|  | **Arquitectura de Computadores**  **Informe del Trabajo Final -** |
| --- | --- |
|  |  |

**Nombres de estudiantes**: Juan Esteban Eraso, Nicolas Cardona, Santiago Espinosa,Thomas Brueck

**Códigos**:A00399655 A00373470 A00399531 A00399947

**-Título del experimento**-

Experimento de 4 factores: multiplicación de matrices

**Fecha de entrega**: 12 de Junio 2025

**Profesor**: Carlos Diaz Andrade

**INFORME DEL DESARROLLO DE EXPERIMENTO - Título del experimento**

**1. Introducción**

Este experimento se centra en evaluar el impacto combinado de cuatro factores clave —versión del algoritmo, tamaño de la matriz, tipo de dato y lenguaje de programación— sobre el rendimiento computacional en la multiplicación de matrices cuadradas, una operación fundamental en arquitectura de computadores. La multiplicación de matrices es intensiva en cómputo y depende en gran medida de la localidad espacial y temporal de los datos, aspectos críticos para optimizar el uso de la memoria caché. La jerarquía de memoria, que incluye niveles de caché (L1, L2, L3) y memoria principal, influye significativamente en el desempeño, ya que el reuso efectivo de datos en caché reduce los accesos a memorias de mayor latencia.

El objetivo de este estudio es analizar cómo las decisiones algorítmicas e implementativas afectan el rendimiento, considerando dos unidades experimentales con configuraciones de caché distintas. Este enfoque se alinea con los principios de diseño de experimentos, que destacan la importancia de controlar variables y realizar repeticiones para manejar la variabilidad. La localidad de referencia, un concepto clave, sugiere que algoritmos con patrones de acceso optimizados aprovechan mejor la caché, disminuyendo fallos y mejorando el rendimiento.

La relevancia de este experimento radica en su aporte práctico para la optimización de código en arquitecturas modernas, donde la brecha entre la velocidad de los procesadores y la memoria sigue ampliándose(Memory Wall). Los resultados permitirán identificar configuraciones óptimas y sugerir mejoras como la multiplicación por bloques para manejar matrices que excedan la capacidad de la caché.

**2. Definición del Problema, Objetivos y Teoría Relevante**

2.1 **Identificación del problema**

El rendimiento de los algoritmos de multiplicación de matrices varía significativamente dependiendo de cómo se acceden los datos en la memoria. Esta variación está influenciada por factores como la **localidad espacial** y **temporal**, que determinan la eficiencia con la que los datos son aprovechados por la memoria caché dentro de la jerarquía de memoria. Además, los lenguajes de programación utilizados (C, Python y Java) introducen diferencias en la gestión de memoria y la ejecución del código debido a sus modelos de traducción (compilado, interpretado y máquina virtual, respectivamente), lo que también afecta el tiempo de ejecución.

El estudio se centra en el impacto de la localidad de datos en el rendimiento de algoritmos de multiplicación de matrices, implementados en C, Python y Java. Se analizarán específicamente:

* El **orden de los bucles** en los algoritmos, que afecta la forma en que se recorren las matrices.
* El **tamaño de las matrices**, que influye en cómo se distribuyen los datos en la jerarquía de memoria.
* El **tipo de dato** (float o double), que determina el tamaño de los elementos y su interacción con la caché.
* El **lenguaje de programación**, que introduce variaciones en la ejecución y gestión de memoria.

Ahora, pasando con las hipótesis que se pretenden probar, las plantearemos teniendo en cuenta que se utilizará un ANOVA de cuatro vías. Este tipo de ANOVA es apropiado porque estamos analizando simultáneamente el efecto de cuatro factores independientes sobre la variable dependiente/de respuesta: el **tiempo de ejecución**. Cada factor tiene los siguientes niveles:

* **Versión del Algoritmo**: 6 niveles (a, b, c, d, e, f)
* **Tamaño del Arreglo N**: 12 niveles (91, 128, 157, 181, 200, 256, 362, 400, 512, 836, 1182, 1672)
* **Tipo de Dato**: 2 niveles (float, double)
* **Lenguaje de Programación**: 3 niveles (C, Python, Java)

El número de "vías" en el ANOVA corresponde al número de factores analizados, y este diseño permite evaluar:

* **Efectos principales**: Cómo cada factor individual afecta el tiempo de ejecución.
* **Interacciones**: Cómo las combinaciones de factores (por ejemplo, una versión del algoritmo con un tamaño de arreglo específico en un lenguaje dado) influyen en el rendimiento de manera no aditiva.

Teniendo esto en cuenta, planteamos **4 hipótesis principales:**

* **Versión del Algoritmo** (6 niveles: a, b, c, d, e, f)
  + (H\_0): No hay diferencia en el tiempo de ejecución promedio entre las seis versiones.
  + (H\_1): Al menos una versión tiene un tiempo de ejecución diferente.
* **Tamaño del Arreglo N** (12 niveles: 32, 64, ..., 384)
  + (H\_0): No hay diferencia en el tiempo de ejecución promedio entre los tamaños.
  + (H\_1): Al menos un tamaño tiene un tiempo de ejecución diferente.
* **Tipo de Dato** (2 niveles: float, double)
  + (H\_0): No hay diferencia en el tiempo de ejecución entre float y double.
  + (H\_1): Hay una diferencia entre float y double.
* **Lenguaje de Programación** (3 niveles: C, Python, Java)
  + (H\_0): No hay diferencia en el tiempo de ejecución entre C, Python y Java.
  + (H\_1): Al menos un lenguaje tiene un tiempo de ejecución diferente.

