de tiempo de cómputo ($\text{cpp}_cpu, cpp_gpu_v1$)

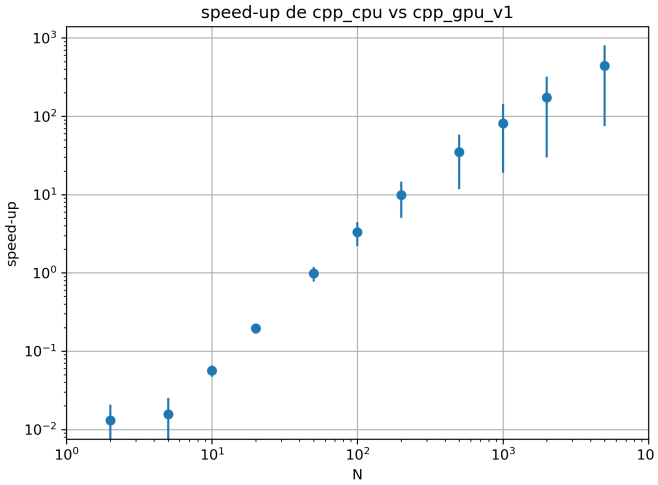


Figura 11: Gráfico de speed-up respecto a $\text{cpp}_cpu$

esto?

- Analizando el profiling, vemos que logramos disminuir aún más el tiempo de cómputo del cálculo de fuerzas.
- . Ver qué CPU y GPU estoy usando y análisis

## VII. RESULTADOS

Veamos cómo es la evolución de la densidad
-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

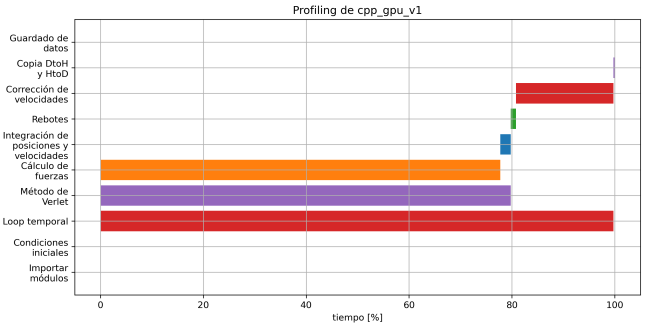


Figura 12: Profiling

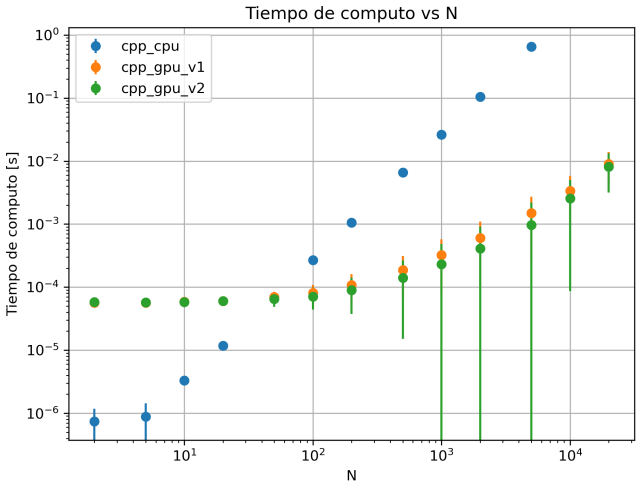


Figura 13: Gráfico de tiempo de cómputo ($\text{cpp}_cu, cpp_gpu_v1 y cpp_gpu_v2$)

$\text{py}_cpu$ :
Para hacer profiling: jupyter nbconvert $\text{py}_cpu_simulacion_de_particulas.ipynb$ – $-topythonpython$ – $mcProfile$ – $opy_cpu_simulacion_de_particulas.profilepy_cpu_simulacion_de_particul$
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
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 ———-
$\text{py}_gpu$ : $Parahacerprofiling$ : $jupyternbconvertpy_gpu_simulacion_de_particulas.ipynb-$ $-topythonpython$ – $mcProfile$ – $opy_gpu_simulacion_de_particulas.profilepy_gpu_simulacion_de_particul$

