Acelerando Código Científico en Python usando Numba

Andrés Milla[1] and Enzo Rucci[2]

[1] Facultad de Informática, UNLP. La Plata (1900), Bs As, Argentina
andressmilla@gmail.com
[2] III-LIDI, Facultad de Informática, UNLP – CIC. La Plata (1900), Bs As, Argentina
erucci@lidi.info.unlp.edu.ar

Resumen En la actualidad, Python es uno de los lenguajes más utilizados en diversas áreas de aplicación. Una de ellas es el ámbito científico, donde resulta habitual la existencia de algoritmos numéricos que requieren un gran costo computacional. Sin embargo, Python presenta limitaciones a la hora de poder paralelizar esta clase de código. Para solucionar esta problemática surge Numba, un compilador JIT que traduce Python en código de máquina optimizado a través de LLVM. Esta herramienta cuenta con primitivas para paralelizar algoritmos, autovectorización mediante instrucciones SIMD, entre otras características. En este estudio, se analizan algunas capacidades y limitaciones de Numba para acelerar algoritmos numéricos, utilizando como caso de estudio *N-Body*, un problema popular en simulación y con alta demanda computacional. Partiendo desde una implementación base desarrollada en Python con NumPy, se muestra como la integración de diferentes opciones de Numba la mejoran hasta 687×, presentando rendimientos cercanos a una implementación de C+OpenMP en una arquitectura multicore Intel de 56 núcleos.

Palabras claves: Python · Numba · N-body · HPC · Multi-hilado

1. Introducción

Desde su surgimiento a comienzos de la década del 90, Python se ha convertido en uno de los lenguajes más populares en la actualidad. De acuerdo con el índice TIOBE, Python se ubica segundo entre los lenguajes más populares del 2021 [17].

Python es un lenguaje de programación de alto nivel, interpretado, interactivo, dinámico y multi-paradigma [3]. Su notable poder de programación se debe a su sintaxis limpia y clara, la cual provoca que el esfuerzo de programación sea menor comparado con otros lenguajes [7,6]. Entre las áreas de aplicación, se pueden mencionar desarrollo web, educación, aplicaciones de escritorio, desarrollo de videojuegos, inteligencia artificial, *web scraping*, procesamiento de imágenes, entre otras [8,2].

La computación científica es otra área donde Python es muy usado, en parte debido a la existencia de un diverso ecosistema formado por herramientas y

librerías tanto de propósito general como específico [16]. Aun así, Python es considerado "lento" en comparación a lenguajes compilados como C, C++ y Fortran, especialmente para aplicaciones de cómputo numérico. Entre las causas de su pobre rendimiento, se encuentran su naturaleza de lenguaje interpretado y sus limitaciones al momento de implementar soluciones multi-hiladas [13]. En particular, el principal problema es la utilización de un componente llamado *Global Interpreter Lock* (GIL), el cual permite que único un hilo se ejecute a la vez. Es por lo que la comunidad de Python ha desarrollado varias propuestas que buscan superar esta limitación y así mejorar el rendimiento.

Entre estas soluciones, se encuentra Numba, un compilador *Just-In-Time* (JIT) que traduce Python en código de máquina optimizado [4]. Numba utiliza decoradores [1] para intervenir lo menos posible en el código del programador y, de acuerdo con su documentación, es capaz de acercarse a los rendimientos de C, C++ y Fortran.

En este artículo se propone evaluar y verificar las prestaciones de Numba en el ámbito de la computación científica. Como caso de estudio, se selecciona la simulación de N cuerpos computacionales (*N-Body*), un problema *cpu-bound* que resulta popular en la comunidad científica. Mediante este estudio se espera contribuir con la comunidad Python al explorar algunas de las capacidades y limitaciones de Numba para implementar aplicaciones paralelas sobre arquitecturas CPU multicore.

