Un Estudio Comparativo entre Traductores de Python para Aplicaciones Paralelas de Memoria Compartida

Autor: Andrés Milla

Director: Enzo Rucci

Tesina de grado correspondiente a la carrera **Licenciatura en Informática**
Facultad de Informática de la **Universidad Nacional de La Plata**
Marzo 2022

Agenda

1. Motivación
2. Objetivos
3. N-Body
4. Optimización de N-body usando CPython y PyPy
5. Optimización de N-body usando Numba
6. Optimización de N-body usando Cython
7. Comparación de prestaciones
8. Conclusiones y trabajos futuros

1

Motivación

Motivación

➡ En la actualidad, Python se ha convertido en uno de los lenguajes más populares

TIOBE Index for February 2022

February Headline: TIOBE index top 3 benefits from technology changes

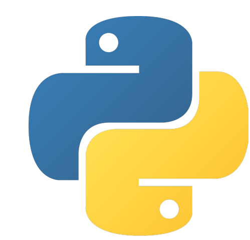
As of the 1st of May, the Alexa web traffic ranking engine is going to stop its services. Alexa was used to select the search engines for the TIOBE index until now. So now something has to change. Similarweb has been chosen as the alternative for Alexa. We have used Similarweb for the first time this month to select search engines and fortunately, there are no big changes in the index due to this swap. The only striking difference is that the top 3 languages, Python, C, and Java, all gained more than 1 percent in the rankings. We are still fine-tuning the integration with Similarweb, which is combined with a shift to HtmlUnit in the back-end. Some websites are not onboarded yet, but will follow soon. Now that HtmlUnit is applied for web crawling, it will become possible to add more sites to the index, such as Stackoverflow and Github. This will hopefully happen in the next few months. *--Paul Jansen CEO TIOBE Software*

The TIOBE Programming Community index is an indicator of the popularity of programming languages. The index is updated once a month. The ratings are based on the number of skilled engineers world-wide, courses and third party vendors. Popular search engines such as Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Baidu are used to calculate the ratings. It is important to note that the TIOBE index is not about the *best* programming language or the language in which *most lines of code* have been written.

The index can be used to check whether your programming skills are still up to date or to make a strategic decision about what programming language should be adopted when starting to build a new software system. The definition of the TIOBE index can be found here.

Feb 2022	Feb 2021	Change	Programming Language	Ratings	Change
1	3	^	Python	15.33%	+4.47%
2	1	v	C	14.08%	-2.26%
3	2	v	Java	12.13%	+0.84%

Motivación

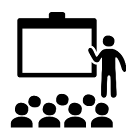


- **Python** es un lenguaje:
  - Alto nivel
  - Interpretado
  - Dinámico
  - Multi-paradigma

- Presenta las siguientes **cualidades**:
  - Notable poder de programación
  - Sintaxis clara y limpia
  - Menor esfuerzo de programación en comparación a otros lenguajes

- Ampliamente utilizado en las siguientes **áreas**:

Desarrollo web



Educación

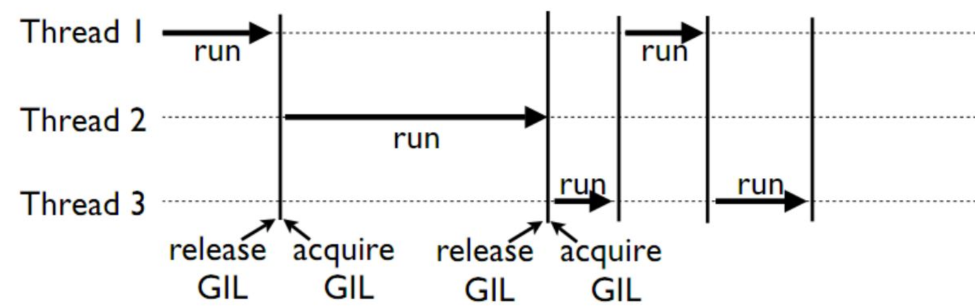
Videojuegos

Inteligencia artificial

Computación científica

Motivación

- Problema de Python → **Tiempo de ejecución lento**, debido a:
  - Su naturaleza de lenguaje interpretado
  - Limitaciones para implementar código multi-hilado paralelo → **GIL** (Global Interpreter Lock)
    - *Componente de CPython (intérprete oficial de Python)*
    - Asegura que a lo sumo 1 hilo esté ejecutándose en un instante de tiempo
    - Afecta fuertemente a los procesos **CPU-Bound**
    - Indiferente para procesos **IO-Bound**



Motivación

➦ Surgen diferentes **soluciones** para acelerar el cómputo, cada una con un enfoque diferente y con su propia relación de costo-rendimiento:

1 Procesos

- A través del módulo **multiprocessing**.
- Mayor consumo de recursos.
- Espacio de direcciones distribuido → Mayor esfuerzo de programación.
- Permite “esquivar” al GIL.

2 Traductores

- **Cython** → Compilador AOT que permite utilizar código y librerías de C como OpenMP.
- **PyPy** → Compilador JIT que acelera código numérico.
- **Jython** → Implementación en Java de Python2 (deprecado).
- **Numba** → Compilador JIT que traduce Python en código de máquina optimizado.

3 Módulos externos

- **NumPy** → Paquete dedicado a la computación científica y numérica.

Permiten desactivar el GIL y realizar una ejecución paralela

Motivación

- Para no tomar una decisión a “ciegas”, se exploraron estudios relacionados.
- **Estado del arte:**
  - Lamentablemente, la literatura disponible en la temática no es exhaustiva.
  - Se utilizan **solo** versiones secuenciales → No evalúan paralelismo.
    - I Wilbers, Hans Petter Langtangen y Åsmund Ødegård. ((Using Cython to Speed Up Numerical Python Programs)). En: ene. de 2009, págs. 495-512. isbn: 978-82-519-2421-4.
    - Alexander Roghult. ((Benchmarking Python Interpreters: Measuring Performance of CPython, Cython, Jython and PyPy)). En: 2016.

Motivación

- **Estado del arte:**
  - Lamentablemente, la literatura disponible en la temática no es exhaustiva.
  - En el caso de evaluar paralelismo, lo hacen sobre lenguajes y **no entre traductores**.
    - Jan Gmys y col. ((A comparative study of high-productivity high-performance programming languages for parallel metaheuristics)). en. En: Swarm and Evolutionary Computation 57 (sep. de 2020)
    - F. Wilkens. ((Evaluation of performance and productivity metrics of potential programming languages in the HPC environment)). En: 2015.
    - Xing Cai, Hans Petter Langtangen y Halvard Moe. ((On the Performance of the Python Programming Language for Serial and Parallel Scientific Computations)). en. En: Scientific Programming 13.1 (2005)
    - M. Varsha y col. ((A Review of Existing Approaches to Increase the Computational Speed of the Python Language)). en. En: International Journal of Research in Engineering, Science and Management (2019).

Motivación

- **Estado del arte:**
  - Lamentablemente, la literatura disponible en la temática no es exhaustiva.
  - En la mayoría de ellos no se hace un estudio sobre la **productividad** y el **esfuerzo de programación** que contrae desarrollar cada solución.

2

Objetivo

Objetivo

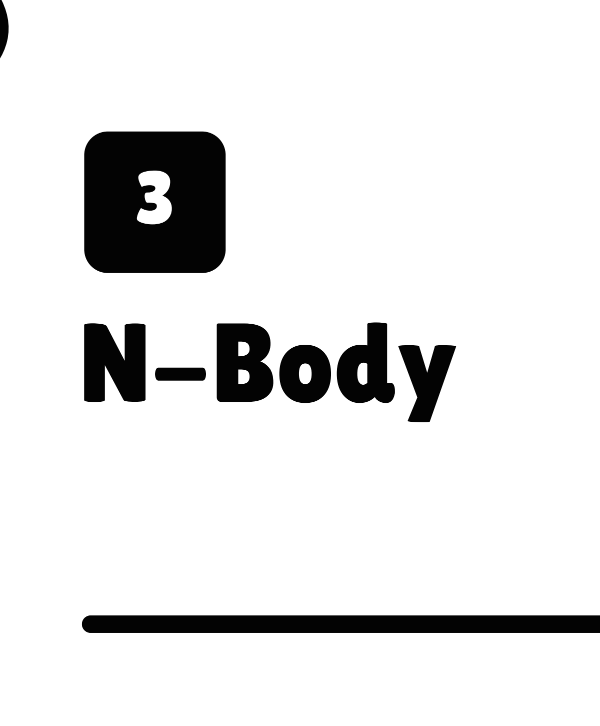
➡ **Comparar** las prestaciones (**rendimiento y esfuerzo de programación**) de **traductores de Python** en una arquitectura multicore, utilizando como caso de estudio el **problema de N Cuerpos computacionales con atracción gravitacional**.

Objetivo

- **Comparar** las prestaciones (**rendimiento y esfuerzo de programación**) de **traductores de Python** en una arquitectura multicore, utilizando como caso de estudio el **problema de N Cuerpos computacionales con atracción gravitacional**.

  Elegido por:

  - *CPU-Bound*
  - Paralelizable
  - Complejidad temporal de $O(n^2)$

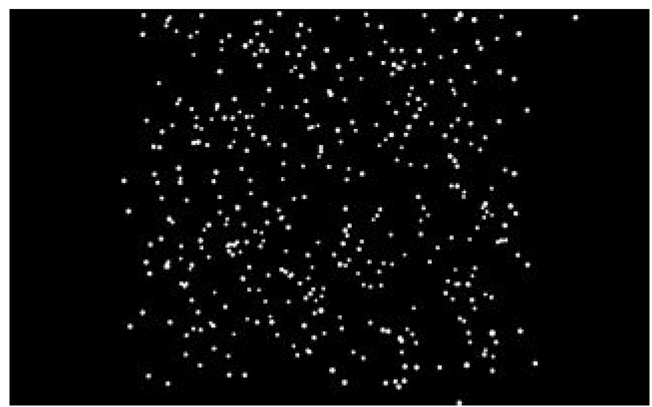


3

N-Body

N-Body

- **Problema**: Simular la evolución de un sistema compuesto por N cuerpos durante una cantidad de tiempo dada
  - Para cada cuerpo se conoce su masa y su estado inicial (posición y velocidad)
  - En cada instante de tiempo discreto, todo cuerpo experimenta una aceleración que surge de la atracción gravitacional, lo que afecta a su estado



N-Body

- **Fundamentos físicos**
  - **Ley de gravitación universal de Newton** → Fuerzas de atracción gravitacional
    - **N = 2** → $F = \frac{G \times m_1 \times m_2}{r^2}$
    - **N > 2** → Sumatoria de las fuerzas ejercidas por los demás cuerpos.
  - **Segunda ley de Newton** → Cálculo de la aceleración

$$a = \frac{F}{m}$$

  - **Movimiento de la mecánica Newtoniana** → Tasa de cambio de velocidad

$$dv_i = a_i dt$$

  - **Método de integración *velocity verlet*** → Desplazamiento de los cuerpos

$$dp_i = \left(v_i + \frac{dv_i}{2}\right) dt$$

N-Body

- Pseudocódigo de la versión directa (también llamada **all-pairs**):

```
Para cada cuerpo de i = 1 a N:

    Para cada cuerpo de j = 1 a N:
        Calcular la fuerza ejercida de j sobre i.
        Sumar las fuerzas que afectan a i.

    Calcular desplazamiento del cuerpo i.

Para cada cuerpo de i = 1 a N:
    Mover cuerpo i.
```

N-Body

- Pseudocódigo de la versión directa (también llamada **all-pairs**):

```
Para cada cuerpo de i = 1 a N:

    Para cada cuerpo de j = 1 a N:
        Calcular la fuerza ejercida de j sobre i.
        Sumar las fuerzas que afectan a i.

    Calcular desplazamiento del cuerpo i.

Para cada cuerpo de i = 1 a N:
    Mover cuerpo i.
```

- **Complejidad temporal → O(n^2)**

N-Body

- Pseudocódigo de la versión directa (también llamada **all-pairs**):

```
Para cada cuerpo de i = 1 a N:

    Para cada cuerpo de j = 1 a N:
        Calcular la fuerza ejercida de j sobre i.
        Sumar las fuerzas que afectan a i.

    Calcular desplazamiento del cuerpo i.

Para cada cuerpo de i = 1 a N:
    Mover cuerpo i.
```

El cálculo de fuerzas depende de iteraciones anteriores

Para mover al cuerpo se necesita calcular la fuerza previamente

- Complejidad temporal → $O(n^2)$
- **Paralelizable → Dependencias de datos**

N-Body

- Pseudocódigo de la versión directa (también llamada **all-pairs**):

```
Para cada cuerpo de i = 1 a N:

    Para cada cuerpo de j = 1 a N:
        Calcular la fuerza ejercida de j sobre i.
        Sumar las fuerzas que afectan a i.

    Calcular desplazamiento del cuerpo i.

Para cada cuerpo de i = 1 a N:
    Mover cuerpo i.
```

- Complejidad temporal → $O(n^2)$
- Paralelizable → Dependencias de datos
- *CPU-Bound*

4
Optimización de N-Body usando CPython y PyPy

Optimización de N-Body usando CPython y PyPy

- **CPython:**
  - Intérprete oficial de Python

- **PyPy:** pypy
  - **Compilador JIT** cuyo objetivo es acelerar código numérico. Asegura:
    - Incrementar la velocidad de ejecución
    - Disminuir memoria utilizada de los programas escritos en Python
  - **Limitaciones**:
    - Posee GIL.
    - No optimiza paquetes que están escritos en otros lenguajes → No puede utilizarse con NumPy.

Optimización de N-Body usando CPython y PyPy

- **Implementaciones**:
  - Enfoque incremental
    - Se comenzó con una versión secuencial → **naive**
    - Se exploraron diferentes optimizaciones de acuerdo a los resultados encontrados

Optimización de N-Body usando CPython y PyPy

➡ **Versión naive:**

```python
def nbody(
    N, D,
    positions_x, positions_y, positions_z,
    masses,
    velocities_x, velocities_y, velocities_z,
    dp_x, dp_y, dp_z,
):
    for _ in range(D):
        for i in range(N):
            forces_x = forces_y = forces_z = 0.0

            for j in range(N):
                dpos_x = positions_x[j] - positions_x[i]
                dpos_y = positions_y[j] - positions_y[i]
                dpos_z = positions_z[j] - positions_z[i]

                dsquared = (
                    dpos_x ** 2.0 + dpos_y ** 2.0 + dpos_z ** 2.0 + SOFT
                )

                gm = GRAVITY * masses[j] * masses[i]

                d32 = dsquared ** -1.5

                forces_x += gm * d32 * dpos_x
                forces_y += gm * d32 * dpos_y
                forces_z += gm * d32 * dpos_z

            aceleration_x = forces_x / masses[i]
            aceleration_y = forces_y / masses[i]
            aceleration_z = forces_z / masses[i]

            velocities_x[i] += aceleration_x * DT / 2.0
            velocities_y[i] += aceleration_y * DT / 2.0
            velocities_z[i] += aceleration_z * DT / 2.0

            dp_x[i] = velocities_x[i] * DT
            dp_y[i] = velocities_y[i] * DT
            dp_z[i] = velocities_z[i] * DT

        for i in range(N):
            positions_x[i] += dp_x[i]
            positions_y[i] += dp_y[i]
            positions_z[i] += dp_z[i]
```

Para cada instante discreto de tiempo de la simulación

```
Para cada cuerpo de i = 1 a N:
    Para cada cuerpo de j = 1 a N:
        Calcular la fuerza ejercida de j sobre i.
        Sumar las fuerzas que afectan a i.
    Calcular desplazamiento del cuerpo i.
Para cada cuerpo de i = 1 a N:
    Mover cuerpo i.
```

Pseudocódigo de **N-Body**

Optimización de N-Body usando CPython y PyPy

- NumPy: NumPy
  - Paquete dedicado a la computación científica y numérica.
  - Características principales:
    - Provee **utilidades para el cómputo científico** → funciones matemáticas, módulos para álgebra lineal, etc.
    - Permite realizar **operaciones multidimensionales entre arreglos**, denominadas *broadcasting*.

