Optimización del Cálculo de los Conjuntos de Julia utilizando Herramientas de Computación de Alto Rendimiento (HPC)

Benjamín Cea
Ingeniería Civil en Informática
Universidad Austral de chile
Email: benjamin.cea01@alumnos.uach.cl

***Abstract*—El cálculo de conjuntos de Julia puede implicar tiempos de ejecución extensos cuando se utilizan métodos tradicionales en Python, alcanzando incluso las dos horas para tamaños grandes. En este trabajo proponemos optimizar dicho cálculo utilizando herramientas de computación de alto rendimiento (HPC). La metodología contempla una implementación secuencial inicial, seguida de versiones optimizadas mediante vectorización con NumPy, compilación con Cython, vinculación con C++ y paralelismo CPU. Los resultados muestran mejoras significativas: se logra reducir el tiempo de ejecución de dos horas a tan solo cinco minutos utilizando técnicas avanzadas. Cada técnica se compara en términos de tiempo de ejecución bajo diferentes tamaños de entrada.**

***Index Terms*—Conjuntos de Julia, HPC, NumPy, Cython, Python, Paralelismo CPU (OpenMP), Optimización.**

I. Introducción

Los conjuntos de Julia son fractales generados a partir de funciones complejas iterativas. Su cálculo es intensivo en cómputo, especialmente cuando se requiere alta resolución o visualización animada. En implementaciones tradicionales en Python puro, los tiempos de ejecución pueden volverse prohibitivos.

En la última década, lenguajes dinámicos de alto nivel como Python y R han ganado protagonismo en áreas como inteligencia artificial, ciencia de datos y computación interactiva. Estos lenguajes, a pesar de su naturaleza interpretada, han logrado integrarse de manera eficiente en entornos de alto rendimiento mediante herramientas como Numba, Cython y JAX en Python (Godoy, 2023). Mientras que las implementaciones tradicionales de HPC siguen basadas en lenguajes compilados como C, C++ y Fortran, el avance de modelos de programación de alto nivel ha permitido cerrar la brecha entre productividad y rendimiento. Esta evolución ha sido fundamental para afrontar los desafíos de portabilidad y escalabilidad en arquitecturas modernas, caracterizadas por CPUs altamente paralelas y GPUs optimizadas para tareas específicas como aprendizaje profundo y álgebra lineal (Godoy, 2023).

A pesar de esto, portar aplicaciones computacionalmente intensivas a GPU sigue siendo un reto debido a la complejidad de programar en lenguajes de bajo nivel. En este contexto, propuestas como la de Besard et al. demuestran que es posible extender lenguajes de alto nivel como Julia para soportar dispositivos como GPUs, manteniendo una alta productividad sin sacrificar rendimiento. Su trabajo con CUDAnative.jl logra rendimientos comparables a los del toolkit oficial de NVIDIA CUDA, facilitando el desarrollo de código eficiente y portable desde un entorno de alto nivel (Besard, 2018).

Pregunta de investigación: ¿Es posible optimizar la generación de conjuntos de Julia utilizando herramientas modernas de HPC sin abandonar la productividad de lenguajes como Python?

II. Estado del Arte

Actualmente, el cálculo de fractales como los conjuntos de Julia se realiza principalmente en el ámbito académico y artístico, en áreas como la matemática computacional, gráficos generativos y visualización científica. Herramientas como NumPy, Cython, y frameworks como TensorFlow y JAX han sido aplicadas para acelerar tareas numéricas intensivas. En entornos de HPC, se aprovechan arquitecturas paralelas (CPU/GPU) para mejorar el rendimiento en simulaciones físicas y procesamiento de imágenes, incluyendo generación de fractales.

Agregando que también hoy en dia además de Python, lenguajes como C/C++ se emplean ampliamente para implementar algoritmos de fractales debido a su eficiencia y control sobre el hardware. Rust ultimamente también ha ganado popularidad por su rendimiento y seguridad, pero para este trabajo no lo pondremos aprueba. En el ámbito web, WebAssembly (WASM) permite ejecutar cálculos intensivos de fractales en navegadores modernos con alto rendimiento.