El resto del artículo se organiza de la siguiente forma. La Sección 2 introduce a Numba mientras que la Sección 3 describe al problema N-Body. Luego, la Sección 4 detalla las implementaciones realizadas. A continuación, la Sección 5 analiza los resultados experimentales mientras que la Sección 6 discute trabajos relacionados. Finalmente, la Sección 7 resume las conclusiones junto al trabajo futuro.

2. Numba

Numba es un compilador JIT que permite traducir código Python a código de máquina optimizado a través de LLVM [3]. De acuerdo con su documentación, es capaz de alcanzar aceleraciones similares a las de lenguajes compilados como C, C++ y Fortran [4], sin necesidad de re-escribir su código gracias a un enfoque de anotaciones llamados decoradores [1].

Compilación JIT. La librería ofrece dos modos de compilación: (1) modo objeto, el cual permite compilar código que haga uso de objetos; (2) modo *nopython*, que le permite a Numba generar código evitando a la API de

```
1  from numba import njit
2
3  # Equivalente a indicar
4  # @jit(nopython=True)
5  @njit
6  def f(x, y):
7      return x + y
```

Figura 1: Compilación en modo *nopython*.

[3] The LLVM Compiler Infrastructure, `https://llvm.org/`

```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()
```

Figura 2: Compilación en modo *nopython* con el parámetro `signature`

```python
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
```

Figura 3: Compilación en modo *nopython* con el parámetro `parallel`

CPython. Para indicar dichos modos, se utilizan los decoradores `@jit` y `@njit` (ver Fig. 1), respectivamente [4].

Por defecto, cada función será compilada al momento de ser invocada y se mantendrá en la caché para futuras llamadas. Sin embargo, la inclusión del parámetro `signature` provocará que la función sea compilada al momento de la declaración. Además, también posibilitará indicar los tipos de datos que usará la función y controlar la organización de los datos [4] en memoria (ver Fig. 2).

Multi-hilado. Numba permite activar un sistema de paralelización automática estableciendo el parámetro `parallel=True`, como también indicar una paralelización explícita mediante la función `prange` (ver Fig. 3), la cual distribuye las iteraciones entre los hilos de manera similar a la directiva `parallel for` de OpenMP. Además, también soporta reducciones y se encarga de declarar las variables como privadas a cada hilo si son declaradas dentro del alcance de la zona paralela. Lamentablemente, Numba aún no soporta primitivas que permitan controlar la sincronización de los hilos, como pueden ser semáforos o *locks* [4].

Vectorización. Numba delega en LLVM la autovectorización del código y la generación de instrucciones SIMD, pero le permite al programador controlar ciertos parámetros que podrían influir en esta tarea, como la precisión numérica mediante el argumento `fastmath=True`. También ofrece la posibilidad de utilizar *Intel SVML* en caso de estar disponible en el sistema [4].

Integración con NumPy. Cabe destacar que Numba soporta un gran número de funciones de NumPy, lo cual le permite al programador controlar la organización de memoria de los arreglos y realizar operaciones entre ellos [5,4].

Soporte para GPUs. Además de CPUs, Numba es capaz de aprovechar las capacidades de las GPUs, tanto de NVIDIA como de AMD.

3. N-Body

El problema consiste en simular la evolución de un sistema compuesto por N cuerpos durante una cantidad de tiempo determinada. Dados la masa y el

estado inicial (velocidad y posición) de cada cuerpo, se simula el movimiento del sistema a través de instantes discretos de tiempo. En cada uno de ellos, todo cuerpo experimenta una aceleración que surge de la atracción gravitacional del resto, lo que afecta a su estado.

La física subyacente es fundamentalmente la mecánica de Newton [18]. La simulación se realiza en 3 dimensiones espaciales y la atracción gravitacional entre dos cuerpos C_1 y C_2 se computa usando la ley de gravitación universal de Newton (ver Ecuación 1), donde F corresponde a la magnitud de la fuerza gravitacional entre los cuerpos; G corresponde a la constante de gravitación universal [4]; m_1 y m_2 corresponden a las masas de los cuerpos C_1 y C_2, respectivamente; y r corresponde a la distancia Euclídea [5] entre los cuerpos C_1 y C_2.