Arreglos 1D con listas de Python

```
dpos_x = positions_x[j] - positions_x[i]
dpos_y = positions_y[j] - positions_y[i]
dpos_z = positions_z[j] - positions_z[i]
```

→

Arreglos 2D con *arrays* de NumPy

```
dpos = positions - positions[i]
```

- Permite controlar la **organización de memoria** de los arreglos.
- Su **núcleo** está escrito y optimizado **en C** → Incrementa la velocidad de Python.

Optimización de N-Body usando CPython y PyPy

➡ Integración de NumPy:

```
def nbody(
    N, D,
    positions_x, positions_y, positions_z,
    masses,
    velocities_x, velocities_y, velocities_z,
    dp_x, dp_y, dp_z,
):
    for _ in range(D):
        for i in range(N):
            forces_x = forces_y = forces_z = 0.0

            for j in range(N):
                dpos_x = positions_x[j] - positions_x[i]
                dpos_y = positions_y[j] - positions_y[i]
                dpos_z = positions_z[j] - positions_z[i]

                dsquared = (
                    dpos_x ** 2.0 + dpos_y ** 2.0 + dpos_z ** 2.0 + SOFT
                )

                gm = GRAVITY * masses[j] * masses[i]

                d32 = dsquared ** -1.5

                forces_x += gm * d32 * dpos_x
                forces_y += gm * d32 * dpos_y
                forces_z += gm * d32 * dpos_z

            aceleration_x = forces_x / masses[i]
            aceleration_y = forces_y / masses[i]
            aceleration_z = forces_z / masses[i]

            velocities_x[i] += aceleration_x * DT / 2.0
            velocities_y[i] += aceleration_y * DT / 2.0
            velocities_z[i] += aceleration_z * DT / 2.0

            dp_x[i] = velocities_x[i] * DT
            dp_y[i] = velocities_y[i] * DT
            dp_z[i] = velocities_z[i] * DT

        for i in range(N):
            positions_x[i] += dp_x[i]
            positions_y[i] += dp_y[i]
            positions_z[i] += dp_z[i]
```

Arreglos de NumPy

Optimización de N-Body usando CPython y PyPy

➡ **Broadcasting:**

```
def nbody(N, D, positions, masses, velocities, dp):
    for _ in range(D):
        for i in range(N):
            dpos = np.subtract(positions, positions[i])
            dsquared = (dpos ** 2.0).sum(axis=1) + SOFT
            gm = masses * (masses[i] * GRAVITY)
            d32 = dsquared ** -1.5
            gm_d32 = (gm * d32).reshape(N, 1)

            forces = np.multiply(gm_d32, dpos).sum(axis=0)

            acceleration = forces / masses[i]
            velocities[i] += acceleration * DT / 2.0
            dp[i] = velocities[i] * DT

        for i in range(N):
            positions[i] += dp[i]
```

Arreglos 2D

➡ Reemplazo de operaciones tradicionales entre arreglos por *broadcasting*.

Optimización de N-Body usando CPython y PyPy

➡ **Procesamiento por bloques:**

```
def nbody(N, D, positions, masses, velocities, dp):

    for _ in range(D):
        for first in range(0, N, BLOCKSIZE):
            last = first + BLOCKSIZE
            forces = np.zeros((BLOCKSIZE, 3), dtype=DATATYPE)

            for j in range(N):
                dpos = np.subtract(positions[j], positions[first:last])

                dsquared = (dpos ** 2.0).sum(axis=1) + SOFT
                gm = masses[first:last] * (masses[j] * GRAVITY)
                d32 = dsquared ** -1.5
                gm_d32 = (gm * d32).reshape(BLOCKSIZE, 1)

                forces += np.multiply(gm_d32, dpos)

            aceleration = forces / masses[first:last].reshape(BLOCKSIZE, 1)
            velocities[first:last] += aceleration * DT / 2.0

            dp[first:last] = velocities[first:last] * DT

        for first in range(0, N, BLOCKSIZE):
            last = first + BLOCKSIZE
            positions[first:last] += dp[first:last]
```

➡ Objetivo: Minimizar el tráfico a memoria principal utilizando la caché de forma más adecuada.

➡ Se separó el bucle **i** en dos particiones:

1. La primera partición se coloca del *bucle j* → Calcula la **fuerza de atracción gravitacional de Newton.**
2. La segunda partición computa la **integración de Verlet**.

Optimización de N-Body usando CPython y PyPy

- **Threading** → Módulo para implementar *multi-threading* en Python.
  - **Creación**:

```
barrier = Barrier(T)
chunks = np.array_split(np.arange(N), T)

threads = [
    Thread(target=nbody, args=(N, D, *arrays, chunk, barrier))
    for chunk in chunks
]
```

División de carga de trabajo en partes iguales entre los T hilos.

Función a ejecutar por los T hilos.

  - **Ejecución** → Modelo *fork-join*:

```
for t in threads:
    t.start()

for t in threads:
    t.join()
```

Optimización de N-Body usando CPython y PyPy

- **Threading:**
  - Función que ejecutan los T hilos:

```
def nbody(N, D, positions, masses, velocities, dp, chunk, barrier):
    for _ in range(D):
        for i in chunk:
            dpos = np.subtract(positions, positions[i])

            dsquared = (dpos ** 2.0).sum(axis=1) + SOFT
            gm = masses * (masses[i] * GRAVITY)
            d32 = dsquared ** -1.5
            gm_d32 = (gm * d32).reshape(N, 1)

            forces = np.multiply(gm_d32, dpos).sum(axis=0)

            acceleration = forces / masses[i]
            velocities[i] += acceleration * DT / 2.0
            dp[i] = velocities[i] * DT

        barrier.wait()

        for i in chunk:
            positions[i] += dp[i]

        barrier.wait()
```

Cálculos sobre la carga de trabajo correspondiente

Dependencias de datos:

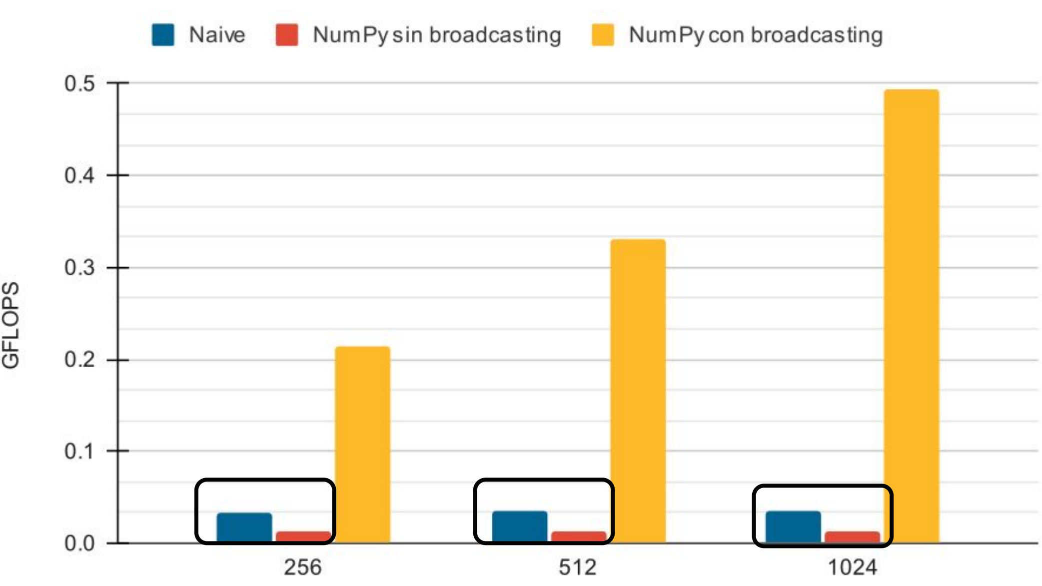
1º Esperar cálculo de fuerza
2º Esperar actualización de posiciones

Optimización de N-Body usando CPython y PyPy

- **Resultados experimentales**
  - **Plataforma**
    - Servidor Intel con 2× Xeon Platinum 8276 de 28 núcleos (2 hilos hw por núcleo) y 256 GB de memoria RAM → *112 hilos en total*.
    - El sistema operativo fue Ubuntu 20.04.2 LTS
    - Python v3.8.10 **-** NumPy v1.20.1 **-** PyPy v7.3.1 .
  - **Parámetros**
    - **N** = {256, 512, 1024, 2048, 4096, 8192}
    - **I** = {100}
    - **T** = {2,4,8,16}
  - **Métrica de rendimiento:**
    - **GFLOPS** → $$GFLOPS = \frac{20 \times N^2 \times I}{t \times 10^9}$$
  - Cada optimización propuesta, fue aplicada y evaluada incrementalmente a partir de la versión *naive*.

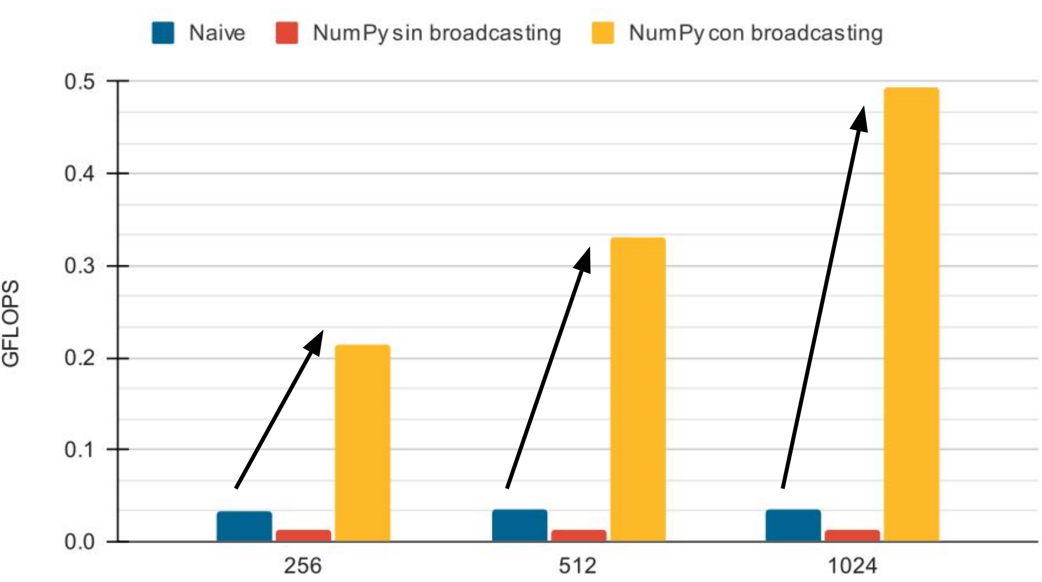
Optimización de N-Body usando CPython y PyPy

- **Rendimiento obtenido con la incorporación de NumPy al variar N.**
  - Rendimiento *sin broadcasting* empeora 2.9× en promedio → **Conversiones innecesarias**



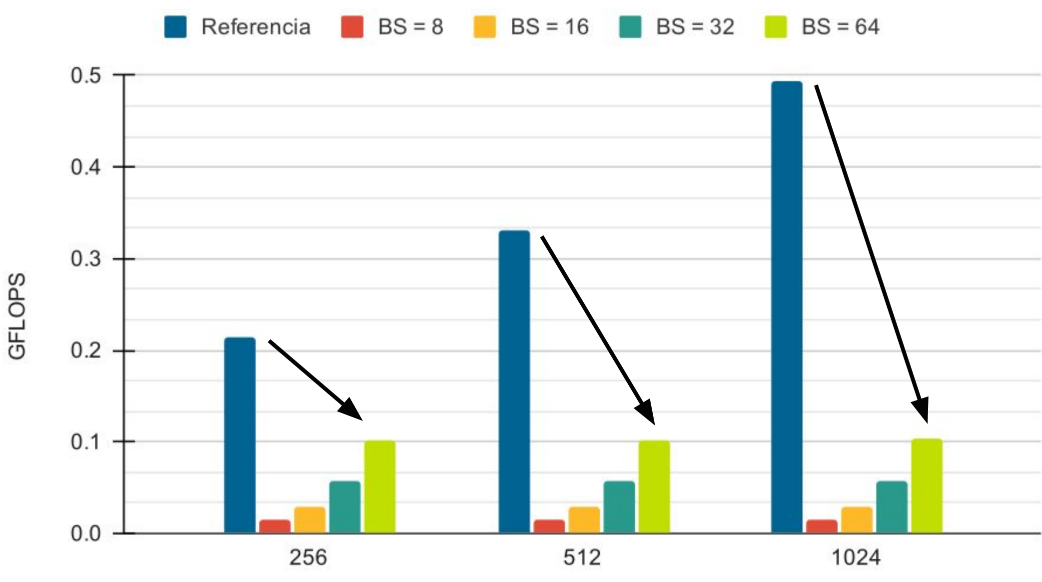
Optimización de N-Body usando CPython y PyPy

- **Rendimiento obtenido con la incorporación de NumPy al variar N.**
    - Rendimiento *con broadcasting* mejora 10× en promedio → **Operaciones directas en el núcleo de NumPy**



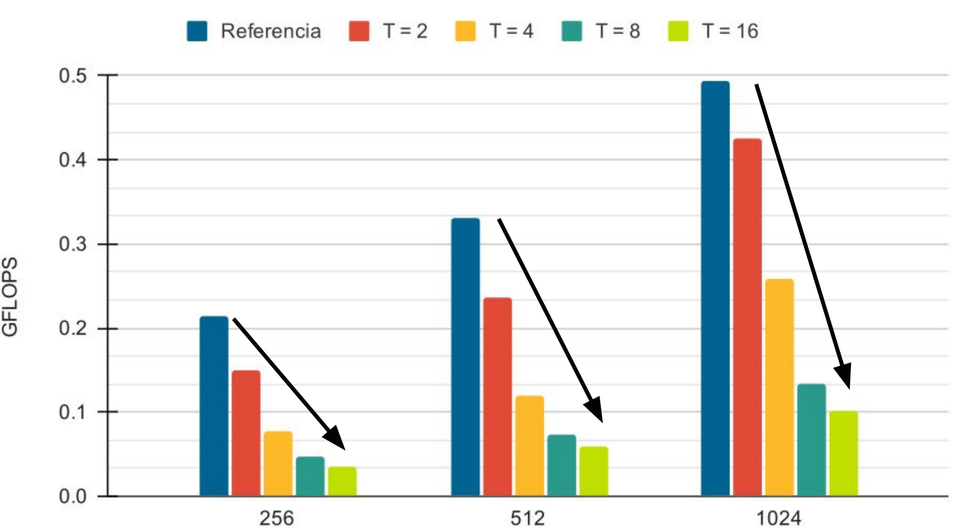
Optimización de N-Body usando CPython y PyPy

- **Rendimiento obtenido del procesamiento de a bloques al variar N y BS (tamaño del bloque).**
  - Rendimiento empeora 3.4× en promedio



Optimización de N-Body usando CPython y PyPy

- **Rendimiento obtenido de la solución multi-hilada utilizando threading al variar N y T (número de hilos).**
  - Empeoró notablemente el rendimiento →**Efecto del GIL**
    - No permite la ejecución paralela de los hilos
    - *Overhead* al liberar y adquirir el *lock* constantemente

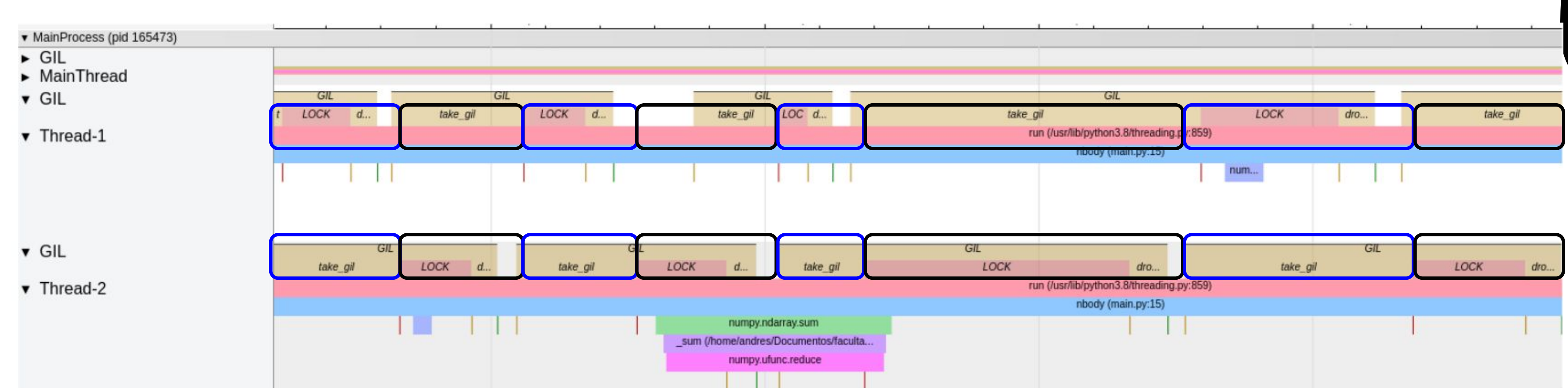


Optimización de N-Body usando CPython y PyPy

➦ **Captura del comportamiento del GIL** con la herramienta *per4m*.