**2.2 Teoría Relevante**

##### Jerarquía de Memoria y Memoria Caché

La jerarquía de memoria es un sistema organizado en niveles de almacenamiento con diferentes capacidades, costos y tiempos de acceso. Según el Capítulo 6 del libro "Computer Systems: A Programmer's Perspective" (páginas 560-561), esta jerarquía incluye:

* **Registros de la CPU:** Almacenan datos de uso frecuente con acceso instantáneo (0 ciclos).
* **Memoria caché (L1, L2, L3):** Memorias rápidas cercanas a la CPU (1-30 ciclos), que actúan como áreas de preparación para datos e instrucciones de la memoria principal.
* **Memoria principal (DRAM):** Más lenta (50-200 ciclos), pero con mayor capacidad.
* **Discos (HDD/SSD):** Mucho más lentos (millones de ciclos), usados para almacenamiento masivo.

Cada nivel inferior es más lento, pero ofrece mayor capacidad y menor costo por bit. El objetivo de la jerarquía es proporcionar un acceso rápido a los datos más utilizados, aprovechando el **principio de localidad** (página 560):

* **Localidad temporal:** Si un dato se accede, es probable que se acceda de nuevo pronto.
* **Localidad espacial:** Si un dato se accede, es probable que los datos cercanos en memoria también se necesiten pronto.

La **memoria caché** juega un papel crucial en el rendimiento (Sección 6.4, páginas 596-615). Actúa como un buffer entre la CPU y la memoria principal, almacenando subconjuntos de datos en **líneas de caché** (típicamente de 4 a 64 bytes). Su eficiencia depende de cómo los programas acceden a los datos. Por ejemplo, en la multiplicación de matrices, el orden de los bucles determina si los accesos son secuenciales (mejor localidad espacial) o dispersos (más fallos de caché). Un acceso secuencial a una matriz en orden row-major (como en C) reduce los fallos de caché, mejorando el tiempo de ejecución (Sección 6.5, páginas 615-620).

El tamaño de la caché también es crítico: si las matrices exceden su capacidad, los datos se desalojan con frecuencia, aumentando los fallos. Además, el tipo de dato (float de 4 bytes vs. double de 8 bytes) afecta cuántos elementos caben en una línea de caché, influyendo en el rendimiento (Sección 6.6, página 620).

##### Diferencias en la Traducción de Lenguajes

Los lenguajes de programación utilizados (C, Python y Java) tienen modelos de ejecución distintos que impactan el rendimiento:

* **C (Compilado):** C se traduce directamente a código máquina (Capítulo 6, página 615, implícito en ejemplos de optimización). Esto permite un control preciso sobre la gestión de memoria y optimizaciones a nivel de hardware, minimizando la sobrecarga y maximizando la eficiencia en tareas intensivas como la multiplicación de matrices.
* **Python (Interpretado):** Python ejecuta el código línea por línea mediante un intérprete, lo que introduce una sobrecarga significativa en tiempo de ejecución. Aunque no se detalla en el Capítulo 6, esta característica implica que Python es menos eficiente en comparación con lenguajes compilados, especialmente en operaciones de bajo nivel como accesos repetitivos a memoria.
* **Java (Máquina Virtual):** Java se compila a bytecode, ejecutado por la Máquina Virtual Java (JVM). La JVM realiza optimizaciones en tiempo de ejecución (compilación JIT), pero introduce una sobrecarga inicial y depende de la gestión automática de memoria (garbage collection), lo que puede generar pausas impredecibles y afectar el rendimiento en comparación con C. Aún así, se espera un rendimiento superior a Python.

**3. Selección de Factores**

3.1 **Identificación de factores primarios y secundarios**

El experimento del proyecto final se estructurará considerando cuatro factores con los siguientes niveles:

* **Factor 1: Versión del Algoritmo (6 niveles):** a, b, c, d, e, f  
  Se seleccionan seis versiones diferentes del algoritmo de multiplicación de matrices, cada una con una estrategia distinta de recorrido y acceso a memoria.
* **Factor 2: Tamaño del Arreglo N (12 niveles):** 91, 128, 157, 181, 200, 256, 362, 400, 512, 836, 1182, 1672  
  Los tamaños de matriz se definieron con base en la jerarquía de memoria caché del sistema (L1, L2, L3) del computador de un estudiante, calculados considerando la capacidad de caché y el tamaño en bytes del tipo de dato (4 bytes para float, 8 bytes para double), asegurando un mínimo de 12 niveles para evaluar el impacto de la jerarquía de memoria.
* **Factor 3: Tipo de Dato (2 niveles):** float y double  
  Se analizará cómo el uso de diferentes tipos de datos afecta la precisión numérica y el tiempo de ejecución.
* **Factor 4: Lenguaje de Programación (3 niveles):** C, Python, Java  
  Se evaluará el impacto del lenguaje de programación en el rendimiento computacional, considerando aspectos como la optimización de compiladores y la administración de memoria.

**Factores secundarios o de ruido:** Procesos en segundo plano, temperatura del procesador, diferencias en compiladores/intérpretes, carga de caché por otros procesos.

**Estrategias de control:** Ejecutar en modo consola aislado, cerrar aplicaciones innecesarias, usar versiones estandarizadas de compiladores/intérpretes, monitorear temperatura y pausar si hay picos térmicos.

**4. Diseño Experimental**

4.1 **Elección del diseño experimental**

Se seleccionará un diseño factorial completo 6×12×2×3 para evaluar todas las combinaciones posibles de los factores (432 tratamientos únicos), justificado por la necesidad de analizar interacciones entre versión del algoritmo, tamaño de matriz, tipo de dato y lenguaje de programación. Este diseño se adapta a la hipótesis de que la localidad espacial y temporal, influenciada por estos factores, afecta el rendimiento en la jerarquía de memoria.

Se realizarán 15 repeticiones por tratamiento para reducir variabilidad, resultando en 6480 ejecuciones totales. Para elegir nuestro número de repeticiones se tomó en cuenta el cálculo del tamaño de muestra:

Inicialmente, la idea fue implementar tamaños de muestra específicos para cada combinación de factores, calculados a partir de 5 ejecuciones preliminares por tratamiento. Esto se basó en la fórmula estándar para estimar la media con un 95% de confianza y un error relativo del 5%:



donde (z = 1.96) (valor crítico para un 95% de confianza), (σ) es la desviación estándar de las ejecuciones preliminares, y (E = 0.05 x promedio) es el error absoluto permitido.