Cuando N es mayor a 2, la fuerza de gravitación sobre un cuerpo, se obtiene con la sumatoria de todas las fuerzas de gravitación ejercidas por los $N-1$ cuerpos restantes. La fuerza de atracción se traduce entonces en una aceleración del cuerpo mediante la aplicación de la segunda ley de Newton, la cual está dada por la Ecuación 2, donde F es el vector fuerza, calculado utilizando la magnitud obtenida con la ecuación de gravitación y la dirección y sentido del vector que va desde el cuerpo afectado hacia el cuerpo que ejerce la atracción.

La aceleración de un cuerpo se puede calcular a partir de la Ecuación 2, dividiendo la fuerza total por su masa. Durante un pequeño intervalo de tiempo dt, la aceleración a_i del cuerpo C_i es aproximadamente constante, por lo que el cambio en velocidad está dado aproximadamente por la Ecuación 3.

El cambio en la posición de un cuerpo es la integral de su velocidad y aceleración sobre el intervalo de tiempo dt, el cual se aproxima a la Ecuación 4. Esta fórmula emplea el esquema de integración Leapfrog [19], en el cual una mitad del cambio de posición emplea la velocidad *vieja* mientras que la otra considera la velocidad *nueva*.

$$F = \frac{G \times m_1 \times m_2}{r^2} \qquad (1) \qquad\qquad F = m \times a \qquad (2)$$

$$dv_i = a_i dt \qquad (3) \qquad\qquad dp_i = v_i dt + \frac{a_i}{2} dt^2 = (v_i + \frac{dv_i}{2}) dt \qquad (4)$$

4. Propuesta

En esta sección se describen las diferentes implementaciones propuestas.

4.1. Implementación Naive

Inicialmente se desarrolló una implementación Python "pura" (denominada *naive*), la cual servirá como referencia para evaluar las mejoras introducidas por el uso de Numba. Esta implementación utiliza las operaciones entre vectores que provee NumPy [5] y su código es el de las líneas 6-25 de la Fig. 4.

[4] Equivalente a $6{,}674 \times 10^{11}$

[5] Se calcula utilizando la fórmula $\sqrt{((x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2)}$, siendo ($x_1$, y_1, z_1) las coordenadas de $C1$ y (x_2, y_2, z_2) las coordenadas de C_2.

```
1  @njit(
2      void(int32, int32, double[:, ::1], double[::1], double[:, ::1]),
3      fastmath=True,
4      error_model="numpy"
5  )
6  def nbody(N, D, positions, masses, velocities):
7      # Para cada instante discreto de tiempo
8      for _ in range(D):
9          # Para todo cuerpo que experimenta una fuerza
10         for i in range(N):
11             # Distancias de los cuerpos hacia el cuerpo i
12             dpos = positions - positions[i]
13             # Magnitudes de las distancias
14             dsquared = (dpos ** 2.0).sum(axis=1) + SOFT
15             # Factores de masa
16             gm = masses * (masses[i] * GRAVITY)
17             # Ley de atracción gravitacional de Newton
18             d32 = dsquared ** -1.5
19             gm_d32 = (gm * d32).reshape(N, 1)
20             forces = (gm_d32 * dpos).sum(axis=0)
21             # Integración de Verlet
22             acceleration = forces / masses[i]
23             velocities[i] += acceleration * DT / 2.0
24             # Actualización de las posiciones del cuerpo i
25             positions[i] += velocities[i] * DT
```

Figura 4: Código de la implementación *Naive* con la integración de Numba.

4.2. Integración de Numba

La primera versión Numba se obtuvo al incluir un decorador en la implementación *naive* (ver líneas 1-5 de la Fig. 4). Se indicó que el código se compile con precisión relajada mediante el parámetro `fastmath` (línea 3), con el modelo de división de NumPy para evitar la verificación de división por cero (línea 4) [4] y con *Intel SVML*, el cuál es inferido por Numba por estar disponible en el sistema.