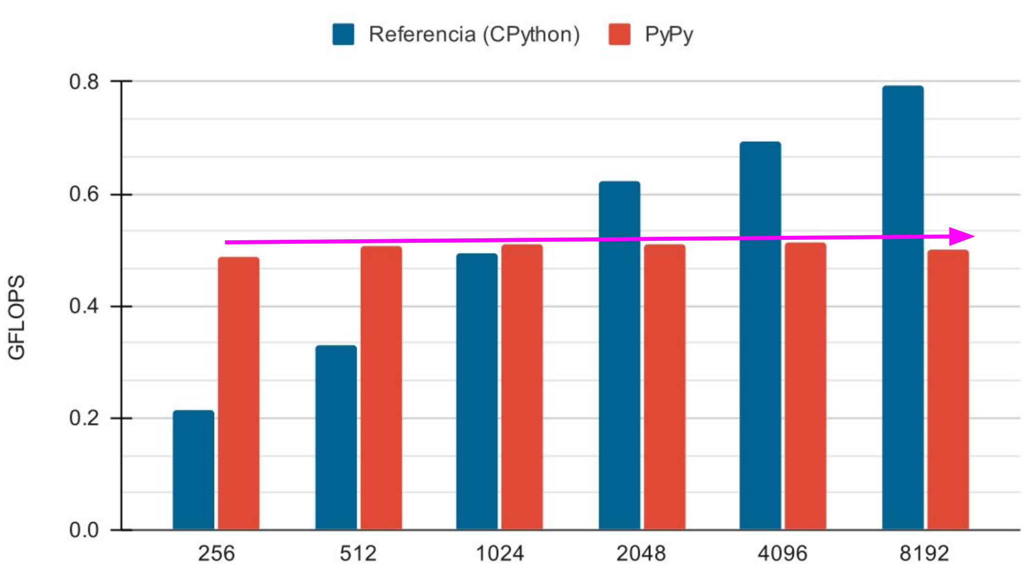
➥ Sólo 1 thread se está ejecutando “a la vez”.

➥ Se adquiere el *lock* reiteradas veces.



Optimización de N-Body usando CPython y PyPy

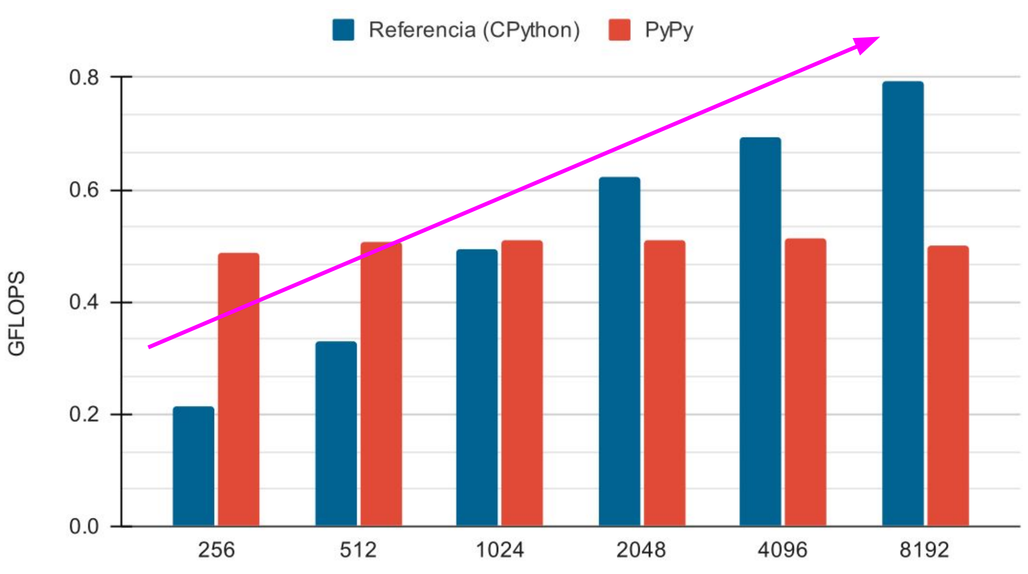
- **Rendimiento obtenido utilizando CPython y PyPy al variar N.**
  - Rendimiento constante de PyPy



Optimización de N-Body usando CPython y PyPy

➦ **Rendimiento obtenido utilizando CPython y PyPy al variar N.**

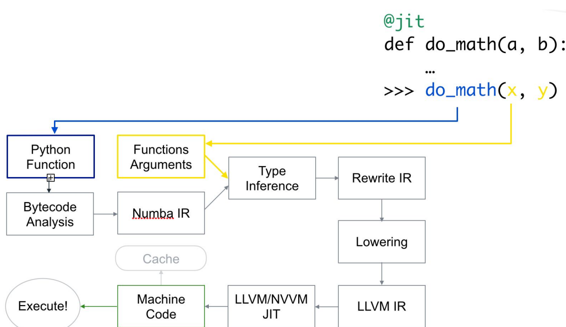
- ➥ Rendimiento de CPython tiende a crecer a medida que el tamaño aumenta
  - ◈ Superior a PyPy en **N** relativamente grandes.



5
Optimización de
N-Body usando
Numba

Optimización de N-Body usando Numba

- Numba: Numba
  - **Compilador JIT** que permite traducir código Python a código de máquina optimizado a través de **LLVM**.
  - Utiliza decoradores para intervenir lo menos posible en el código del programador.
  - Es capaz de acercarse a rendimientos de C, C++ y Fortran.
  - Permite desactivar el **GIL**.



Optimización de N-Body usando Numba

- Numba: Numba
  - Modos de compilación:
    - **Modo objeto:** permite compilar código que usa objetos → **@jit**
    - **Modo nopython:** genera código que evita la API de CPython →**@njit**

```
from numba import njit

# Equivalente a indicar
# @jit(nopython=True)
@njit
def f(x, y):
    return x + y
```

Decorador →

Compilación en modo **nopython**

Optimización de N-Body usando Numba

- Numba: Numba
  - **Momento de la compilación:**
    - Por defecto, la función es compilada al momento de su invocación.
    - El parámetro **signature** permite que la función sea compilada en su declaración.
      - Además, especificar los tipos que usará la función y la organización de los datos en la memoria.

```python
from numba import njit, double

@njit(double(double[::1, :],
             double[:, ::1]))
def f(x, y):
    """
    x:  Vector 2D de tipo Double organizado por columnas.
    y:  Vector 2D de tipo Double organizado por filas.
    Retorna la suma del producto entre los vectores x e y.
    """
    return (x * y).sum()
```

Optimización de N-Body usando Numba

- Numba: Numba
  - **Multi-hilado**
    - Paralelización implícita → **parallel=true**
    - Paralelización explícita → **función prange**
  - **Vectorización (SIMD)**
    - Delega en LLVM la autovectorización y generación de instrucciones SIMD.
  - **Integración con NumPy**
    - Permite controlar la organización de memoria de los arreglos y realizar operaciones entre ellos.

```
from numba import njit, double, prange

@njit(double(double[::1]), parallel=True)
def f(x):
    """
    x: Vector 1D.
    Retorna la suma del vector x mediante
    una reducción.
    """
    N = x.shape[0]
    z = 0

    for i in prange(N):
        z += x[i]

    return z
```

Compilación en modo **nopython** con el parámetro **parallel=True**.

Optimización de N-Body usando Numba

- **Implementaciones:**
  - Enfoque incremental:
    - Como versión **naive** se tomó la versión que emplea *broadcasting*.

```python
def nbody(N, D, positions, masses, velocities, dp):
    for _ in range(D):
        for i in range(N):
            dpos = np.subtract(positions, positions[i])
            dsquared = (dpos ** 2.0).sum(axis=1) + SOFT
            gm = masses * (masses[i] * GRAVITY)
            d32 = dsquared ** -1.5
            gm_d32 = (gm * d32).reshape(N, 1)

            forces = np.multiply(gm_d32, dpos).sum(axis=0)

            acceleration = forces / masses[i]
            velocities[i] += acceleration * DT / 2.0
            dp[i] = velocities[i] * DT

        for i in range(N):
            positions[i] += dp[i]
```

    - Se exploraron diferentes optimizaciones de acuerdo a los resultados encontrados

Optimización de N-Body usando Numba

- Integración de Numba:

```
@njit(
    void(
        int64, int64,
        double[:, ::1], double[::1], double[:, ::1], double[:, ::1],
    ),
    fastmath=True,
    error_model="numpy",
)
def nbody(N, D, positions, masses, velocities, dp):
    for _ in range(D):
        for i in range(N):
            dpos = np.subtract(positions, positions[i])

            dsquared = (dpos ** 2.0).sum(axis=1) + SOFT
            gm = masses * (masses[i] * GRAVITY)
            d32 = dsquared ** -1.5
            gm_d32 = (gm * d32).reshape(N, 1)

            forces = np.multiply(gm_d32, dpos).sum(axis=0)

            acceleration = forces / masses[i]
            velocities[i] += acceleration * DT / 2.0

            dp[i] = velocities[i] * DT

        for i in range(N):
            positions[i] += dp[i]
```

Compilación modo *nopython*

Tipos de datos de los parámetros

Opciones de compilación

- **Notar:**
  - Uso de NumPy
  - No “invasión” en el código

Optimización de N-Body usando Numba

- **Multi-hilada:**
  - 2 funciones paralelas:

1° Función: Cálculo de posiciones y fuerzas	**2° Función:** Actualización de posiciones
(code 1 below)	(code 2 below)

```
@njit(
    void(
        int64, double[:, ::1], double[::1],
        double[:, ::1], double[:, ::1]
    ),
    fastmath=True,
    parallel=True,
    error_model="numpy",
)
def calculate_positions(N, positions, masses, velocities, dp):
    for i in prange(N):
        dpos = np.subtract(positions, positions[i])

        dsquared = (dpos ** 2.0).sum(axis=1) + SOFT
        gm = masses * (masses[i] * GRAVITY)
        d32 = dsquared ** -1.5
        gm_d32 = (gm * d32).reshape(N, 1)

        forces = np.multiply(gm_d32, dpos).sum(axis=0)

        acceleration = forces / masses[i]
        velocities[i] += acceleration * DT / 2.0

        dp[i] = velocities[i] * DT
```

```
@njit(
    void(int64, double[:, ::1], double[:, ::1]),
    fastmath=True,
    parallel=True,
    error_model="numpy",
)
def update_positions(N, positions, dp):
    for i in prange(N):
        positions[i] += dp[i]
```

- Se indican las zonas paralelas con **prange**
- **Barrera implícita** para cumplir con las dependencias de datos

Optimización de N-Body usando Numba

- **Arreglos con tipos de datos simples:**
  - Eliminación de *broadcasting* → Ayudar a Numba a autovectorizar (**SIMD**)

```
@njit(
    void(
        int64,
        double[::1], double[::1], double[::1],
        double[::1],
        double[::1], double[::1], double[::1],
        double[::1], double[::1], double[::1],
    ),
    fastmath=True, parallel=True, error_model="numpy",
)
def calculate_positions(
    N,
    positions_x, positions_y, positions_z,
    masses,
    velocities_x, velocities_y, velocities_z,
    dp_x, dp_y, dp_z,
):
    for i in prange(N):
        forces_x = forces_y = forces_z = 0.0

        for j in range(N):
            dpos_x = positions_x[j] - positions_x[i]
            dpos_y = positions_y[j] - positions_y[i]
            dpos_z = positions_z[j] - positions_z[i]

            dsquared = (dpos_x ** 2.0) + (dpos_y ** 2.0) + (dpos_z ** 2.0) + SOFT
            gm = GRAVITY * masses[j] * masses[i]
            d32 = dsquared ** -1.5

            forces_x += gm * d32 * dpos_x
            forces_y += gm * d32 * dpos_y
            forces_z += gm * d32 * dpos_z

        aceleration_x = forces_x / masses[i]
        aceleration_y = forces_y / masses[i]
        aceleration_z = forces_z / masses[i]

        velocities_x[i] += aceleration_x * DT / 2.0
        velocities_y[i] += aceleration_y * DT / 2.0
        velocities_z[i] += aceleration_z * DT / 2.0

        dp_x[i] = velocities_x[i] * DT
        dp_y[i] = velocities_y[i] * DT
        dp_z[i] = velocities_z[i] * DT
```

```
@njit(
    void(
        int64,
        double[::1], double[::1], double[::1],
        double[::1], double[::1], double[::1],
    ),
    fastmath=True,
    parallel=True,
    error_model="numpy",
)
def update_positions(
    N,
    positions_x, positions_y, positions_z,
    dp_x, dp_y, dp_z,
):
    for i in prange(N):
        positions_x[i] += dp_x[i]
        positions_y[i] += dp_y[i]
        positions_z[i] += dp_z[i]
```