Por ejemplo, para **C float versión E** con (N=91, donde N es el tamaño de la matriz), las ejecuciones preliminares dieron (σ = 0.727) y un promedio de 1.8578, resultando en (n = 235). Para **Java double versión C** con (N=91), con (σ = 1.253) y promedio = 1.0250, se obtuvo (n = 2295). Sin embargo, en el caso de **Python**, tanto para tipos *float* como *double*, en todas las versiones del algoritmo y tamaños de matriz, la variabilidad fue tan baja que (σ ≈ 0), lo que llevó a (n ≈ 0). Esto sugiere que, teóricamente, incluso con muy pocas repeticiones, la estimación de la media sería suficientemente precisa en estos casos.

Aunque esta aproximación inicial de tamaños específicos parecía ideal para capturar la variabilidad única de cada tratamiento, la disparidad en los resultados —desde (n ≈ 0) en Python hasta (n = 235) en C y (n = 2295) como máximo en Java— generó desafíos prácticos. Implementar tamaños de muestra distintos para cada combinación aumentaría la complejidad del análisis posterior, y sería computacionalmente inviable. Por ejemplo, adoptar el máximo calculado ((n = 235)) para los 432 tratamientos implicaría más de 900,000 ejecuciones, superando las restricciones de tiempo y de recursos del proyecto (por ejemplo, la imposibilidad de dejar dos computadores prendidos durante tanto tiempo).

Por ello, se decidió usar un tamaño de muestra uniforme de 15 repeticiones por tratamiento, totalizando 6,480 ejecuciones, basándose en los siguientes argumentos:

1. **Aprovechamiento de la baja variabilidad en Python:** En los tratamientos de Python, donde (n ≈ 0) debido a una variabilidad mínima, incluso una sola repetición sería suficiente para estimar la media con alta precisión. Elegir 15 repeticiones no solo satisface esta necesidad teórica, sino que también nos ayuda a identificar posibles fluctuaciones o datos atípicos no detectados en las ejecuciones preliminares, alineándose con el principio de precaución estadística.
2. **Reducción razonable de variabilidad en casos más dispersos:** Para tratamientos con mayor variabilidad, como algunos en C o Java, 15 repeticiones no alcanzan el tamaño calculado (e.g., (n = 235)), pero sí reducen significativamente el error muestral. Dado que el objetivo principal es comparar el rendimiento relativo entre tratamientos mediante ANOVA, y no obtener estimaciones absolutas exactas, 15 repeticiones ofrecen suficiente información estadística para detectar diferencias significativas entre los niveles de los factores.
3. **Balance y simplicidad del diseño:** Un tamaño uniforme mantiene nuestro diseño factorial del experimento balanceado, facilitando el análisis estadístico con ANOVA para evaluar efectos principales e interacciones, sin complicar la interpretación debido a tamaños de muestra variables.

No se implementará bloqueo ni otras restricciones de aleatorización, ya que no son necesarias para el diseño experimental seleccionado. Además, se realizarán repeticiones en lugar de réplicas, dado que todas las ejecuciones se llevarán a cabo bajo las mismas condiciones experimentales, sin variaciones en el entorno entre ellas.

**5. Experimentación Preliminar**

5.1 **Desarrollo de experimentos preliminares**

La experimentación preliminar se diseñó para validar los algoritmos de multiplicación de matrices, establecer parámetros experimentales clave y garantizar un entorno de ejecución confiable. Estas pruebas iniciales permitieron adquirir experiencia con las variables experimentales, confirmar la ausencia de variables no identificadas, determinar límites seguros para los factores, verificar la precisión de los procedimientos, validar el funcionamiento de los algoritmos y estimar la variabilidad de las respuestas para definir el tamaño de muestra adecuado. Este apartado detalla los experimentos realizados, los resultados obtenidos, los problemas identificados y los ajustes realizados para minimizar riesgos en los experimentos definitivos.

Se realizaron experimentos preliminares para abordar los vacíos de conocimiento en el diseño del experimento, ejecutando pruebas con los siguientes factores y niveles: Tipo de algoritmo (A, B, C, D, E, F), Tamaño (91, 128, 157, 181, 200, 256, 362, 400, 512, 836, 1182, 1672), Tipo de dato (float, double), y Lenguaje de programación (C, Python, Java).

### Objetivo

Los experimentos preliminares buscaron validar la correcta implementación de los algoritmos de multiplicación de matrices, determinar el número óptimo de repeticiones, evaluar el entorno de ejecución, establecer límites seguros para los tamaños de matriz, optimizar el acceso a caché y resolver problemas iniciales para reducir la incertidumbre en las fases posteriores del diseño de experimentos (DOE).

### Metodología

Se llevaron a cabo dos fases de experimentos preliminares:

#### Fase 1: Pruebas Iniciales con n = 5

* **Descripción**: Se desarrollaron algoritmos de multiplicación de matrices en C, Java y Python para un tamaño de muestra fijo de n = 5.
* **Entorno**: Las pruebas se ejecutaron en una máquina con procesador AMD Ryzen 5 (arquitectura x64), utilizando Visual Studio 2022 para C y Java, y un entorno compatible para Python Visual Studio Code.
* **Propósito**: Confirmar la funcionalidad de los algoritmos y comparar el rendimiento inicial entre lenguajes.
* **Mediciones**: Se midieron los tiempos de ejecución para validar la implementación y establecer una línea base.