4.3. Multihilado

Para introducir paralelismo a nivel de hilos, se utilizó la sentencia `prange`. Para ello, fue necesario antes separar el bucle que itera sobre los cuerpos (línea 10 de la Fig. 4) en dos. El primer bucle se encarga de computar la ley de atracción gravitacional de Newton y la integración de Verlet, mientras que el otro actualiza la posición de los cuerpos.

4.4. Arreglos con tipos de datos simples

Se reemplazaron las operaciones vectoriales de NumPy por operaciones numéricas, y las estructuras bidimensionales fueron sustituidas por unidimensionales para ayudar a Numba a la hora de autovectorizar el código (ver Fig. 5).

4.5. Operaciones matemáticas

Se propone evaluar alternativas para el cálculo del denominador de la ley de atracción universal de Newton por: (1) calcular la potencia positiva y luego

```
# Para cada instante discreto de tiempo
for _ in range(D):
    # Para todo cuerpo que experimenta una fuerza
    for i in prange(N):
        # Inicialización del vector de fuerza
        forces_x = 0.0
        forces_y = 0.0
        forces_z = 0.0
        # Para todo cuerpo que ejerce una fuerza
        for j in range(N):
            # Distancia hacia el cuerpo i
            dpos_x = positions_x[j] - positions_x[i]
            dpos_y = positions_y[j] - positions_y[i]
            dpos_z = positions_z[j] - positions_z[i]
            # Magnitud de la distancia
            dsquared = ((dpos_x ** 2.0) + (dpos_y ** 2.0) + (dpos_z ** 2.0) + SOFT)
            # Factor de masa
            gm = GRAVITY * masses[j] * masses[i]
            # Ley de atracción gravitacional de Newton
            d32 = dsquared ** -1.5
            forces_x += gm * d32 * dpos_x
            forces_y += gm * d32 * dpos_y
            forces_z += gm * d32 * dpos_z
        # Integración de Verlet: aceleración
        aceleration_x = forces_x / masses[i]
        aceleration_y = forces_y / masses[i]
        aceleration_z = forces_z / masses[i]
        # Integración de Verlet: velocidad
        velocities_x[i] += aceleration_x * DT / 2.0
        velocities_y[i] += aceleration_y * DT / 2.0
        velocities_z[i] += aceleration_z * DT / 2.0
        # Integración de Verlet: posición
        dp_x[i] = velocities_x[i] * DT
        dp_y[i] = velocities_y[i] * DT
        dp_z[i] = velocities_z[i] * DT
    # Actualización de la posición de los cuerpos
    for i in prange(N):
        positions_x[i] += dp_x[i]
        positions_y[i] += dp_y[i]
        positions_z[i] += dp_z[i]
```

Figura 5: Código de la implementación paralela sin operaciones de NumPy.

dividir; y (2) multiplicar por el inverso multiplicativo, calculando la potencia positiva previamente. Adicionalmente, se ponen a prueba las siguientes funciones de potencias: (1) función `pow` del módulo `math` de Python; y (2) función `power` que provee NumPy.

4.6. Vectorización

Como se indicó en la Sección 2, Numba delega la autovectorización en LLVM. Aun así, se indicaron los flags `avx512f, avx512dq, avx512cd, avx512bw, avx512vl` para favorecer el uso de esta clase particular de instrucciones.

4.7. Localidad de datos

Con el fin de mejorar la localidad de los datos, se implementó una versión que itera los cuerpos de a bloques, en forma similar a [9]. Para ello, el bucle de

la línea 4 de la Fig. 5 iterará sobre bloques de cuerpos, y en otros dos bucles más internos, se calculará la fuerza de atracción gravitacional de Newton y la integración de Verlet, respectivamente.