Optimización de N-Body usando Numba

- Operaciones matemáticas
  - Funciones de potencias

```python
@njit(
    void(
        int64,
        double[::1], double[::1], double[::1],
        double[::1],
        double[::1], double[::1], double[::1],
        double[::1], double[::1], double[::1],
    ),
    fastmath=True, parallel=True, error_model="numpy",
)
def calculate_positions(
    N,
    positions_x, positions_y, positions_z,
    masses,
    velocities_x, velocities_y, velocities_z,
    dp_x, dp_y, dp_z,
):
    for i in prange(N):
        forces_x = forces_y = forces_z = 0.0

        for j in range(N):
            dpos_x = positions_x[j] - positions_x[i]
            dpos_y = positions_y[j] - positions_y[i]
            dpos_z = positions_z[j] - positions_z[i]

            dsquared = (POW(dpos_x, 2.0) + POW(dpos_y, 2.0) + POW(dpos_z, 2.0) + SOFT)
            gm = GRAVITY * masses[j] * masses[i]
            d32 = POW(dsquared, -1.5)

            forces_x += gm * d32 * dpos_x
            forces_y += gm * d32 * dpos_y
            forces_z += gm * d32 * dpos_z

        aceleration_x = forces_x / masses[i]
        aceleration_y = forces_y / masses[i]
        aceleration_z = forces_z / masses[i]

        velocities_x[i] += aceleration_x * DT / 2.0
        velocities_y[i] += aceleration_y * DT / 2.0
        velocities_z[i] += aceleration_z * DT / 2.0

        dp_x[i] = velocities_x[i] * DT
        dp_y[i] = velocities_y[i] * DT
        dp_z[i] = velocities_z[i] * DT
```

Variante 1

```python
POW = math.pow
```

Variante 2

```python
POW = numpy.power
```

Optimización de N-Body usando Numba

- Operaciones matemáticas
  - Cálculo del denominador de la ley de atracción universal de Newton

```
@njit(
    void(
        int64,
        double[::1], double[::1], double[::1],
        double[::1],
        double[::1], double[::1], double[::1],
        double[::1], double[::1], double[::1],
    ),
    fastmath=True, parallel=True, error_model="numpy",
)
def calculate_positions(
    N,
    positions_x, positions_y, positions_z,
    masses,
    velocities_x, velocities_y, velocities_z,
    dp_x, dp_y, dp_z,
):
    for i in prange(N):
        forces_x = forces_y = forces_z = 0.0

        for j in range(N):
            dpos_x = positions_x[j] - positions_x[i]
            dpos_y = positions_y[j] - positions_y[i]
            dpos_z = positions_z[j] - positions_z[i]

            dsquared = (POW(dpos_x, 2.0) + POW(dpos_y, 2.0) + POW(dpos_z, 2.0) + SOFT)
            gm = GRAVITY * masses[j] * masses[i]
            d32 = POW(dsquared, -1.5)

            forces_x += gm * d32 * dpos_x
            forces_y += gm * d32 * dpos_y
            forces_z += gm * d32 * dpos_z

        aceleration_x = forces_x / masses[i]
        aceleration_y = forces_y / masses[i]
        aceleration_z = forces_z / masses[i]

        velocities_x[i] += aceleration_x * DT / 2.0
        velocities_y[i] += aceleration_y * DT / 2.0
        velocities_z[i] += aceleration_z * DT / 2.0

        dp_x[i] = velocities_x[i] * DT
        dp_y[i] = velocities_y[i] * DT
        dp_z[i] = velocities_z[i] * DT
```

Variante 2

```
d32 = POW(dsquared, 1.5)
forces_x += (gm * dpos_x) / d32
forces_y += (gm * dpos_y) / d32
forces_z += (gm * dpos_z) / d32
```

Variante 1

```
d32 = 1.0 / POW(dsquared, 1.5)
forces_x += gm * d32 * dpos_x
forces_y += gm * d32 * dpos_y
forces_z += gm * d32 * dpos_z
```

Optimización de N-Body usando Numba

- **Vectorización (SIMD):**
  - Se habilitaron de forma explícita las instrucciones AVX-512
    - Archivo `.numba_config.yaml`

```yaml
ENABLE_AVX: 1
CPU_FEATURES: +avx512f,+avx512dq,+avx512cd,+avx512bw,+avx512vl
```

Optimización de N-Body usando Numba

➡ **Procesamiento de a bloques:**

```
def calculate_positions(
    N,
    positions_x, positions_y, positions_z,
    masses,
    velocities_x, velocities_y, velocities_z,
    dp_x, dp_y, dp_z,
):
    forces_x = np.zeros(N, dtype=DATATYPE)
    forces_y = np.zeros(N, dtype=DATATYPE)
    forces_z = np.zeros(N, dtype=DATATYPE)

    for b in prange(BLOCKS):
        first = b * BLOCKSIZE
        last = first + BLOCKSIZE
        for j in range(N):
            for i in range(first, last):

                dpos_x = positions_x[j] - positions_x[i]
                dpos_y = positions_y[j] - positions_y[i]
                dpos_z = positions_z[j] - positions_z[i]

                dsquared = (
                    (dpos_x ** 2.0) + (dpos_y ** 2.0) + (dpos_z ** 2.0) + SOFT
                )

                gm = GRAVITY * masses[j] * masses[i]
                d32 = dsquared ** -1.5

                forces_x[i] += gm * dpos_x * d32
                forces_y[i] += gm * dpos_y * d32
                forces_z[i] += gm * dpos_z * d32
        for i in range(first, last):
            aceleration_x = forces_x[i] / masses[i]
            aceleration_y = forces_y[i] / masses[i]
            aceleration_z = forces_z[i] / masses[i]

            velocities_x[i] += aceleration_x * DT / 2.0
            velocities_y[i] += aceleration_y * DT / 2.0
            velocities_z[i] += aceleration_z * DT / 2.0

            dp_x[i] = velocities_x[i] * DT
            dp_y[i] = velocities_y[i] * DT
            dp_z[i] = velocities_z[i] * DT
```

➡ Objetivo: Minimizar el tráfico a memoria principal utilizando la caché de forma más adecuada.

➡ Se separó el bucle **i** en dos particiones:

1. La primera partición se coloca del *bucle j* → Calcula la **fuerza de atracción gravitacional de Newton.**
2. La segunda partición computa la **integración de Verlet**.

Optimización de N-Body usando Numba

- **Threading layer**
  - API de hilos que traducen las regiones paralelas (`prange`).
  - Se probaron:
    - default → Elige la mejor opción en base al sistema de soporte.
    - workqueue → *built-in* de Numba, independiente del SO.
    - omp → OpenMP, solo para Linux.
  - Archivo `.numba_config.yaml`

```yaml
THREADING_LAYER: "workqueue"

THREADING_LAYER: "omp"

THREADING_LAYER: "default"
```

Optimización de N-Body usando Numba

- Threading layer
  - Además, se probó una distribución de hilos manual
    - Se utilizó `threading` y se desactivó el GIL a través de Numba.

```python
def nbody(
    N, D, chunk,
    positions_x, positions_y, positions_z,
    masses,
    velocities_x, velocities_y, velocities_z,
    dp_x, dp_y, dp_z,
    barrier,
):
    for _ in range(D):
        calculate_positions(
            N, chunk,
            positions_x, positions_y, positions_z,
            masses,
            velocities_x, velocities_y, velocities_z,
            dp_x, dp_y, dp_z,
        )

        barrier.wait()

        update_positions(
            chunk,
            positions_x, positions_y, positions_z,
            dp_x, dp_y, dp_z,
        )

        barrier.wait()
```

```python
@njit(
    void(
        int64, int64[::1],
        float32[::1], float32[::1], float32[::1],
        float32[::1],
        float32[::1], float32[::1], float32[::1],
        float32[::1], float32[::1], float32[::1],
    ),
    nogil=True, fastmath=True, error_model="numpy",
)
def calculate_positions(
    N, chunk,
    positions_x, positions_y, positions_z,
    masses,
    velocities_x, velocities_y, velocities_z,
    dp_x, dp_y, dp_z,
):
    for i in chunk:
        ...
```

```python
@njit(
    void(
        int64[::1],
        float32[::1], float32[::1], float32[::1],
        float32[::1], float32[::1], float32[::1],
    ),
    nogil=True,
    fastmath=True,
)
def update_positions(
    chunk,
    positions_x, positions_y, positions_z,
    dp_x, dp_y, dp_z,
):

    for i in chunk:
        positions_x[i] += dp_x[i]
        positions_y[i] += dp_y[i]
        positions_z[i] += dp_z[i]
```

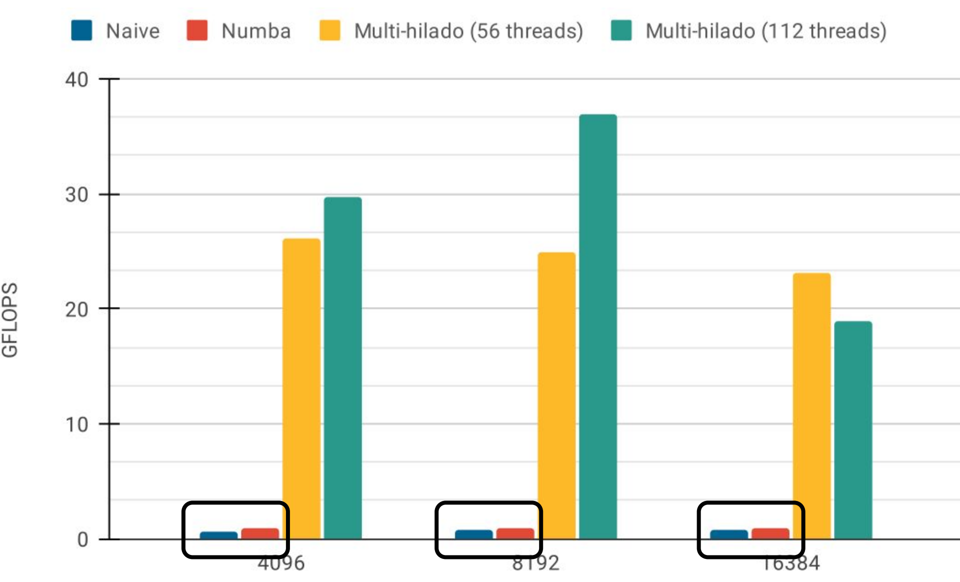
- Barreras explícitas
- Desactivación del GIL
- División manual de la carga de trabajo

Optimización de N-Body usando Numba

- **Resultados experimentales**
  - **Plataforma**
    - Servidor Intel con 2× Xeon Platinum 8276 de 28 núcleos (2 hilos hw por núcleo) y 256 GB de memoria RAM.
    - El sistema operativo fue Ubuntu 20.04.2 LTS
    - Python v3.8.10 **-** NumPy v1.20.1 **- Numba v0.52.0**.
  - **Parámetros**
    - **N** = {4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288}
    - **I** = {100}
    - **T** = {1, 28, 56, 112}
  - **Métrica de rendimiento:**
    - **GFLOPS** → $GFLOPS = \frac{20 \times N^2 \times I}{t \times 10^9}$
  - Cada optimización propuesta, fue aplicada y evaluada incrementalmente a partir de la versión *naive*.

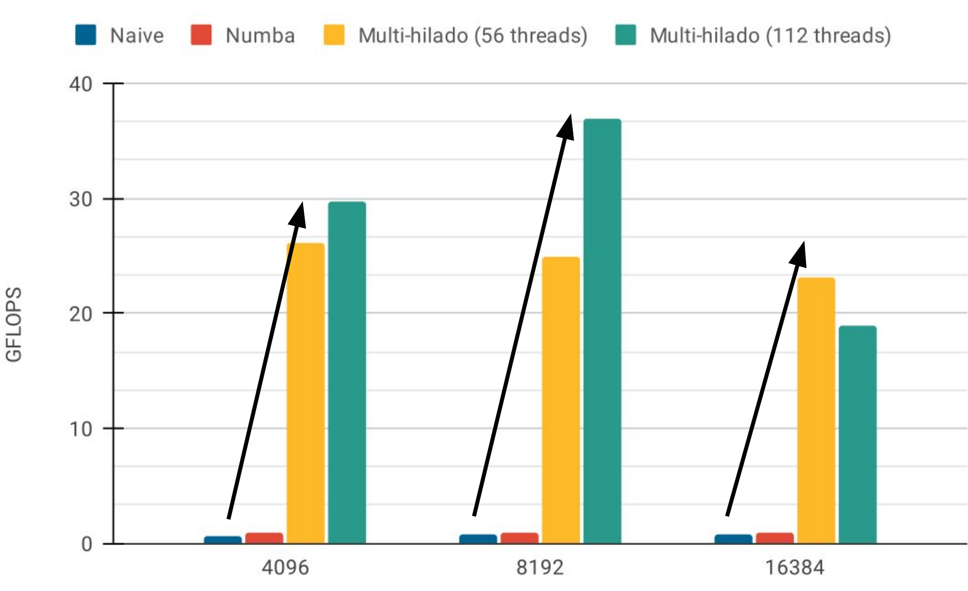
Optimización de N-Body usando Numba

- **Rendimientos obtenidos para opciones de compilación y multi-hilado al variar N.**
  - La simple integración de Numba no tuvo incidencia



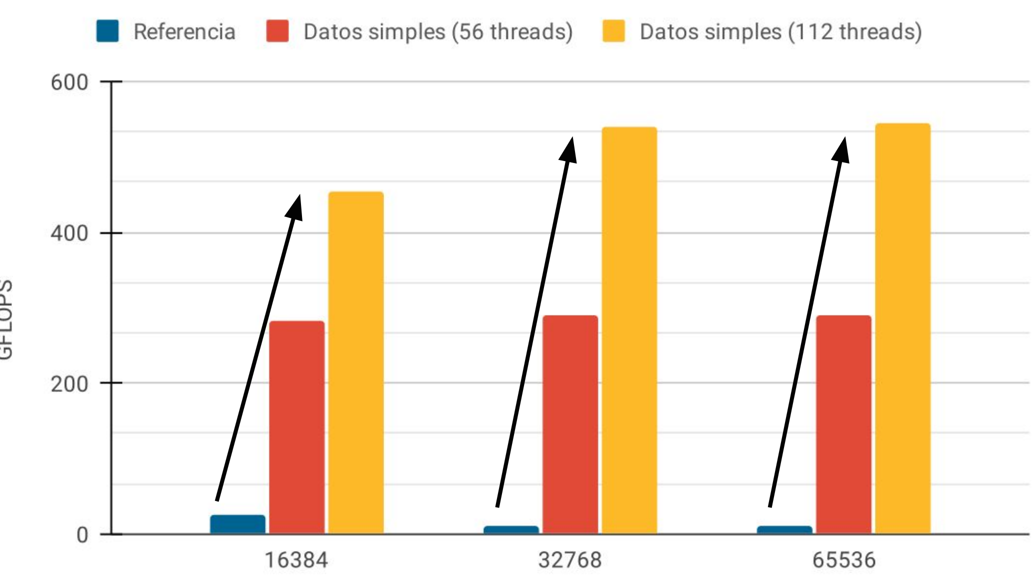
Optimización de N-Body usando Numba

- **Rendimientos obtenidos para opciones de compilación y multi-hilado al variar N.**
    - Mejora en promedio de 33× y 38× para 56 y 112 hilos, respectivamente