#### Fase 2: Pruebas con Seis Versiones en C (a-f)

* **Descripción**: Se implementaron seis variantes del algoritmo de multiplicación de matrices en C (a, b, c, d, e, f), presumiblemente con diferentes órdenes de bucles (ej., ijk, ikj, kji).
* **Tamaños de Matriz (N)**: Se probaron tamaños de N = 91, 128, 157, 181, 200, 256, 362, 400, 512, 836, 1182 y 1672, comenzando con un rango de 91 a 128 y extendiéndose para analizar límites de rendimiento.
* **Tipos de Datos**: Se utilizaron float y double para evaluar el impacto de la precisión.
* **Entorno**: Visual Studio 2022 en modo Release, arquitectura x64, con aislamiento de procesos y frecuencia del procesador fija.
* **Repeticiones**: Se realizaron 5 repeticiones por tratamiento inicialmente, con análisis estadístico para determinar el número óptimo de repeticiones.
* **Propósito**: Identificar la variante más eficiente, optimizar el uso de caché, establecer límites seguros para N y refinar los procedimientos experimentales.

#### Factores Experimentales

* **Factores Primarios**:
  + Lenguaje de programación: C, Java, Python (Fase 1).
  + Variante del algoritmo: a, b, c, d, e, f (Fase 2).
  + Tamaño de matriz (N): 91, 128, 157, 181, 200, 256, 362, 400, 512, 836, 1182, 1672.
  + Tipo de dato: float, double.
* **Variable de Respuesta**: Tiempo de ejecución (en segundos) y rendimiento relativo (operaciones por segundo).
* **Factores Secundarios**:
  + Entorno de ejecución: Visual Studio 2022 (x64).
  + Capacidad de caché L3: Límite seguro N=384.
  + Frecuencia del procesador: Fija para consistencia.
* **Control de Factores Secundarios**:
  + Aislamiento de procesos no esenciales mediante configuración del sistema.
  + Fijación de la frecuencia del procesador para minimizar variaciones.
  + Uso de herramientas de medición de alta precisión en Visual Studio 2022.

#### Entorno de Ejecución

* **Hardware**: Procesador AMD Ryzen 5, arquitectura x64, con caché L3 suficiente para matrices hasta N=384.
* **Software**: Visual Studio 2022, configurado en modo Release con optimizaciones habilitadas.
* **Procedimientos**:
  + Se desactivaron procesos no esenciales durante las pruebas.
  + Se fijó la frecuencia del procesador para garantizar mediciones consistentes.
  + Se utilizaron temporizadores de alta precisión integrados en Visual Studio.

### Resultados

#### Fase 1: Pruebas Iniciales (n=5)

* **Validación**: Los algoritmos en C, Java y Python produjeron resultados correctos para matrices de n=5, confirmando su funcionalidad.
* **Rendimiento**: C mostró los mejores tiempos de ejecución, seguido por Java, mientras que Python fue significativamente más lento. Los tiempos fueron mínimos debido al pequeño tamaño de la matriz, pero sirvieron para justificar el uso de C en pruebas posteriores.
* **Conclusión**: La validación exitosa permitió centrar los experimentos en C para optimizar el rendimiento.

#### Fase 2: Pruebas con Versiones a-f en C

Se analizaron los tiempos de ejecución promedio (en segundos) de las seis versiones en C (a-f) para float y double, basados en 5 repeticiones. A continuación, se presenta una tabla resumida para tamaños de matriz seleccionados:

| **N** | **Ver(c) float** | **Ver(d) float** | **Ver(c) double** | **Ver(d) double** |
| --- | --- | --- | --- | --- |
| 91 | 0.0018 | 0.0018 | 0.0018 | 0.0018 |
| 128 | 0.0046 | 0.0050 | 0.0048 | 0.0046 |
| 157 | 0.0088 | 0.0088 | 0.0088 | 0.0090 |
| 256 | 0.0376 | 0.0372 | 0.0378 | 0.0376 |
| 512 | 0.2984 | 0.2990 | 0.2982 | 0.3006 |
| 836 | 1.2920 | 1.2952 | 1.2956 | 1.3048 |
| 1182 | 3.6624 | 3.6624 | 3.6616 | 3.6810 |
| 1672 | 10.3166 | 10.3644 | 10.3622 | 10.4236 |

En esta tabla se presenta una breve comparación entre la versión C y D de float y double para el lenguaje C, sin embargo, los datos se pueden apreciar mejor y lo mismo que las siguientes observaciones, al visualizar el excel con todos los datos por cada lenguaje, tamaño y tipo de dato.

**Observaciones**:

* La versión c (float y double) mostró los mejores tiempos de ejecución, especialmente para N ≥ 256, debido al orden de bucles ikj, que optimiza el acceso a caché.
* Las versiones e y f fueron las menos eficientes para N ≥ 1182, con tiempos significativamente mayores (e.g., 17.2202 s para Ver(e) float en N = 1672).
* Los tiempos con double fueron ligeramente mejores que con float en la versión c para matrices grandes, probablemente por optimizaciones en el manejo de datos.
* La desviación estándar de las repeticiones fue baja (≤ 5% del promedio), indicando alta consistencia en las mediciones, debido al control del entorno (frecuencia fija, aislamiento de procesos).

#### Análisis de Caché

* El orden ikj (versión c) redujo los fallos de caché L1 y L2, mejorando el rendimiento.
* El límite seguro de N=384 se estableció según la capacidad de la caché L3 del Ryzen 5, evitando saturación.

#### Determinación del Número de Repeticiones

* La desviación estándar de las repeticiones iniciales fue baja (< 5%), pero se optó por 10 repeticiones después para garantizar mayor robustez estadística sin un costo computacional excesivo.

### Identificación de Problemas

* **Python**: Los tiempos de ejecución en Python fueron significativamente mayores, lo que llevó a descartarlo.
* **Interferencias**: Las pruebas iniciales mostraron variabilidad por procesos en segundo plano, resuelto mediante aislamiento de procesos y fijación de la frecuencia del procesador.
* **Saturación de Caché**: Para N > 384, los tiempos crecieron de forma no lineal, indicando saturación de caché L3.
* **Versiones e y f**: Mostraron ineficiencias para matrices grandes, sugiriendo problemas en el orden de acceso a memoria.