5. Resultados Experimentales

5.1. Diseño experimental

Todas las pruebas fueron realizadas en un sistema equipado con un 2×Intel 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 y el intérprete utilizado fue Python v3.8.10 junto con Numba v0.52.0 y NumPy v1.20.1.

Para la evaluación de las implementaciones, se varió la carga de trabajo al usar diferentes números de cuerpos: $N = \{4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288\}$ mientras que el número de pasos de simulación se mantuvo fijo (I=100). Cada optimización propuesta, fue aplicada y evaluada incrementalmente a partir de la versión *naive* [6]. Para la comparación final con la versión C+OpenMP se utilizó la implementación presentada en [15] usando el compilador ICC (versión 19.1.0.166).

5.2. Rendimiento

Para evaluar el rendimiento se emplea la métrica GFLOPS (mil millones de FLOPS), utilizando la fórmula $GFLOPS = \frac{20 \times N^2 \times I}{t \times 10^9}$, donde N es el número de cuerpos, I es el número de pasos, t es el tiempo de ejecución (en segundos) y el factor 20 representa la cantidad de operaciones en punto flotante requerida por cada interacción [7].

En la Fig. 6 se pueden observar los rendimientos al activar las opciones de compilación y aplicar multi-hilado al variar N. Aunque las opciones de compilación de Numba (`njit+fastmath+svml`) no tienen incidencia prácticamente en el rendimiento de esta versión, sí se aprecia una mejora importante al utilizar hilos para computar el problema. En particular, se puede notar una mejora en promedio de 33× y 38× para 56 y 112 hilos, respectivamente.

En la Fig. 7 se puede apreciar la mejora significativa que produce emplear arreglos con tipos de datos simples en lugar de compuestos (un promedio de 41× para el caso de 112 hilos). Si bien el segundo simplifica la codificación, también implica organizar los datos en forma de arreglo de estructuras (en inglés, *array of structures*), lo que impone limitaciones al aprovechamiento de las capacidades SIMD del procesador [11]. Adicionalmente, también se puede notar que el uso de *hyper-threading* reporta una mejora de aproximadamente 78 % en este caso.

De la Fig. 8 se puede observar que prácticamente no hay cambios en el rendimiento por el uso de los diferentes cálculos matemáticos y funciones de potencia que fueron descritos en la sección 4.5. Esto se debe a que, independientemente

[6] Cada versión previa está etiquetada como *Referencia* en todos los gráficos.
[7] Una convención ampliamente aceptada en la literatura para este problema.

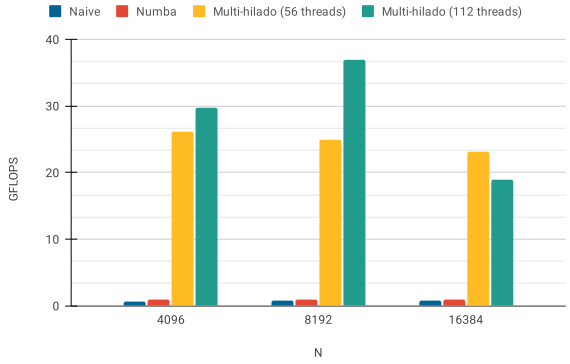

Figura 6: Rendimientos obtenidos para opciones de compilación y multi-hilado al variar N.

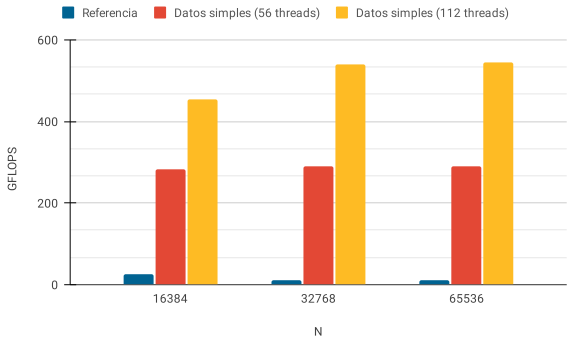

Figura 7: Rendimientos obtenidos de la optimización paralela al variar N.

de cuál opción se utilice, el código máquina resultante es siempre el mismo. Un caso similar se da al indicar explícitamente que utilice instrucciones AVX-512. Tal como se mencionó en la sección 2, Numba intenta autovectorizar el código a través de LLVM. Al observar el código máquina, se notó que las instrucciones generadas ya hacían uso de estas extensiones.