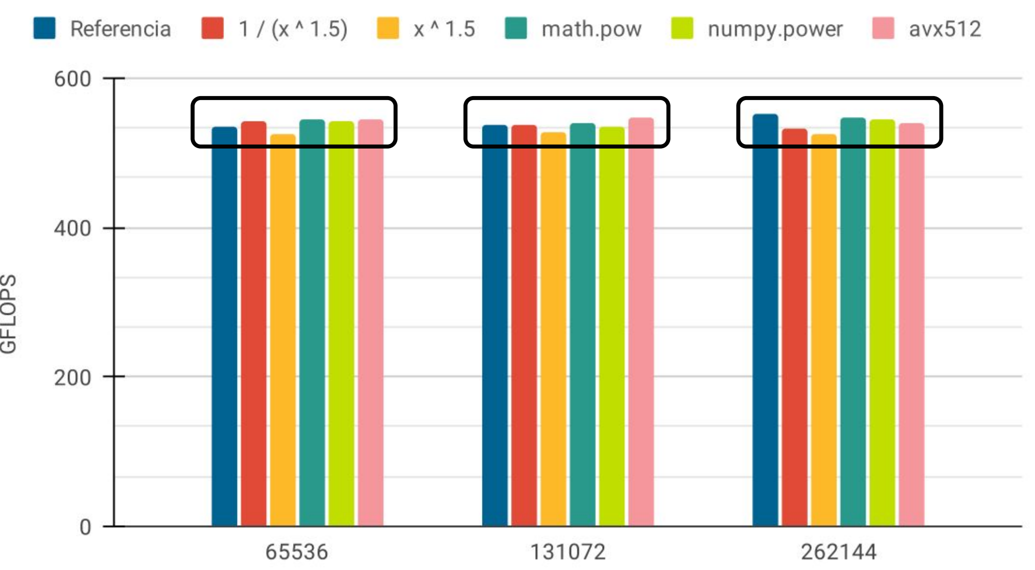
Optimización de N-Body usando Numba

- ➦ **Rendimientos obtenidos al usar tipos de datos simples variando N**
  - ➥ Mejora en promedio de ~41× para 112 hilos



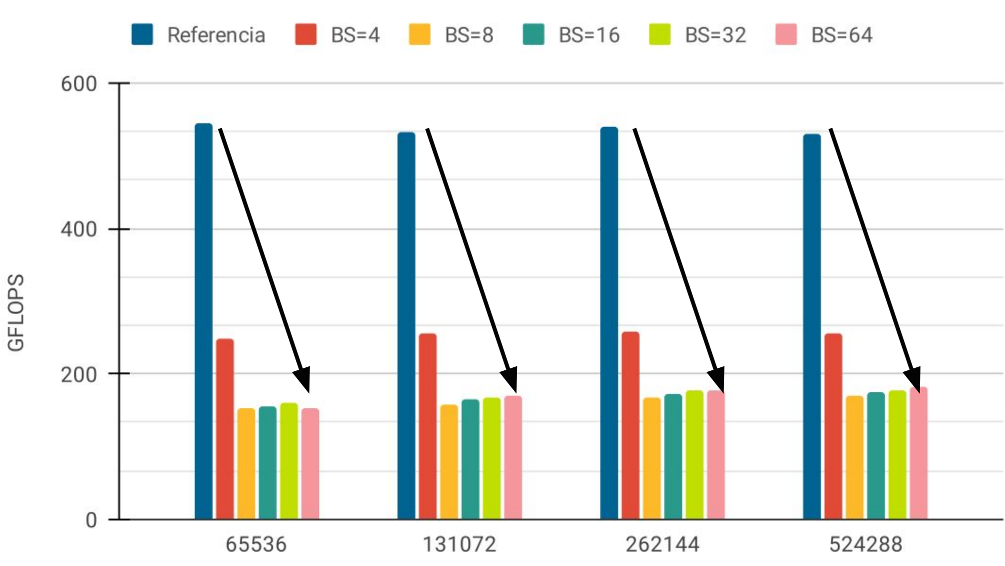
Optimización de N-Body usando Numba

- **Rendimientos obtenidos utilizando el uso de diferentes cálculos matemáticos, funciones de potencia e instrucciones AVX512 al variar N**
  - No hubo mejoras significativas → Código de máquina similar



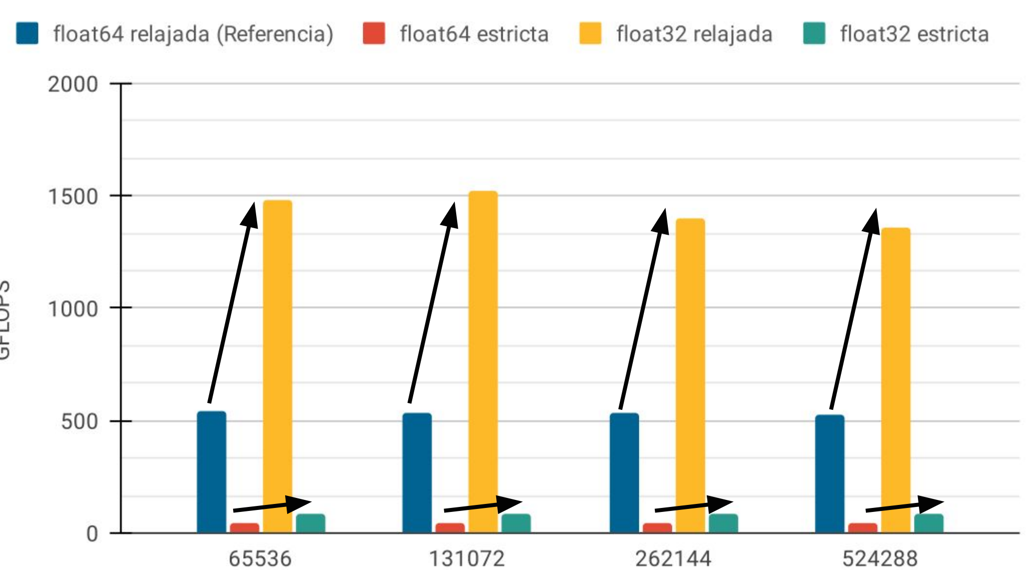
Optimización de N-Body usando Numba

- **Rendimiento obtenido del procesamiento de a bloques al variar N.**
  - Empeoró el rendimiento →Fallos en LLVM a la hora de autovectorizar (SIMD)



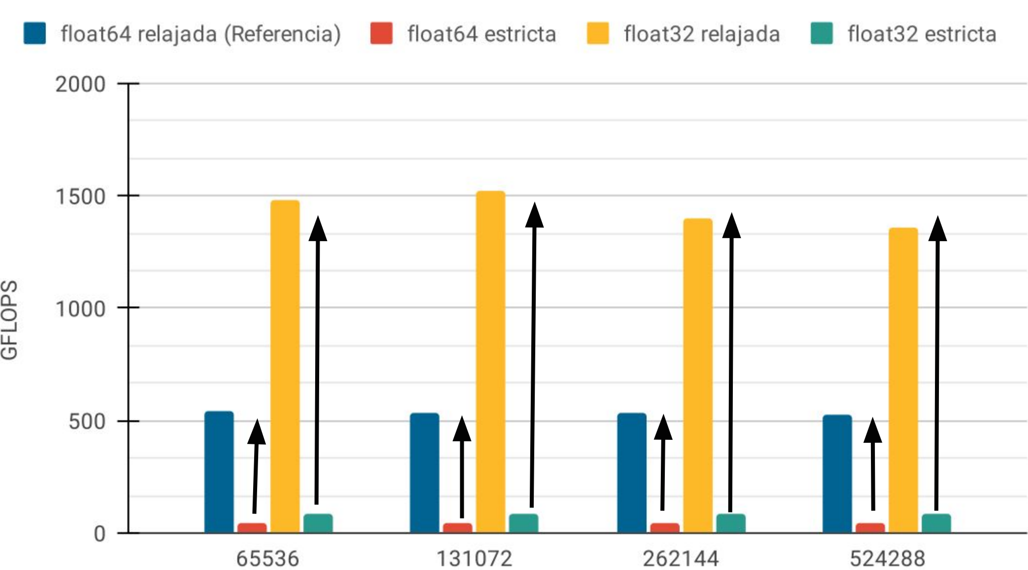
Optimización de N-Body usando Numba

- **Rendimiento obtenido para la relajación de precisión al variar el tipo de dato y N.**
    - Mejora hasta 2.8× por reducción de precisión



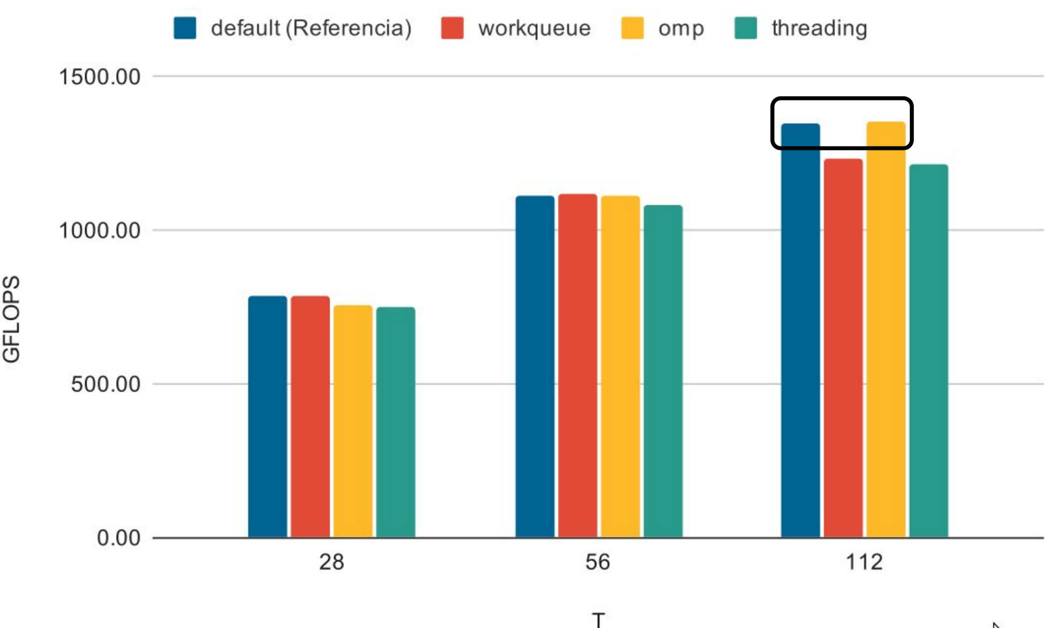
Optimización de N-Body usando Numba

- **Rendimiento obtenido para la relajación de precisión al variar el tipo de dato y N.**
  - Mejora ~17× para *float32* y ~11× para *float64* por relajación de precisión



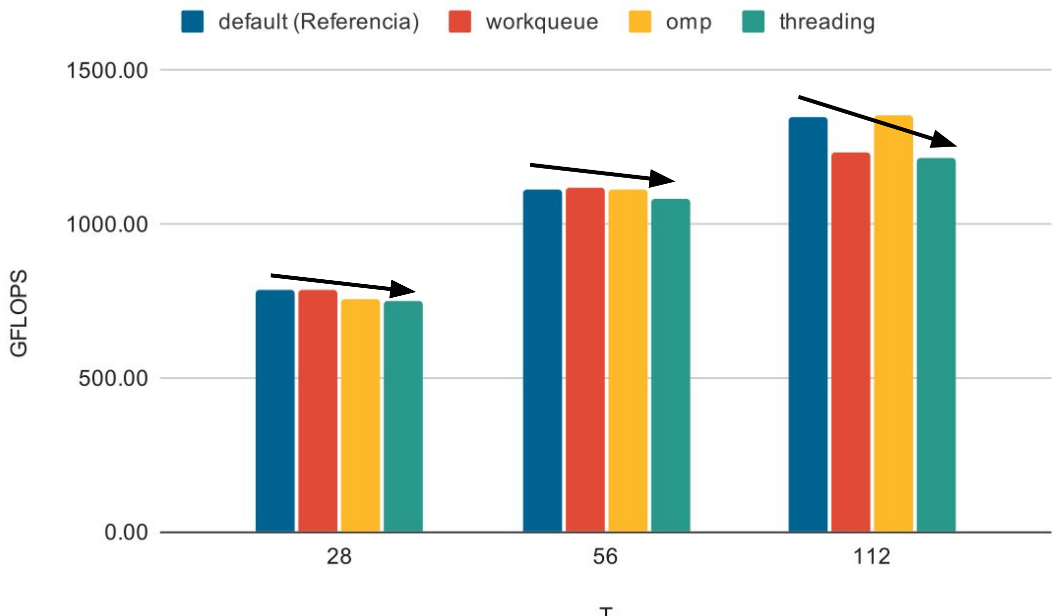
Optimización de N-Body usando Numba

- **Rendimiento obtenido con las diferentes threading layers de Numba al variar T y fijar N = 524288.**
  - En T=112 → OpenMP y *default*, superan por un promedio de ~9% a las demás.



Optimización de N-Body usando Numba

- **Rendimiento obtenido con las diferentes threading layers de Numba al variar T y fijar N = 524288.**
  - `threading` presentó el rendimiento más bajo → *overhead* por objetos de Python para sincronizar los hilos.

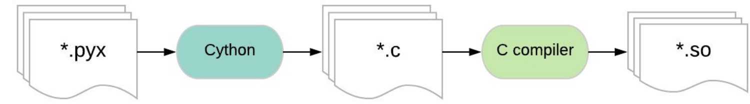


6

Optimización de N-Body usando Cython

Optimización de N-Body usando Cython

- **Cython**
  - **Compilador** para Python
    - Permite interactuar con funciones de C (tipos de datos, malloc, etc).
    - Permite utilizar librerías de C → **OpenMP** → **Multi-hilado**.
  - Permite desactivar el GIL.
  - **Compilación**:
    - Código fuente → **Archivo .pyx**:
      - Sintaxis específica de Cython.
    - Mediante un archivo **setup.py** indicando los flags de compilación.
    - Como salida se tendrá:
      - Archivo **.C** → Código transpilado por Cython.
      - Archivo **.so** → Importar en scripts de Python.



Flujo de programación en Cython

Optimización de N-Body usando Cython

- Cython
  - Tipos de datos
    - *Opcionales*, pero recomendados.
    - Se transpilan a tipos de datos de C.
    - Evita la inferencia de tipos de CPython.
  - Multi-hilado
    - Posibilidad de utilizar OpenMP
    - `prange` → `parallel for` de OpenMP
    - `nogil` → Desactivar GIL
    - Reducciones a través de op. *in-situ*
    - *scheduling* → Static por *default*.

```
cdef int x, y, z
cdef float a, b[100], *c

cdef struct Point:
    double x
    double y
```

Variables declaradas con tipos de datos de C en Cython.

```
from cython.parallel import prange

cdef int i
cdef int N = 30
cdef int total = 0

for i in prange(N, nogil=True):
    total += i
```

Reducción utilizando el bloque **prange** de Cython.