III. Solución Propuesta

Proponemos una mejora iterativa del rendimiento computacional del cálculo de los conjuntos de Julia, utilizando distintas técnicas de HPC. El flujo es el siguiente:

- Implementación secuencial en Python.
- Optimización con NumPy y vectorización.
- Compilación del núcleo con Cython.
- Uso de C++ mediante bindings.
- Paralelismo con `multiprocessing`.

IV. Metodología

Las pruebas se realizaron en un equipo con procesador AMD Ryzen 5 5000 series, 24 GB RAM y AMD Radeon.

Se utilizó un tamaño de imagen de 800x800 píxeles, y se mantuvieron constantes los parámetros de iteración. Las versiones se evaluaron con la herramienta `time` para medir tiempos precisos. Se usaron bibliotecas como NumPy, Cython, OpenMP y pybind11.

A. Consideración sobre el valor de c en la carga computacional

Durante la implementación inicial del conjunto de Julia, se utilizó el valor tradicional $c = -0.8 + 0.156i$, el cual genera un fractal con una estructura definida y que permite visualizar claramente la frontera entre los puntos que escapan y los que no. Pero al ejecutar el código he ir variando el valor de **max_iter** se observó que el tiempo de ejecución con este valor no variaba significativamente. Esto ocurre porque, con este valor de c, una gran parte de los puntos en la grilla escapan rápidamente del conjunto, haciendo que el bucle iterativo interno se interrumpa de forma temprana para la mayoría de los píxeles.

Para obtener un perfil de carga computacional más exigente y homogéneo (necesario para realizar pruebas comparativas entre distintas técnicas de optimización) se optó por utilizar el valor $c = 0$. En este caso, la función iterada se convierte en:

$$z_{n+1} = z_n^2$$

donde:

- Si $|z_0| < 1$, el punto converge a 0 (no escapa).
- Si $|z_0| > 1$, el punto diverge rápidamente (escapa).
- Si $|z_0| = 1$, el punto se mantiene oscilando (límite crítico).

Como consecuencia, una gran proporción de los puntos en la grilla no escapan y alcanzan el límite de iteraciones máximo. Esto implica una carga computacional mucho mayor y más uniforme, lo que hace que el tiempo de ejecución esté más correlacionado con los parámetros `max_iter`, `width` y `height`.

Esta configuración fue clave para evaluar de forma precisa el impacto de cada técnica de optimización, ya que permitió simular un escenario de alto costo computacional.

V. Resultados

A continuación se resumen los tiempos promedio de ejecución obtenidos con cada técnica:

Técnicas
Secuencial Python
NumPy Vectorizado
Cython
C++ Bindings
Paralelismo CPU

TABLE I: Tabla con cada técnica

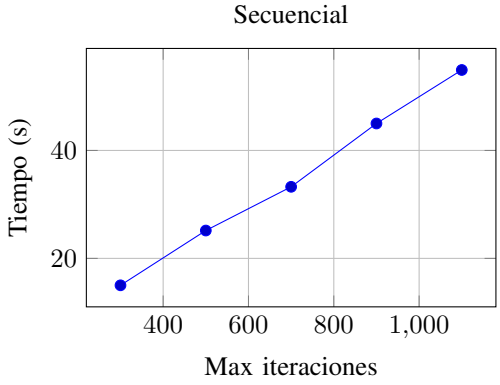

Fig. 1: Tiempos de ejecución de forma secuencial.

Al observar la Fig. 1, se puede afirmar que la ejecución secuencial resulta poco eficiente. Su crecimiento lineal con respecto al número de iteraciones provoca una mala escalabilidad ante cargas mayores. Aunque es útil como referencia base, no es una opción viable para escenarios en tiempo real.

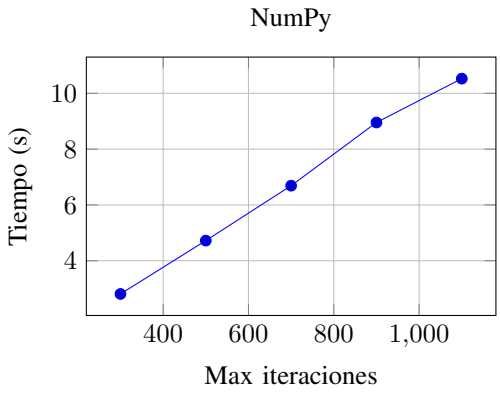

Fig. 2: Tiempos de ejecución usando técnica de NumPy.