El procesamiento por bloques descrito en la sección 4.7 no mejoró el rendimiento de la solución, tal como se puede observar en la Fig. 9. La pérdida de rendimiento se relaciona con que esta reorganización del cómputo produce fallos en LLVM a la hora de autovectorizar. Lamentablemente, debido a que Numba no ofrece primitivas para indicar la utilización de instrucciones SIMD de forma explícita, no hay manera de enmendarlo.

En la Fig. 10 se muestran los rendimientos obtenidos para la relajación de precisión al variar el tipo de dato y la carga de trabajo (N). Se puede observar que el uso del tipo de datos `float32` (en lugar de `float64`) conlleva a una mejora

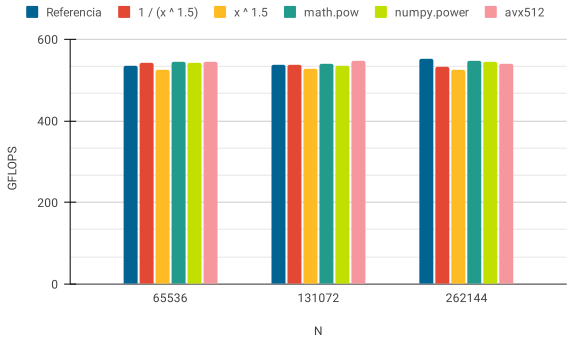

Figura 8: Rendimientos obtenidos utilizando el uso de diferentes cálculos matemáticos, funciones de potencia e instrucciones AVX512 al variar N.

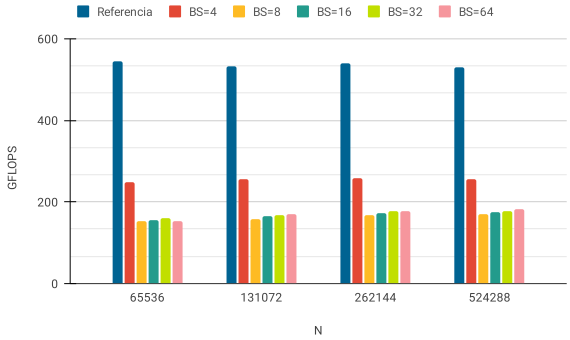

Figura 9: Rendimiento obtenido del procesamiento de a bloques al variar N.

de hasta 2.8× GFLOPS, a costo de una reducción en la precisión del resultado. En forma similar, se puede notar claramente la importante aceleración que produce relajar la precisión en ambos tipos de datos: `float32` (17.2× en promedio) y `float64` (11.4× en promedio). En particular el pico de rendimiento es de 1524/536 GFLOPS en simple/doble precisión. Resulta importante mencionar que la versión final de Numba logra una aceleración de 687× en comparación a la implementación *naive* (caso `float64`).

Por último, en la Fig. 11 se presenta una comparación entre la implementación C+OpenMP y la versión Python+Numba más rápida. Aunque se puede notar prácticamente el mismo rendimiento al utilizar tipos de datos de doble precisión, la versión C muestra una mejora de aproximadamente 1.7× por sobre Python al cambiar a precisión simple.

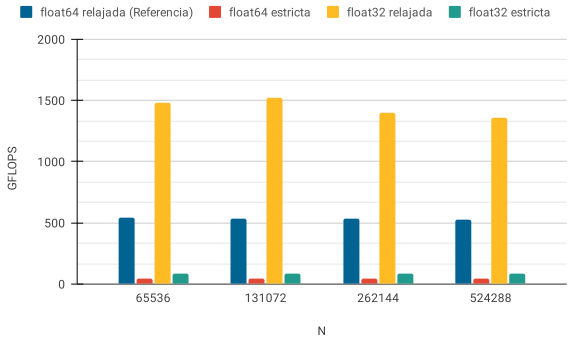

Figura 10: Rendimiento obtenido para la relajación de precisión al variar el tipo de dato y N.