Optimización de N-Body usando Cython

- **Implementaciones**:
  - Enfoque incremental:
    - Como versión **naive** se tomó la versión que **no emplea** *broadcasting* → Debido a que no se puede utilizar dichas operaciones con Cython.
    - Se exploraron diferentes optimizaciones de acuerdo a los resultados encontrados

Optimización de N-Body usando Cython

- Integración de Cython:
  - $.py \rightarrow .pyx$

```
def nbody(
    N, D,
    positions_x, positions_y, positions_z,
    masses,
    velocities_x, velocities_y, velocities_z,
    dp_x, dp_y, dp_z,
):
    for _ in range(D):
        for i in range(N):
            forces_x = forces_y = forces_z = 0.0

            for j in range(N):
                dpos_x = positions_x[j] - positions_x[i]
                dpos_y = positions_y[j] - positions_y[i]
                dpos_z = positions_z[j] - positions_z[i]

                dsquared = (
                    dpos_x ** 2.0 + dpos_y ** 2.0 + dpos_z ** 2.0 + SOFT
                )

                gm = GRAVITY * masses[j] * masses[i]

                d32 = dsquared ** -1.5

                forces_x += gm * d32 * dpos_x
                forces_y += gm * d32 * dpos_y
                forces_z += gm * d32 * dpos_z

            aceleration_x = forces_x / masses[i]
            aceleration_y = forces_y / masses[i]
            aceleration_z = forces_z / masses[i]

            velocities_x[i] += aceleration_x * DT / 2.0
            velocities_y[i] += aceleration_y * DT / 2.0
            velocities_z[i] += aceleration_z * DT / 2.0

            dp_x[i] = velocities_x[i] * DT
            dp_y[i] = velocities_y[i] * DT
            dp_z[i] = velocities_z[i] * DT

        for i in range(N):
            positions_x[i] += dp_x[i]
            positions_y[i] += dp_y[i]
            positions_z[i] += dp_z[i]
```

Optimización de N-Body usando Cython

➡ **Tipado explícito:**

➥ Verificaciones innecesarias en tiempo de ejecución

➥ Tipado de Cython

```
@boundscheck(False)
@wraparound(False)
@nonecheck(False)
@cdivision(True)
cpdef void nbody(
    int N, int D,
    double[::1] positions_x, double[::1] positions_y, double[::1] positions_z,
    double[::1] masses,
    double[::1] velocities_x, double[::1] velocities_y, double[::1] velocities_z,
    double[::1] dp_x, double[::1] dp_y, double[::1] dp_z,
):
    cdef double forces_x, forces_y, forces_z
    cdef double aceleration_x, aceleration_y, aceleration_z
    cdef double dpos_x, dpos_y, dpos_z
    cdef double dsquared, gm, d32
    cdef int i, j

    for _ in range(D):
        for i in range(N):
            forces_x = forces_y = forces_z = 0.0

            for j in range(N):
                dpos_x = positions_x[j] - positions_x[i]
                dpos_y = positions_y[j] - positions_y[i]
                dpos_z = positions_z[j] - positions_z[i]

                dsquared = (dpos_x ** 2.0) + (dpos_y ** 2.0) + (dpos_z ** 2.0) + SOFT
                gm = GRAVITY * masses[j] * masses[i]
                d32 = dsquared ** -1.5

                forces_x += gm * d32 * dpos_x
                forces_y += gm * d32 * dpos_y
                forces_z += gm * d32 * dpos_z

            aceleration_x = forces_x / masses[i]
            aceleration_y = forces_y / masses[i]
            aceleration_z = forces_z / masses[i]

            velocities_x[i] += aceleration_x * DT / 2.0
            velocities_y[i] += aceleration_y * DT / 2.0
            velocities_z[i] += aceleration_z * DT / 2.0

            dp_x[i] = velocities_x[i] * DT
            dp_y[i] = velocities_y[i] * DT
            dp_z[i] = velocities_z[i] * DT

        for i in range(N):
            positions_x[i] += dp_x[i]
            positions_y[i] += dp_y[i]
            positions_z[i] += dp_z[i]
```

Optimización de N-Body usando Cython

- Multi-hilado:

```
@boundscheck(False)
@wraparound(False)
@nonecheck(False)
@cdivision(True)
cpdef void nbody(
    int N, int D,
    double[::1] positions_x, double[::1] positions_y, double[::1] positions_z,
    double[::1] masses,
    double[::1] velocities_x, double[::1] velocities_y, double[::1] velocities_z,
    double[::1] dp_x, double[::1] dp_y, double[::1] dp_z,
):
    cdef double forces_x, forces_y, forces_z
    cdef double aceleration_x, aceleration_y, aceleration_z
    cdef double dpos_x, dpos_y, dpos_z
    cdef double dsquared, gm, d32
    cdef int i, j

    for _ in range(D):

        for i in prange(N, nogil=True, schedule="static", num_threads=T):
            forces_x = forces_y = forces_z = 0.0

            for j in range(N):

                dpos_x = positions_x[j] - positions_x[i]
                dpos_y = positions_y[j] - positions_y[i]
                dpos_z = positions_z[j] - positions_z[i]

                dsquared = (dpos_x ** 2.0) + (dpos_y ** 2.0) + (dpos_z ** 2.0) + SOFT
                gm = GRAVITY * masses[j] * masses[i]
                d32 = dsquared ** -1.5

                forces_x = forces_x + gm * d32 * dpos_x
                forces_y = forces_y + gm * d32 * dpos_y
                forces_z = forces_z + gm * d32 * dpos_z

            aceleration_x = forces_x / masses[i]
            aceleration_y = forces_y / masses[i]
            aceleration_z = forces_z / masses[i]

            velocities_x[i] += aceleration_x * DT / 2.0
            velocities_y[i] += aceleration_y * DT / 2.0
            velocities_z[i] += aceleration_z * DT / 2.0

            dp_x[i] = velocities_x[i] * DT
            dp_y[i] = velocities_y[i] * DT
            dp_z[i] = velocities_z[i] * DT

        for i in prange(N, nogil=True, schedule="static", num_threads=T):
            positions_x[i] += dp_x[i]
            positions_y[i] += dp_y[i]
            positions_z[i] += dp_z[i]
```

Optimización de N-Body usando Cython

- **Operaciones matemáticas:**
  - Cálculo del denominador de la ley de atracción universal de Newton

```
...
for _ in range(D):

    for i in prange(N, nogil=True, schedule="static", num_threads=T):
        forces_x = forces_y = forces_z = 0.0

        for j in range(N):

            dpos_x = positions_x[j] - positions_x[i]
            dpos_y = positions_y[j] - positions_y[i]
            dpos_z = positions_z[j] - positions_z[i]

            dsquared = (dpos_x ** 2.0) + (dpos_y ** 2.0) + (dpos_z ** 2.0) + SOFT
            gm = GRAVITY * masses[j] * masses[i]

            d32 = dsquared ** -1.5

            forces_x = forces_x + gm * d32 * dpos_x
            forces_y = forces_y + gm * d32 * dpos_y
            forces_z = forces_z + gm * d32 * dpos_z

        aceleration_x = forces_x / masses[i]
        aceleration_y = forces_y / masses[i]
        aceleration_z = forces_z / masses[i]

        velocities_x[i] += aceleration_x * DT / 2.0
        velocities_y[i] += aceleration_y * DT / 2.0
        velocities_z[i] += aceleration_z * DT / 2.0

        dp_x[i] = velocities_x[i] * DT
        dp_y[i] = velocities_y[i] * DT
        dp_z[i] = velocities_z[i] * DT

    for i in prange(N, nogil=True, schedule="static", num_threads=T):
        positions_x[i] += dp_x[i]
        positions_y[i] += dp_y[i]
        positions_z[i] += dp_z[i]
```

Variante 2

```
d32 = POW(dsquared, 1.5)
forces_x += (gm * dpos_x) / d32
forces_y += (gm * dpos_y) / d32
forces_z += (gm * dpos_z) / d32
```

Variante 1

```
d32 = 1.0 / POW(dsquared, 1.5)
forces_x += gm * d32 * dpos_x
forces_y += gm * d32 * dpos_y
forces_z += gm * d32 * dpos_z
```

Optimización de N-Body usando Cython

- Procesamiento de a bloques

```
...
    for _ in range(D):

        for b_i in prange(0, N, BLOCKSIZE, nogil=True, schedule="static", num_threads=T):
            for i in range(b_i, b_i + BLOCKSIZE):
                forces_x[i] = 0.0
                forces_y[i] = 0.0
                forces_z[i] = 0.0

            for j in range(N):
                for i in range(b_i, b_i + BLOCKSIZE):
                    dpos_x = positions_x[j] - positions_x[i]
                    dpos_y = positions_y[j] - positions_y[i]
                    dpos_z = positions_z[j] - positions_z[i]

                    dsquared = (dpos_x ** 2.0) + (dpos_y ** 2.0) + (dpos_z ** 2.0) + SOFT
                    gm = GRAVITY * masses[j] * masses[i]
                    d32 = dsquared ** -1.5

                    forces_x[i] += gm * d32 * dpos_x
                    forces_y[i] += gm * d32 * dpos_y
                    forces_z[i] += gm * d32 * dpos_z

            for i in range(b_i, b_i + BLOCKSIZE):
                aceleration_x = forces_x[i] / masses[i]
                aceleration_y = forces_y[i] / masses[i]
                aceleration_z = forces_z[i] / masses[i]

                velocities_x[i] += aceleration_x * DT / 2.0
                velocities_y[i] += aceleration_y * DT / 2.0
                velocities_z[i] += aceleration_z * DT / 2.0

                dp_x[i] = velocities_x[i] * DT
                dp_y[i] = velocities_y[i] * DT
                dp_z[i] = velocities_z[i] * DT

        for i in prange(N, nogil=True, schedule="static", num_threads=T):
            ...
```

- <u>Objetivo:</u> Minimizar el tráfico a memoria principal utilizando la caché de forma más adecuada.

- Se separó el bucle **i** en dos particiones:

1. La primera partición se coloca del *bucle j* → Calcula la **fuerza de atracción gravitacional de Newton.**

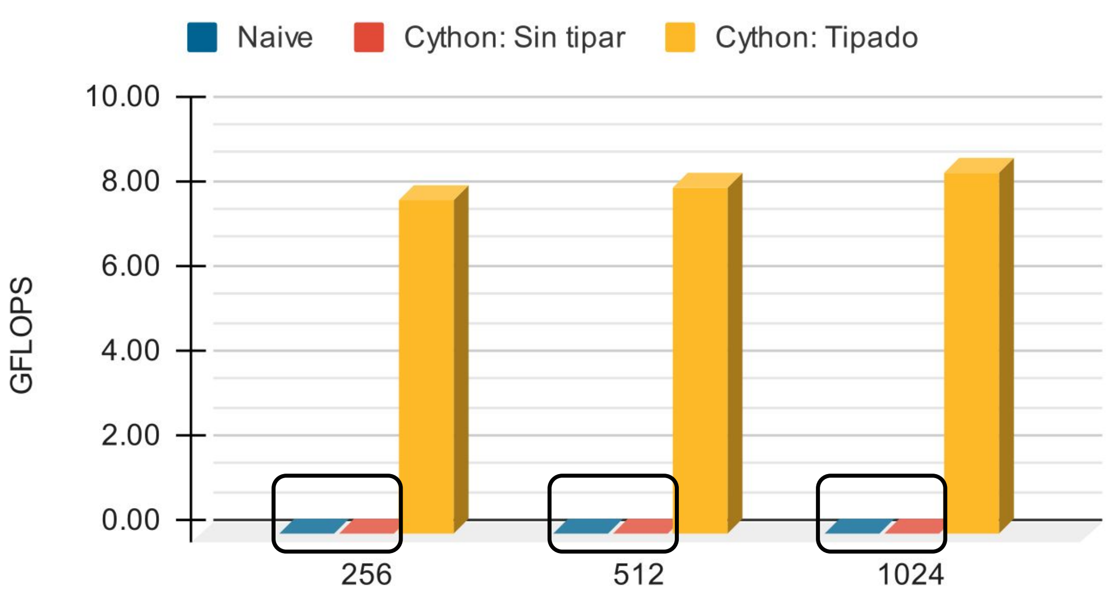
2. La segunda partición computa la **integración de Verlet**.

Optimización de N-Body usando Cython

- **Resultados experimentales**
  - **Plataforma**
    - Servidor Intel con 2× Xeon Platinum 8276 de 28 núcleos (2 hilos hw por núcleo) y 256 GB de memoria RAM.
    - El sistema operativo fue Ubuntu 20.04.2 LTS
    - Python v3.8.10 **-** NumPy v1.20.1 **- Cython v0.29.22** con **ICC v19.1.0.166**.
  - **Parámetros**
    - **N** = {256, 512, 1024, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288}
    - **I** = {100}
    - **T** = {1, 112}
  - **Métrica de rendimiento:**
    - **GFLOPS** → $GFLOPS = \frac{20 \times N^2 \times I}{t \times 10^9}$
  - Cada optimización propuesta, fue aplicada y evaluada incrementalmente a partir de la versión *naive*.
  - Todos compilados con el flag *-O3*.

Optimización de N-Body usando Cython

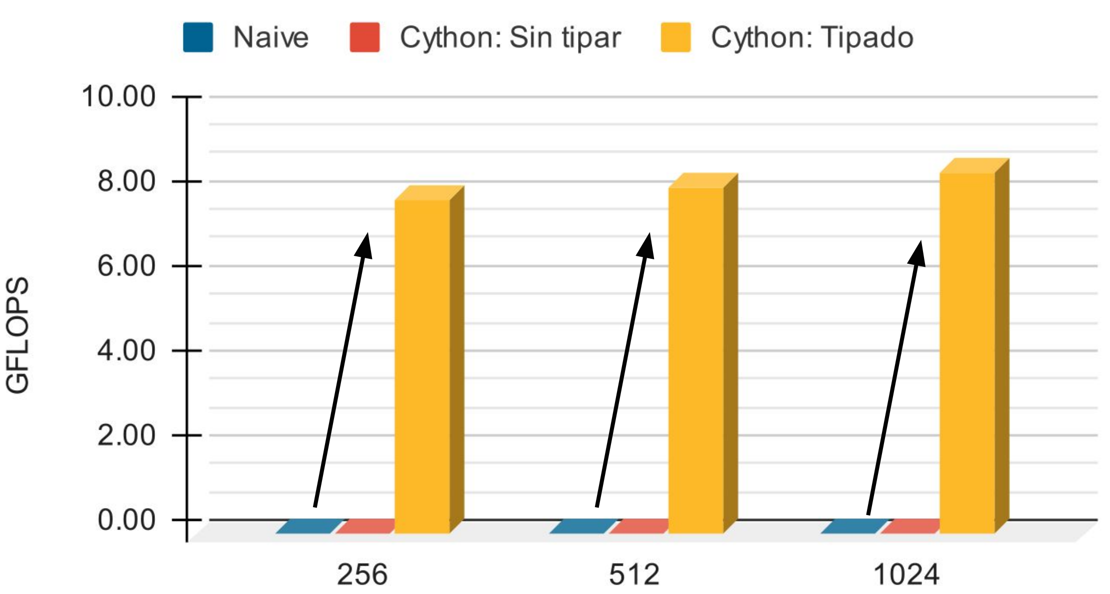
- **Rendimientos obtenidos de la integración de Cython con y sin tipado explícito al variar N.**
  - La simple integración de Cython no tuvo incidencias.



Optimización de N-Body usando Cython

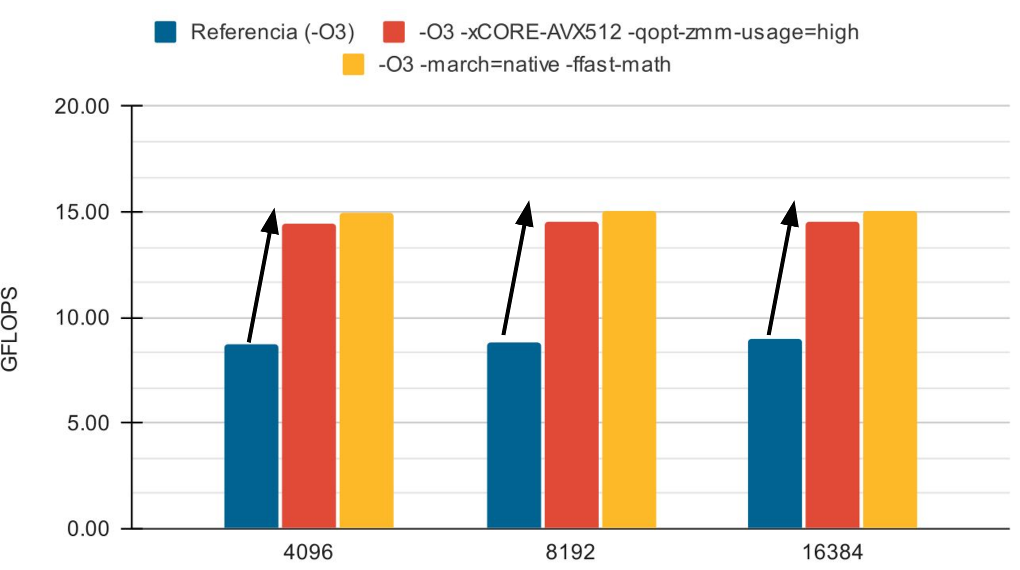
➦ **Rendimientos obtenidos de la integración de Cython con y sin tipado explícito al variar N.**

➥ Cython tipado → mejora de 547.7× en promedio → Casi no hay interacción con CPython.