De la Fig 2 la vectorización con NumPy reduce los tiempos de forma importante (5 veces más rápido que el secuencial). Es una técnica fácil de implementar y útil para obtener mejoras rápidas sin cambiar el lenguaje base. Aun asi no es suficiente si es que se quiere aumentar el número de iteracciones.

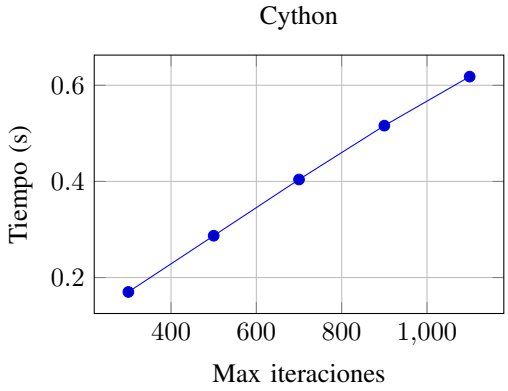

Fig. 3: Tiempos de ejecución usando técnica de Cython.

De la Fig. 3 podes darnos cuenta que compilar la lógica crítica con Cython proporciona una aceleración significativa (90x más rápido que Python puro), siendo ideal cuando se busca rendimiento sin salir completamente de Python.

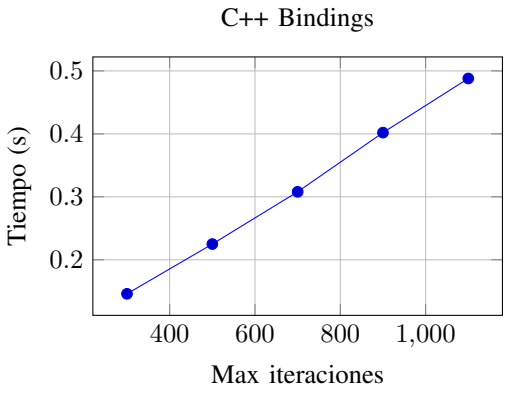

Fig. 4: Tiempos de ejecución usando técnica de C++ bindings.

A partir de la Fig. 4, se observa que utilizar C++ y exponer funciones a Python mediante bindings como pybind11 ofrece un notable mejora en el rendimiento. Aunque su implementación es más compleja que la de Cython, resulta ideal para reutilizar código C/C++ existente o para alcanzar un rendimiento cercano al nativo.

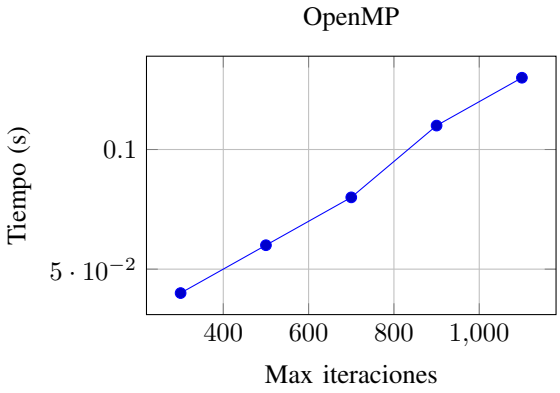

Fig. 5: Tiempos de ejecución usando técnica de Paralelismo CPU (OpenMP).

Según la Fig. 5, el paralelismo en CPU utilizando OpenMP proporciona la mayor eficiencia observada. Al distribuir la carga entre múltiples núcleos, se logra minimizar significativamente los tiempos de ejecución sin requerir una GPU. Esta estrategia es especialmente adecuada para arquitecturas multihilo bien optimizadas.