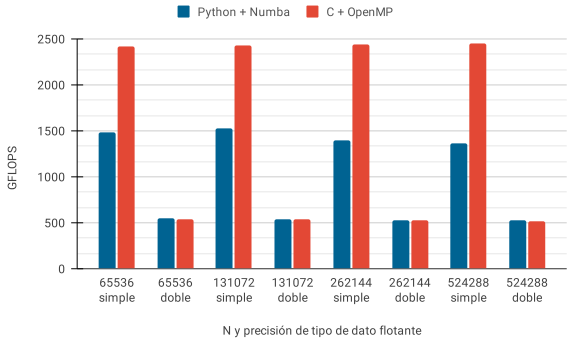

Figura 11: Comparación de rendimiento de la versión final Python+Numba y C+OpenMP variando el tipo de dato y N.

6. Trabajos Relacionados

A la fecha, existen unos pocos trabajos disponibles en la literatura que exploran las capacidades de Numba. El primero es el que lo introduce propiamente como un compilador JIT capaz de optimizar código Python [12].

Posteriormente, se presentaron dos estudios [10,13] que presentan a Numba como una opción factible para acelerar cómputo numérico, estando sus resultados en línea con los del presente trabajo. En [10] se muestra como una reducción a suma en 2D logra una mejora 200× frente una versión Python pura. Por su parte, en [13] se evalúa el rendimiento de Python junto a Numba, utilizando como caso de estudio el clásico algoritmo de producto de matrices, y luego, se lo compara frente a CUDA, C y cuBLAS GEMM. Como resultado, se obtuvo que Numba aumentó el rendimiento 1415×, mientras que con C se incrementó 2162×.

Por último, un estudio reciente [14] evalúa diferentes opciones para paralelizar código Python y las compara con versiones Fortran. Como caso de estudio, se optó por el problema *Five-point stencil* y, entre los resultados, se pudo observar que Numba logra importantes aceleraciones frente a Python, aunque quedó un poco lejos del rendimiento de Fortran. Si bien en este trabajo se comparó con el lenguaje C, las tendencias encontradas son similares.

7. Conclusiones y Trabajo Futuro

En este trabajo se evaluaron las capacidades y limitaciones de Numba como optimizador de código Python. Para ello, se utilizó como caso de estudio el problema numérico *N-Body* debido a su gran costo computacional. A una implementación base desarrollada en Python+NumPy, se le integró Numba y se le aplicaron diferentes optimizaciones posibles de manera incremental. En base los resultados experimentales, se puede decir de Numba lo siguiente:

- El multi-hilado fue una técnica efectiva para mejorar el rendimiento y su introducción no requirió cambios significativos en el código.
- Numba no sólo fue capaz de auto-vectorizar el código sino que lo hizo usando las instrucciones SIMD nativas del procesador (no hubo necesidad de que el programador las especifique). Sin embargo, la ausencia de primitivas de vectorización en este traductor (como `simd` en OpenMP) puede ser una limitación cuando la auto-vectorización no es posible, como se evidenció en la versión por bloques.
- El uso de arreglos de tipos de datos simples (en lugar de compuestos) llevó a importantes mejoras de rendimientos a costo de alargar y complicar un poco el código.
- Las diferentes maneras de computar operaciones y funciones matemáticas no tuvieron impacto en las prestaciones, por lo que su elección termina siendo por preferencia.
- Tanto la reducción como la relajación de precisión produjeron aceleraciones significativas en el rendimiento; sin embargo, se debe tener en cuenta que se vio afectada la representación numérica final como contraparte.