Optimización de N-Body usando Cython

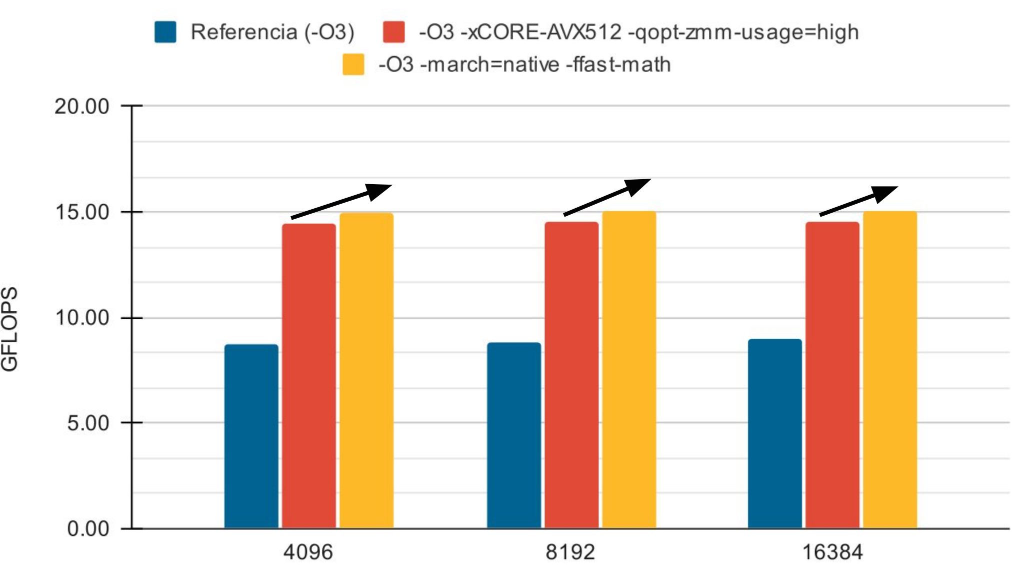
- **Rendimientos obtenidos para las opciones de compilación de Cython al variar N.**
  - AVX-512 mejora 1.7× en promedio.



Optimización de N-Body usando Cython

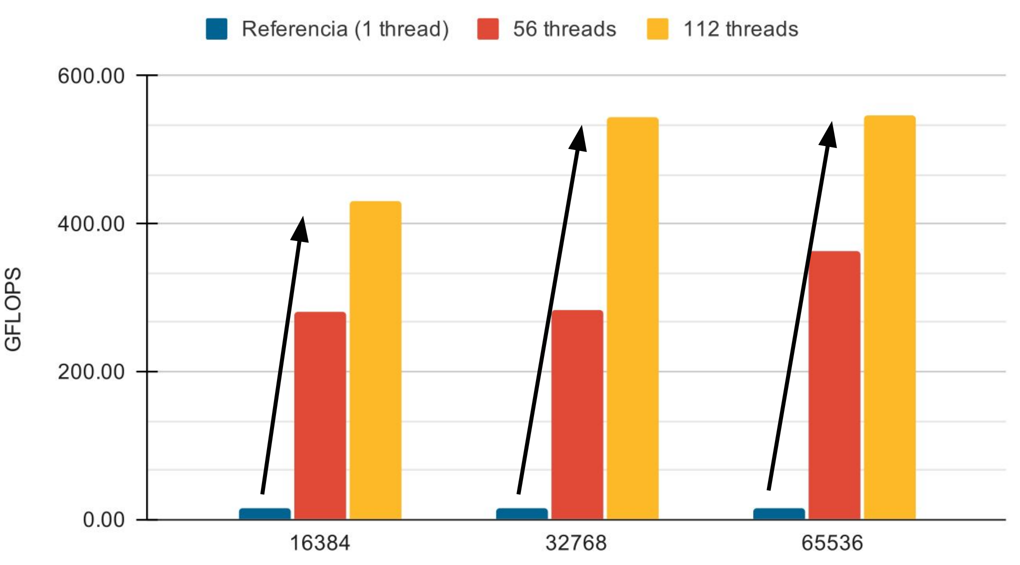
➦ **Rendimientos obtenidos para las opciones de compilación de Cython al variar N.**

➥ `-march=native`~1.4 GFLOPS por encima de `-xCORE-AVX512 -qopt-zmm-usage=high`



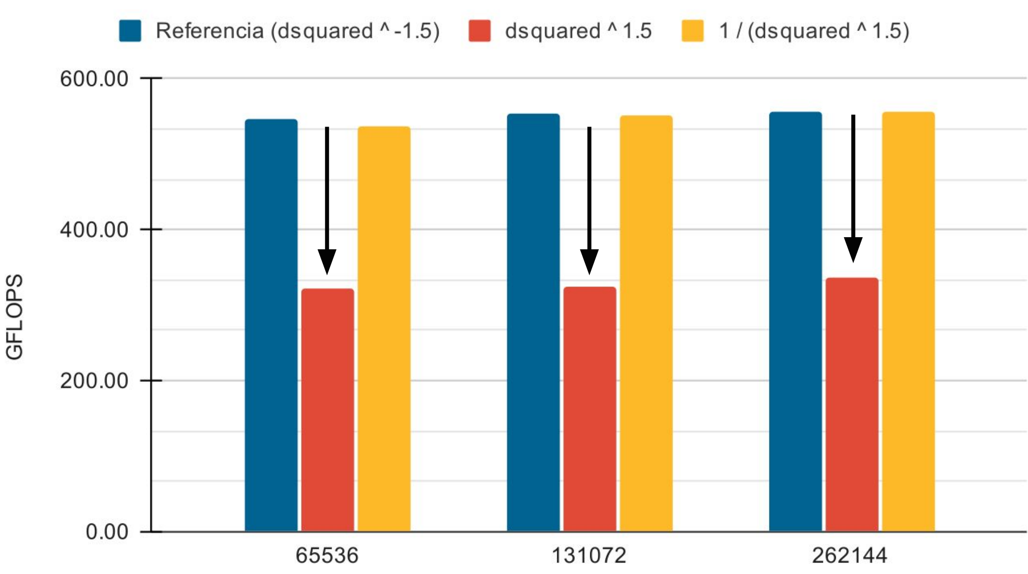
Optimización de N-Body usando Cython

- **Rendimientos obtenidos de la solución multi-hilada con Cython al variar N.**
    - Mejora en promedio de 21.1× con 56 hilos y 34.6× con 112 hilos.



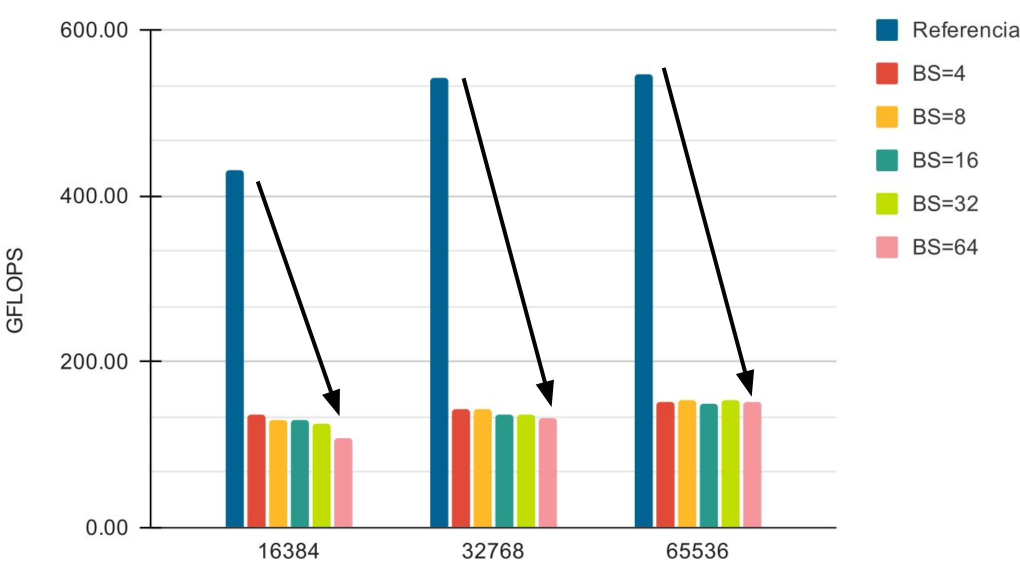
Optimización de N-Body usando Cython

- **Rendimientos obtenidos para los cálculos matemáticos con Cython al variar N.**
    - División directa degradó el rendimiento un 41%.



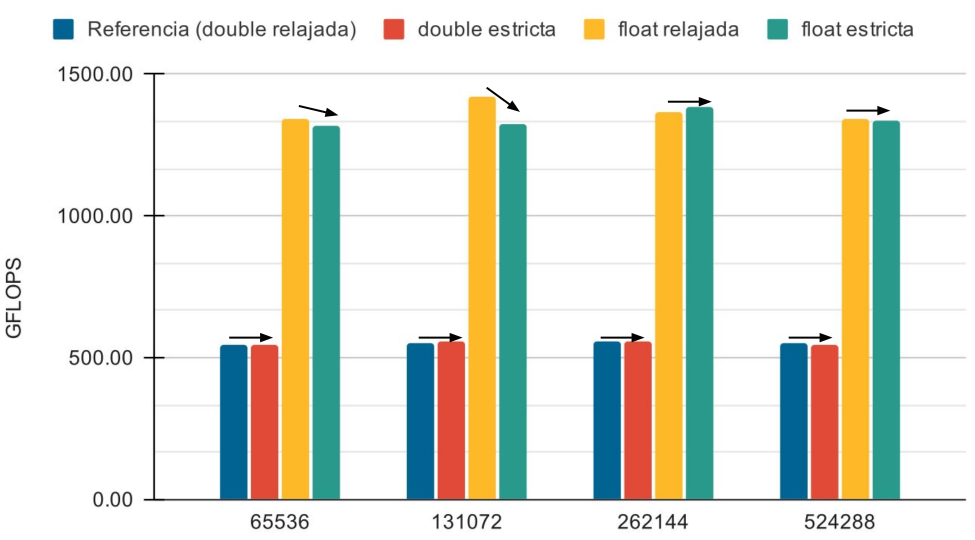
Optimización de N-Body usando Cython

- **Rendimientos obtenidos de procesamiento por bloques utilizando Cython y variando N.**
    - Empeoró notablemente el rendimiento → el compilador identifica falsas dependencias para generar instrucciones **SIMD**



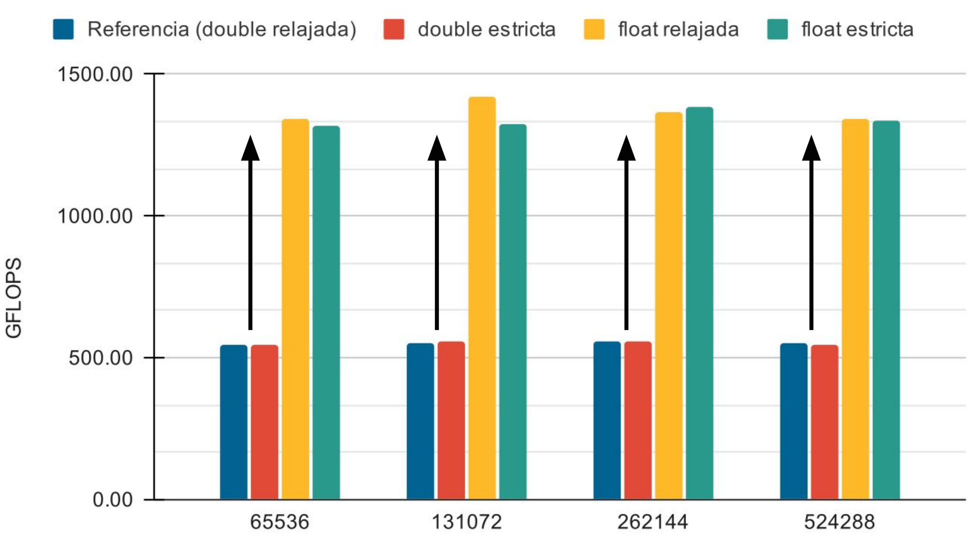
Optimización de N-Body usando Cython

- **Rendimientos obtenidos para la relajación de precisión al variar el tipo de dato y N con Cython.**
    - La relajación de precisión prácticamente no tuvo incidencias.



Optimización de N-Body usando Cython

- **Rendimientos obtenidos para la relajación de precisión al variar el tipo de dato y N con Cython.**
  - *float* mejora 1362 GFLOPS en promedio.



Comparación de prestaciones

Comparación de prestaciones

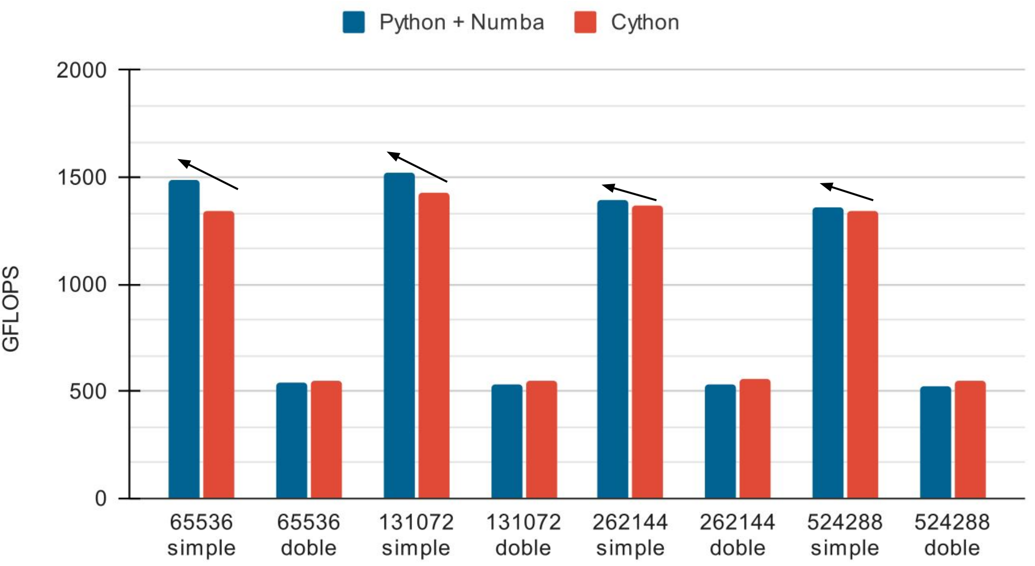
- **Rendimiento**
  - **Plataforma**
    - Servidor Intel con 2× Xeon Platinum 8276 de 28 núcleos (2 hilos hw por núcleo) y 256 GB de memoria RAM.
    - El sistema operativo fue Ubuntu 20.04.2 LTS
    - Python v3.8.10 **-** NumPy v1.20.1 **-** Numba v0.52.0 - Cython v0.29.22 - ICC v19.1.0.166.
  - **Parámetros**
    - **N** = {65536, 131072, 262144, 524288}
    - **I** = {100}
    - **T** = {1, 112}
  - **Métrica de rendimiento:**
    - **GFLOPS** → $GFLOPS = \frac{20 \times N^2 \times I}{t \times 10^9}$

Comparación de prestaciones

- **Rendimiento**
  - No se realizaron experimentos adicionales.
  - Se utilizaron las versiones más rápidas en términos de rendimiento.
  - No se utilizaron las versiones ejecutadas con CPython y PyPy debido a su bajo rendimiento
    - 0.5 GFLOPS en promedio.