### Validación de Algoritmos

* **Fase 1**: Las implementaciones en C, Java y Python generaron resultados correctos para n=5.
* **Fase 2**: Las seis versiones en C (a-f) produjeron resultados correctos para todos los tamaños de matriz.

### Evaluación de Límites Seguros

* **Tipos de Datos**: float y double fueron viables, con double ligeramente mejor para matrices grandes.
* **Repeticiones**: 10 repeticiones como tamaño de muestra después del tamaño de muestra de 5 se consideraron suficientes para minimizar la incertidumbre.

### Ajustes Realizados

* **Entorno**: Visual Studio 2022 se configuró con aislamiento de procesos y frecuencia fija.
* **Repeticiones**: Se aumentó a 10 repeticiones para experimentos definitivos para la siguiente ejecución con otro tamaño de muestra.
* **Posibles problemas identificados:** Fluctuaciones en tiempos debido a temperatura del procesador y procesos en segundo plano.
* **Ajustes realizados:** Se configuró el entorno en "Alto rendimiento", se cerraron aplicaciones innecesarias, y se aumentó el tamaño de muestra de 5 a 15 repeticiones para mayor fiabilidad.

**Conclusiones**

* **Adquirir experiencia con nuevas variables experimentales:** Se probaron las seis versiones del algoritmo en los tamaños especificados, observando que versiones como A y F mostraron diferencias en rendimiento según el tamaño de matriz.
* **Confirmar que no hay variables no identificadas:** No se identificaron factores externos significativos más allá de variaciones de temperatura y procesos en segundo plano, manejables con ajustes.
* **Identificar límites seguros:** Se validaron tamaños de matriz de 91 a 1672 como rango seguro, ajustado para evitar saturación de memoria o tiempos excesivos.
* **Confirmar precisión de procedimientos:** Las mediciones de tiempo con herramientas como clock() en C y equivalentes en Python/Java fueron consistentes tras estandarizar el entorno.
* **Validación de algoritmos y equipo:** Todos los algoritmos funcionaron como esperado en los tres lenguajes, y el equipo mostró estabilidad tras minimizar interferencias.
* **Estimación de desviación estándar y tamaño de muestra:** Con 5 repeticiones iniciales por combinación, se estimó la desviación estándar, determinando que 15 repeticiones por tratamiento eran necesarias. Se ejecutaron 10 adicionales, sumando 15 en total.

**6. Ejecución del Experimento**

6.1 **Procedimiento experimental**

6.1 **Procedimiento Experimental**

Equipos y Configuración:

* Se utilizó un sistema con procesador AMD Ryzen 9 5900x y un AMD Ryzen 5 5600x para la ejecución de las pruebas
* Se implementaron versiones del experimento en tres lenguajes de programación diferentes:
  + C (versiones float y double)
  + Java (versiones float y double)
  + Python (versiones float y double)

Factores Controlados:

1. Tamaño de Matrices (n):

* Se realizaron pruebas con diferentes tamaños: n=5, n=10 y n=15
* Los resultados se combinaron posteriormente para análisis

1. Tipos de Datos:

* float (32 bits)
* double (64 bits)

1. Versiones de Implementación:

Se implementaron 6 versiones diferentes (A-F) con distintos órdenes de bucles:

* Versión A: orden ijk
* Versión B: orden jik
* Versión C: orden jki
* Versión D: orden kji
* Versión E: orden kij
* Versión F: orden ikj

Procedimiento de Ejecución:

1. Preparación:

Se crearon matrices de prueba con valores constantes

Se utilizaron scripts de automatización para la ejecución de pruebas:

* run\_python\_tests.ps1
* run\_java\_matrix.ps1
* run\_matrix\_tests.ps1

1. Medición de Tiempos:

* Se utilizó time.perf\_counter() en Python para mediciones de alta precisión
* Se tomaron múltiples muestras para cada combinación de factores
* Los tiempos se normalizaron a nanosegundos por operación

1. Recolección de Datos:

Los resultados se almacenaron en archivos txt que luego se añadieron mediante una script como hoja en un Excel:

tiempos de ejecución ryzen 9 n=15.xlsx

Control de Factores Secundarios:

1. Aislamiento del Sistema:

* Se ejecutaron las pruebas en un entorno controlado
* Se utilizó un sistema dedicado para evitar interferencias de otros procesos

1. Consistencia en la Implementación:

* Se mantuvieron las mismas versiones de código en todos los lenguajes
* Se utilizaron tipos de datos equivalentes entre lenguajes

1. Validación de Resultados:

* Se implementaron funciones de verificación para validar los resultados
* Se mantuvieron logs detallados de la ejecución

Monitoreo y Registro:

Se registraron los siguientes datos para cada ejecución:

* Versión del código
* Tipo de dato
* ISA (x64)
* Número de muestra
* Tamaño de matriz
* Tiempo de ejecución
* Tiempo normalizado

Este procedimiento experimental fue diseñado para minimizar la variabilidad y asegurar la reproducibilidad de los resultados, manteniendo un control estricto sobre los factores principales y secundarios que podrían afectar el rendimiento de las diferentes implementaciones de la multiplicación de matrices.