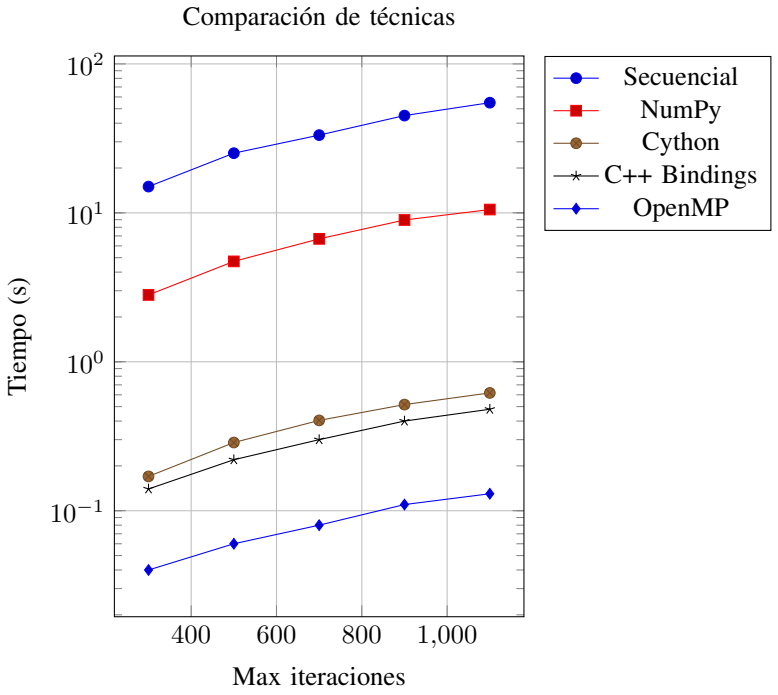

Fig. 6: Comparación de tiempos según técnica utilizada (escala logarítmica).

El análisis de la Fig. 6 muestra que cada técnica de optimización mejora notablemente el rendimiento respecto a la versión secuencial en Python, cuyo tiempo crece rápidamente con las iteraciones (más de 50 s para 1100). NumPy reduce los tiempos gracias a la vectorización (10.5 s vs. 54.9 s), pero sigue siendo interpretado. Cython y C++ Bindings bajan los tiempos a menos de 1 segundo, aprovechando la compilación y estructuras nativas. OpenMP es la opción más eficiente, logrando tiempos menores a 0.15 segundos incluso en los casos más exigentes, gracias al paralelismo en CPU.

VI. Conclusión

El presente trabajo demostró que es posible optimizar significativamente el cálculo de los conjuntos de Julia utilizando herramientas de computación de alto rendimiento (HPC), sin abandonar la productividad del ecosistema Python. Se evaluaron diversas técnicas, comenzando con una implementación secuencial como línea base, y progresivamente incorporando optimizaciones como NumPy (vectorización), Cython (compilación a C), C++ con `pybind11`, y finalmente paralelismo a nivel de CPU mediante OpenMP.

Los resultados muestran una mejora de rendimiento de más de dos órdenes de magnitud al pasar de Python puro a versiones compiladas y paralelizadas. En particular:

- La vectorización con NumPy redujo los tiempos hasta 5 veces en comparación al secuencial, con mínima complejidad adicional.
- Cython y C++ bindings ofrecieron aceleraciones aún mayores, alcanzando mejoras superiores al 90% respecto a Python puro.

- OpenMP resultó ser la técnica más eficiente entre las evaluadas, con tiempos casi constantes frente al incremento de iteraciones, debido al aprovechamiento del paralelismo en CPU.

Además, se destaca que el valor del parámetro c en la función iterativa tiene un impacto directo en la carga computacional, lo cual debe ser cuidadosamente considerado al diseñar benchmarks.

Como trabajo futuro se propone explorar la aceleración por GPU mediante bibliotecas como CuPy o Numba CUDA, así como evaluar técnicas distribuidas con MPI. También se plantea estudiar el uso frameworks modernos para lograr un rendimiento comparable sin salir de Python. Esta investigación reafirma el potencial de las herramientas HPC modernas para hacer viable el uso de lenguajes de alto nivel en aplicaciones computacionalmente intensivas.

Todo el código fuente se encuentra disponible en:
https:/github.com/elD4vil/HPC

REFERENCES

1.- William F. Godoy et al. Evaluating performance and portability of high-level programming models: Julia, Python/Numba, and Kokkos on exascale nodes.

2.- Tim Besard et al. Effective Extensible Programming: Unleashing Julia on GPUs.