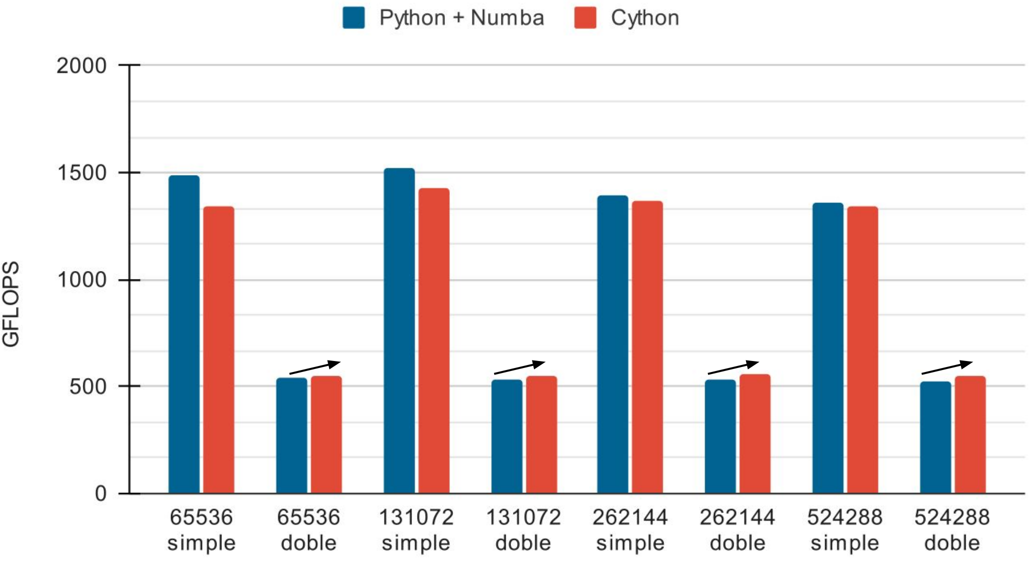
Comparación de prestaciones

- **Comparación de rendimiento de las versiones finales entre Numba y Cython variando el tipo de dato y N.**
  - Simple precisión → Numba mejoró 3% (en promedio) el rendimiento de Cython.



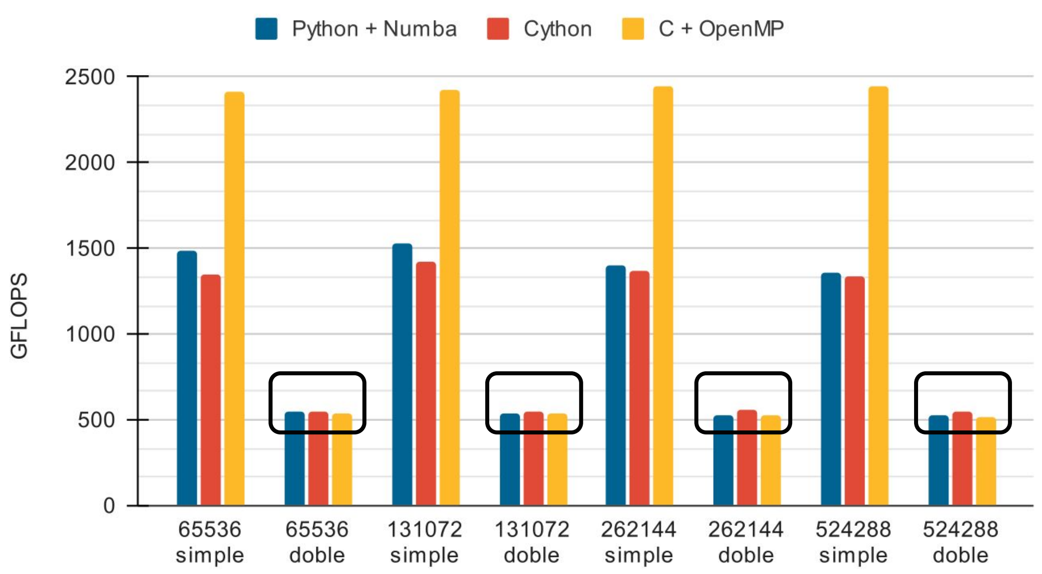
Comparación de prestaciones

- **Comparación de rendimiento de las versiones finales entre Numba y Cython variando el tipo de dato y N.**
  - Doble precisión → Cython mejoró 6% (en promedio) el rendimiento de Numba.



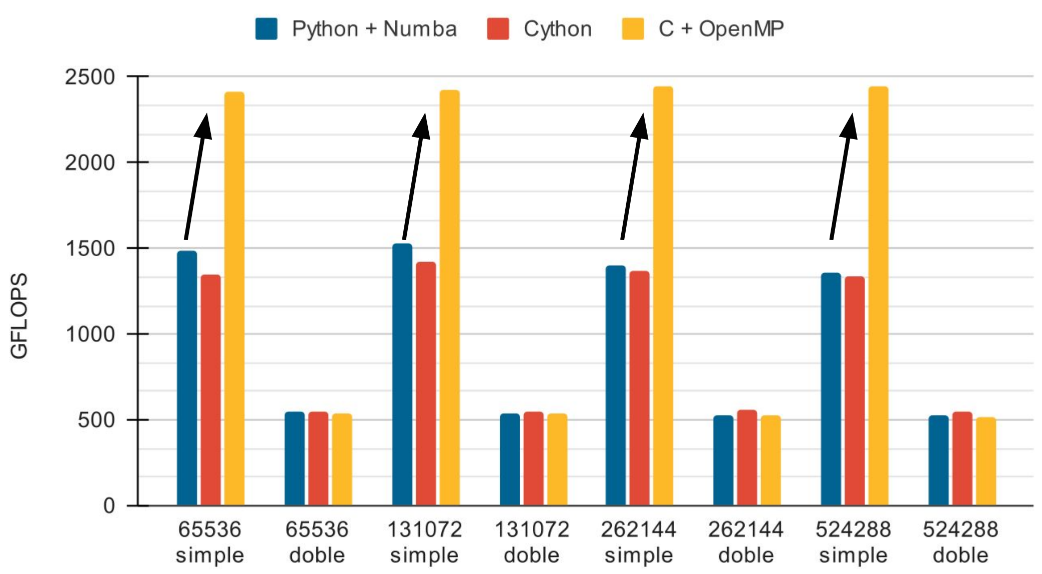
Comparación de prestaciones

- **Adicional: Comparación de rendimiento de las versiones finales entre Python+Numba, C+OpenMP y Cython variando el tipo de dato y N.**
    - Doble precisión → No hay diferencias significativas.



Comparación de prestaciones

- **Adicional:** **Comparación de rendimiento de las versiones finales entre Python+Numba, C+OpenMP y Cython variando el tipo de dato y N.**
  - Simple precisión → C+OpenMP superior 1.7x que Numba y 1.8x que Cython.

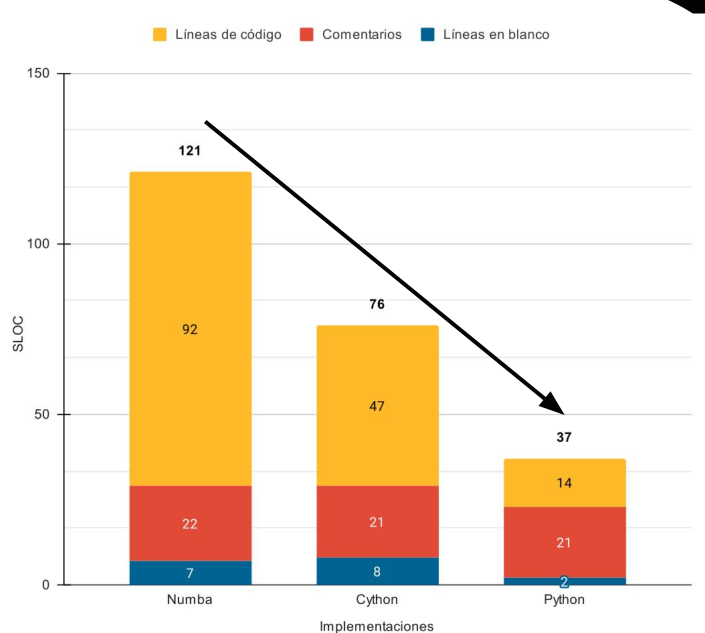


Comparación de prestaciones

- **Esfuerzo de programación**
  - No se realizaron experimentos adicionales.
  - Se utilizaron las versiones más rápidas en términos de rendimiento.
  - **Métrica:**
    - **Análisis cuantitativo** a través del indicador SLOC → subjetividad.
    - **Análisis cualitativo** para complementar el análisis anterior.
  - Se utilizó la herramienta *cloc* para calcular el indicador SLOC.

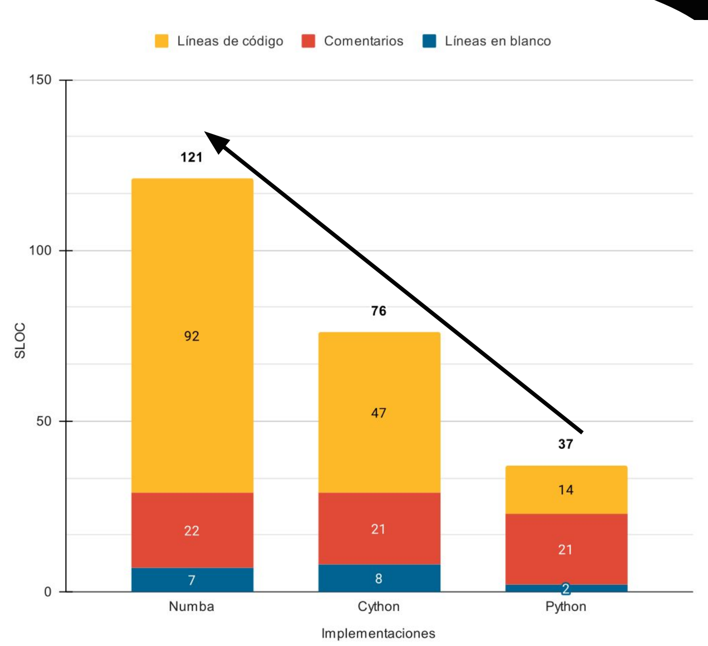
Comparación de prestaciones

- **Cantidad de líneas de código de las optimizaciones finales.**
  - **Python**
    - Representa ~29% de Cython y ~15% de Numba
    - En parte gracias al *broadcasting*
      - Llevó un alto esfuerzo de programación que no se ve representado por el SLOC.



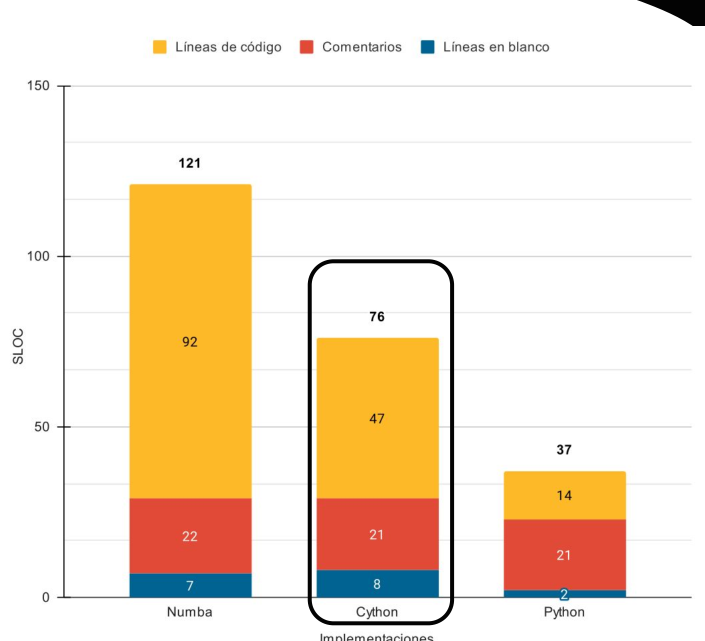
Comparación de prestaciones

- **Cantidad de líneas de código de las optimizaciones finales.**
  - **Numba**
    - Requirió 6.6x más que Python y 2x más que Cython.
    - 32.6% del código de Numba son decoradores.
    - 2 funciones adicionales para paralelizar.
    - Independencia de la API de hilos subyacente.
      - *threading layers*.
    - En ningún momento se tuvo que modificar el código fuente → *decoradores*.



Comparación de prestaciones

- **Cantidad de líneas de código de las optimizaciones finales.**
  - Cython
    - Requirió conocimiento adicional de C+OpenMP.
    - Menos SLOCS que Numba.
    - Posibilidad de utilizar las funciones de OpenMP para sincronizar.
    - Se tuvo que modificar el código para agregar tipos de datos.
    - Necesitó 3 archivos:
      - cynbody.pyx
      - run.py
      - setup.py



Conclusiones y trabajos futuros

Conclusiones

- Rendimiento
  - **PyPy y CPython**
    - No mejoraron notablemente el rendimiento debido a su incapacidad de paralelizar debido al GIL.
  - **Numba y Cython**
    - No hubo diferencias significativas.
    - Ambos traductores mejoraron el rendimiento de CPython → rendimientos similares a OpenMP.

Conclusiones

- **Esfuerzo de programación**
  - **Cython**
    - Requirió menos líneas de código.
    - Conocimiento adicional de C+OpenMP.
    - Proceso más laborioso de ejecución.
  - **Numba**
    - Requirió más líneas de código que el resto.
    - Más simple de desarrollar → Menos modificaciones en el código existente.

Conclusiones

- En base al trabajo realizado, se concluye que **Numba y Cython**:
  - Pueden ser herramientas potentes para acelerar aplicaciones *CPU-bound* desarrolladas en Python.
  - La elección entre uno y otro estará mayormente determinada por el enfoque que el equipo de desarrollo encuentre más conveniente.

Contribuciones

- **Implementaciones** optimizadas y escritas en diferentes traductores de Python que computan N-Body sobre arquitecturas multicore.
  - Disponible en: https://github.com/Pastorsin/python-hpc-study
- Una **comparación rigurosa** de las soluciones para N-Body en arquitecturas multicore considerando rendimiento y esfuerzo de programación.
  - Útil para programadores Python y para la evaluación del potencial de Python para HPC.
- **Publicaciones y presentaciones orales:**
  - CACIC 2021 - Acelerando código científico en Python usando Numba
    - Disponible en: http://sedici.unlp.edu.ar/handle/10915/126012
  - Springer CACIC 2021 - Performance Comparison of Python Translators for a Multi-threaded CPU-bound Application.
    - *En prensa*.
  - PyConAr 2021 - Acelerando aplicaciones paralelas en Python: Numba vs. Cython
    - Disponible en: https://eventos.python.org.ar/events/pyconar2021/activity/448/

Trabajos futuros

- Explorar otras capacidades y limitaciones de los traductores de Python → utilización de GPUs.
- Replicar este estudio, considerando:
  - Otros casos de estudio que sean computacionalmente intensivos pero cuyas características sean diferentes a las de N-Body.
  - Otras arquitecturas multicore distintas a la usada en este trabajo.
- Dado que existen otras tecnologías que permitan implementar paralelismo **a nivel de procesos** en Python → realizar una comparación entre ellas considerando no sólo el rendimiento sino también el costo de programación.

¡Muchas gracias!

¿Preguntas?

CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon**, and infographics & images by **Freepik**.