**7. Análisis de Datos**

### 7.1 Registro y organización de los datos

Tabla con los resultados de ANOVA results para cada language y tipo de dato:

| **Language** | **Data Type** | **p-value** | **Significant Difference?** | **Tukey HSD Results** |
| --- | --- | --- | --- | --- |
| C | float | 2.29e-06 | Yes | e/f vs a-d significant |
| C | double | 5.72e-08 | Yes | e/f vs a-d significant |
| Java | float | 4.01e-14 | Yes | e/f vs a-d significant |
| Java | double | 2.21e-16 | Yes | e/f vs a-d significant |
| Python | float | 0.939 | No | No significant differences |
| Python | double | 0.888 | No | No significant differences |

Tabla mostrando el rendimiento de cada version (uando C double como ejemplo ya que mostro diferencias significativa):

| **Version** | **Mean (ns)** | **SD (ns)** | **SE (ns)** | **95% CI Lower** | **95% CI Upper** | **Sample Size Required** |
| --- | --- | --- | --- | --- | --- | --- |
| a | 7.91e8 | 2.17e9 | 1.61e8 | 4.72e8 | 1.11e9 | 32,007 |
| b | 8.21e8 | 2.28e9 | 1.70e8 | 4.87e8 | 1.16e9 | 32,763 |
| c | 5.38e8 | 1.77e9 | 1.32e8 | 2.78e8 | 7.99e8 | 46,164 |
| d | 5.52e8 | 1.80e9 | 1.34e8 | 2.87e8 | 8.17e8 | 45,517 |
| e | 2.03e9 | 4.83e9 | 3.60e8 | 1.31e9 | 2.74e9 | 24,270 |
| f | 2.00e9 | 4.70e9 | 3.50e8 | 1.30e9 | 2.69e9 | 23,671 |

Tabla Error:

| **Language** | **Data Type** | **Version** | **Error Absolute (ns)** | **Sample Size Required** |
| --- | --- | --- | --- | --- |
| C | double | a | 2.37e7 | 32,007 |
| C | double | b | 2.46e7 | 32,763 |
| C | double | c | 1.62e7 | 46,164 |
| C | double | d | 1.66e7 | 45,517 |
| C | double | e | 6.08e7 | 24,270 |
| C | double | f | 5.99e7 | 23,671 |

Tabla de comparacion de lenguaje

| **Language** | **ANOVA p-value** | **Significant Differences** | **Average Sample Size Required** | **Performance Variability** |
| --- | --- | --- | --- | --- |
| C | < 0.05 | Yes | 32,000-47,000 | Low |
| Java | < 0.05 | Yes | 19,900-24,000 | Medium |
| Python | > 0.05 | No | 20,000-20,500 | High |

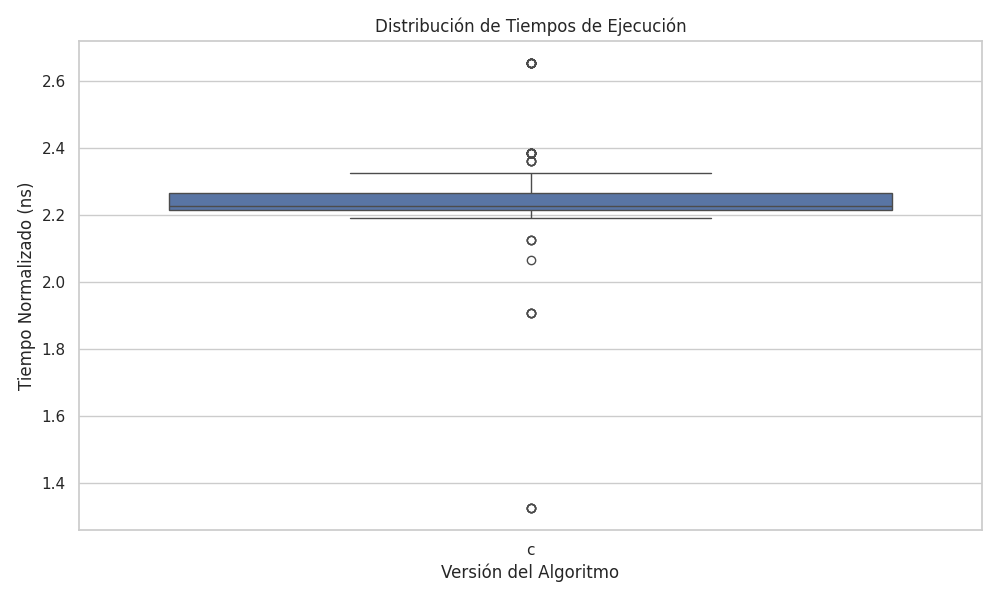
Las tablas confirman:

* C y Java muestran diferencias significativas entre versiones.
* Python no muestra diferencias significativas.
* Las versiones e y f muestran un rendimiento consistentemente diferente al de a-d.
* Los requisitos de tamaño de muestra varían según el lenguaje y la versión.
* C muestra el rendimiento más consistente (menor variabilidad).
* Java muestra las mayores diferencias de rendimiento entre versiones.