Adicionalmente, de la comparación con la versión C+OpenMP, se pudo notar un rendimiento similar al utilizar tipos de datos de doble precisión, mientras que al utilizar simple precisión la versión de C mostró una mejora de 1.7× sobre Python+Numba. Del análisis realizado se concluye que Numba puede ser una opción muy conveniente para acelerar cómputo numérico en Python, especialmente por su bajo costo de programación.

Como trabajos futuros, resulta de interés:

- Explorar otras capacidades y limitaciones de Numba no contempladas en este trabajo, como la utilización de GPUs.
- Replicar el estudio realizado con otros casos de estudio que sean computacionalmente intensivos pero cuyas características sean diferentes, en favor de robustecer los resultados encontrados.

- Dado que existen otras tecnologías que permitan implementar paralelismo en Python, realizar una comparación entre ellas considerando no sólo el rendimiento sino también el costo de programación.

Referencias

1. 7. Decorators — Python Tips 0.1 documentation, https://book.pythontips.com/en/latest/decorators.html
2. Applications for Python | Python.org, https://www.python.org/about/apps/
3. General Python FAQ — Python 3.9.5 documentation, https://docs.python.org/3/faq/general.html#what-is-python
4. Numba documentation — Numba 0.53.1-py3.7-linux-x86_64.egg documentation, https://numba.readthedocs.io/en/stable/index.html
5. NumPy, https://numpy.org/
6. Python vs C++ Comparison: Compare Python vs C++ Speed and More, https://www.bitdegree.org/tutorials/python-vs-c-plus-plus/
7. Python vs Java: What's The Difference? – BMC Software | Blogs, https://www.bmc.com/blogs/python-vs-java/
8. Top 12 Fascinating Python Applications in Real-World [2021] | upGrad blog, https://www.upgrad.com/blog/python-applications-in-real-world/
9. Costanzo, M., Rucci, E., Naiouf, M., Giusti, A.D.: Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body. In: 2021 XLVII Latin American Computer Conference (CLEI). p. In press (2021)
10. Crist, J.: Dask amp; numba: Simple libraries for optimizing scientific python code. In: 2016 IEEE International Conference on Big Data (Big Data). pp. 2342–2343 (2016). https://doi.org/10.1109/BigData.2016.7840867
11. Intel Corp.: How to manipulate data structure to optimize memory use on 32-bit intel® architecture (2018), https://tinyurl.com/26h62f76
12. Lam, S.K., Pitrou, A., Seibert, S.: Numba: a LLVM-based Python JIT compiler. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC. pp. 1–6. Austin, Texas (2015). https://doi.org/10.1145/2833157.2833162
13. Marowka, A.: Python accelerators for high-performance computing. The Journal of Supercomputing **74**(4), 1449–1460 (Apr 2018). https://doi.org/10.1007/s11227-017-2213-5, https://doi.org/10.1007/s11227-017-2213-5
14. Miranda, E.F., Stephany, S.: Common HPC Approaches in Python Evaluated for a Scientific Computing Test Case. REVISTA CEREUS **13**(2), 84–98 (Jul 2021), http://www.ojs.unirg.edu.br/index.php/1/article/view/3408, number: 2
15. Rucci, E., Moreno, E., Pousa, A., Chichizola, F.: Optimization of the n-body simulation on intel's architectures based on avx-512 instruction set. In: Computer Science – CACIC 2019. pp. 37–52. Springer International Publishing (2020)
16. SciPy.org: Scientific computing tools for Python (2021), https://www.scipy.org/about.html
17. TIOBE Software BV: TIOBE Index for August 2021 (08 2021), https://www.tiobe.com/tiobe-index/
18. Tipler, P.: Physics for Scientists and Engineers: Mechanics, Oscillations and Waves, Thermodynamics. Freeman and Co (2004)
19. Young, P.: The leapfrog method and other "symplectic" algorithms for integrating newton's laws of motion. Tech. rep., Physics Department, University of California, USA (4 2014), https://young.physics.ucsc.edu/115/leapfrog.pdf