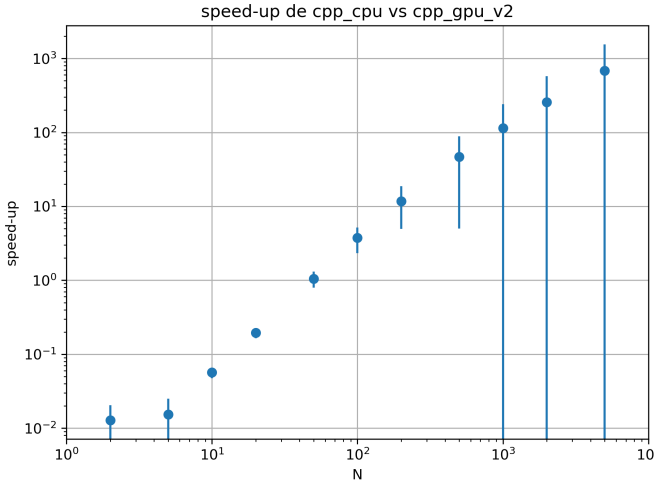


Figura 14: Gráfico de speed-up respecto a cpp$_c pu$

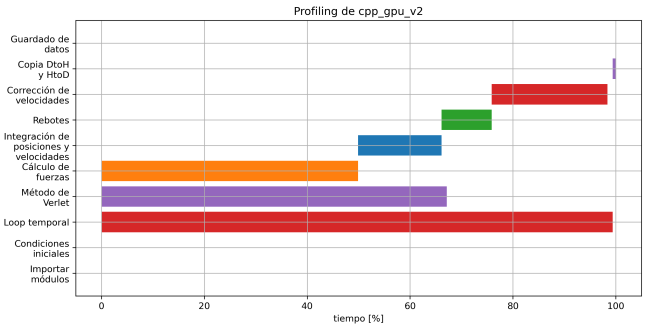


Figura 15: Profiling

Información de la CPU: Nombre: Intel(R) Xeon(R) CPU @ 2.00GHz Arquitectura: X86$_6$4$Frecuencia$ : $[2000144000, 0]N\acute{u}cleosf\acute{i}sicos : 2Device0Name : TeslaT4$ cargar librerías: 55.45 condiciones iniciales: 24.68avanzo$_d t$ : 18,87$save_t xt$ : 0,52 dentro de avanzo$_d tmetodo_V erlet$ : 2,89$where$ : 14,84$compilaci\acute{o}ndecupy$ : 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

cpp$_c pu$

Para hacer profiling:

1. Compilar usando la opción-pg 2. Ejecutarlo g++ -O3 -pg cpp$_c pu_f unciones.cppcpp_c pu_s imulacion_d e_p articulas.cpp-ocpp_c pu_s imulacion_d e_p articulas$

cpp$_c pu_f unciones.exe$

gprof cpp$_c pu_s imulacion_d e_p articulasgmon.out >$ $cpp_c pu_s imulacion_d e_p articulas_p rofiling.txt$

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$_g pu_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$_c pu_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

## VIII. COMENTARIOS EXTRAS DE LO QUE HICE/NO HICE

Agregar *Justificación de py$_g pucteparaNchicoymayorquepy_c puSiemprehayuncostoasociad$ $Verificarquecorraenelcluster * Releerelpdfdelspeech$

*Nuevo profiling de cpp$_g puconsiderandocopiasHtoDyDtoH$

*Graficar evolución con cpp$_g pu_v$2 * $Graficarelpromediodeladensidadradialenelequilibrio*$ $Fijarmesilosresultadosdecpp_g pu_v 2soncorrectos$

Si yo quisiera simular $10^4 partículas, necesitaríaPorlotanto,$"$0x28000$"$endecimalesigua$

¿Qué sol existe? dynamic shared memory

Muy complicado

Otro modo: To handle this case, the simulation would need to be split into multiple runs of the kernel function, with each run processing a subset of the particles

Podría si no iterar en cada kernel, pero ya estoy trabajando en serie

CUDA:

Profiling Determinar qué hay que medir

*Serie C++ .Buscar con ChatGPT cómo hacer profiling

*Paralelo C++ .Buscar con ChatGPT cómo hacer profiling