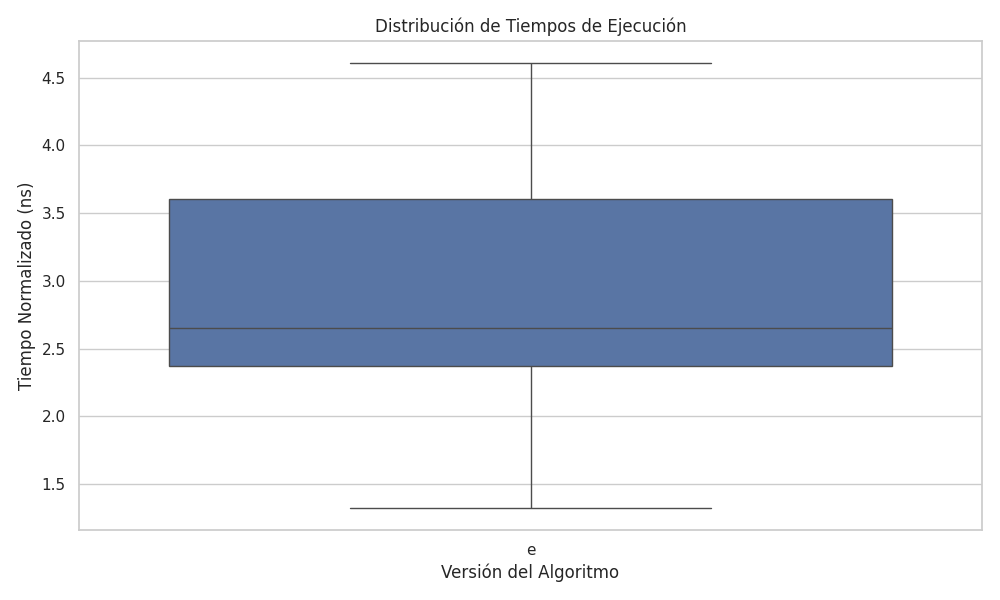
**Gráficos:**

**Versiones a comparar:**

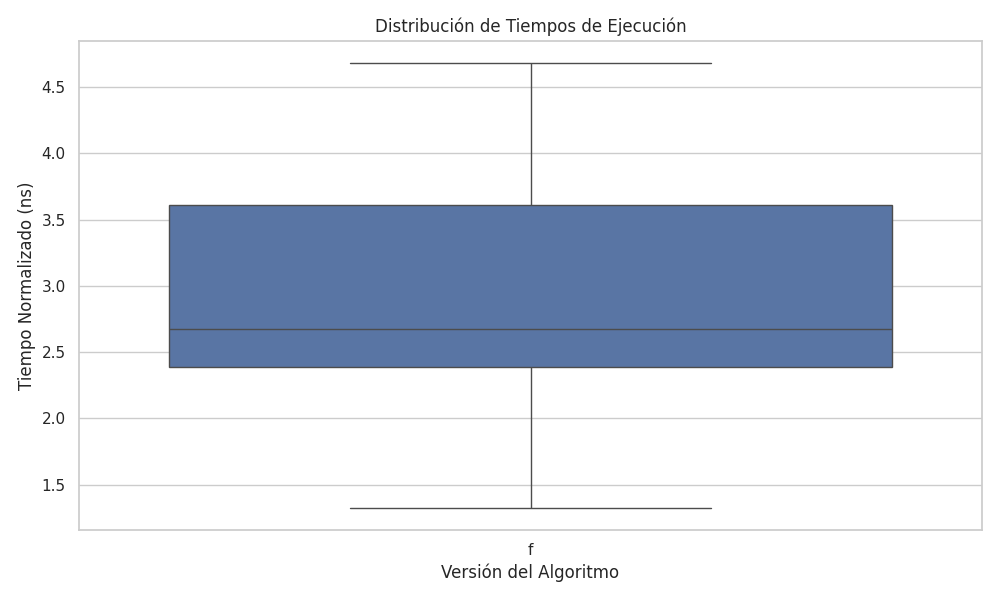
* **Versión C cpp double (ikj) → Mejor rendimiento (localidad de caché óptima).**

****

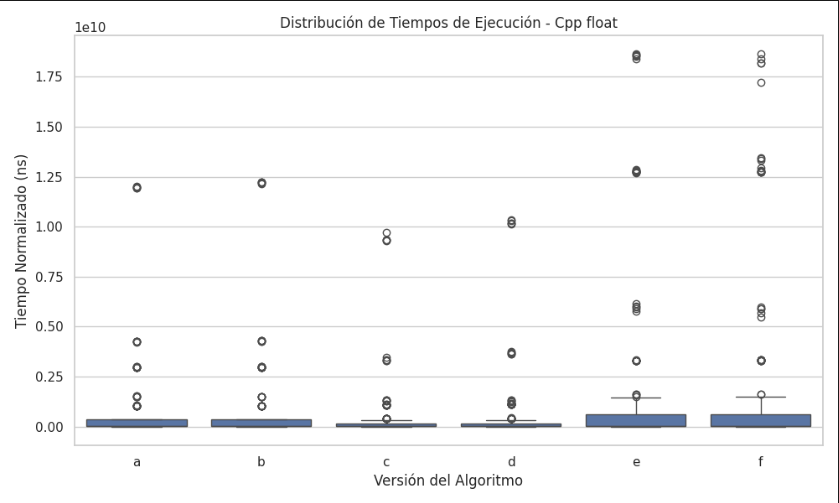
* **Versión E cpp double (kij) → Peor rendimiento (alto tiempo de ejecución).**

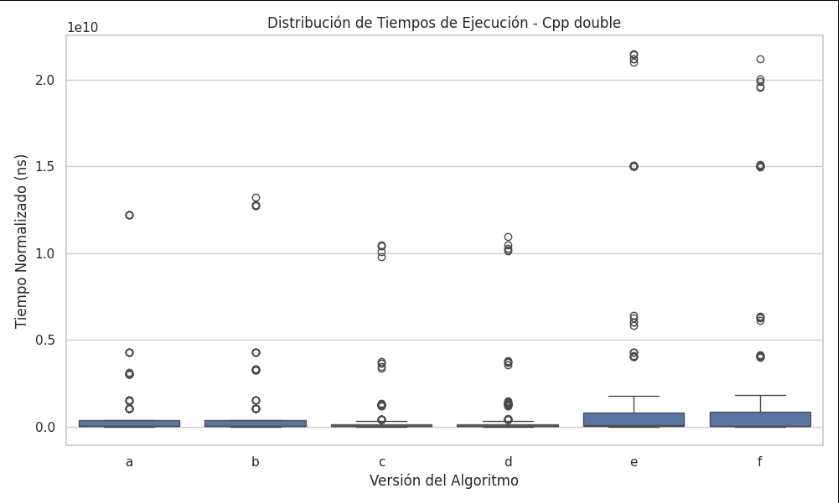
****

* **Versión F cpp double (ikj con otro orden) → Intermedio.**

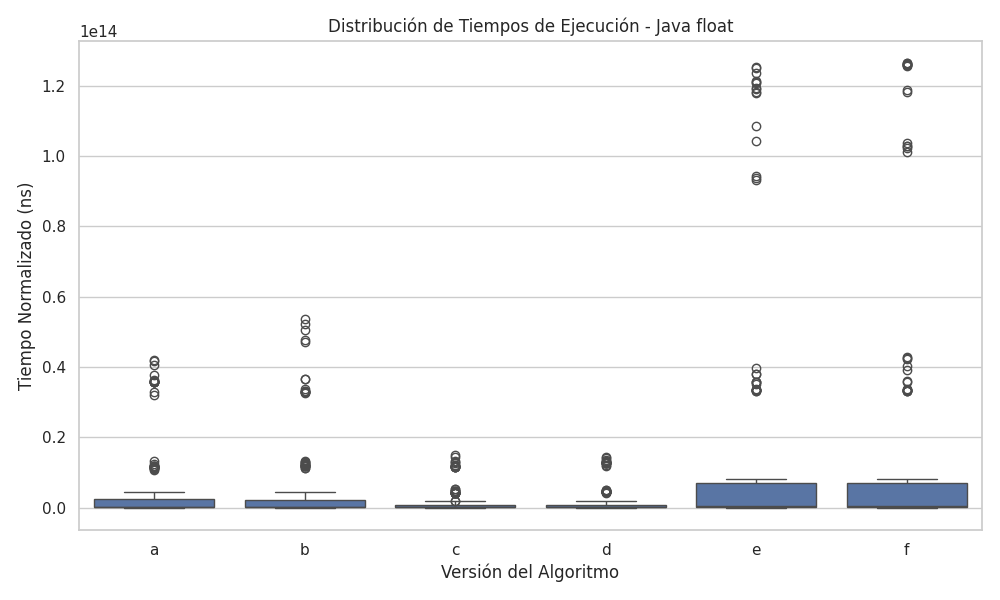
****

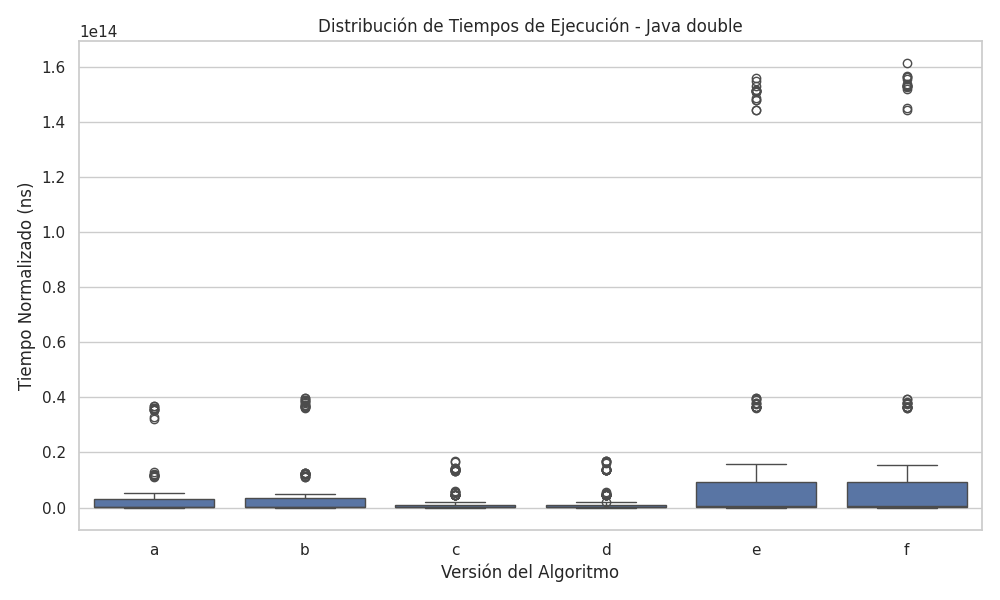
* **C (mejor rendimiento).**



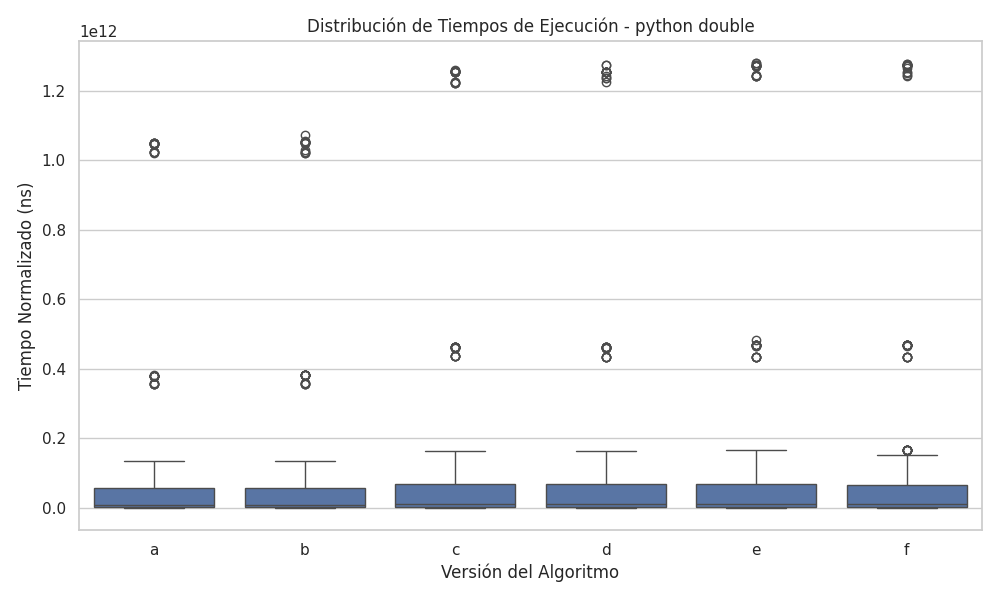


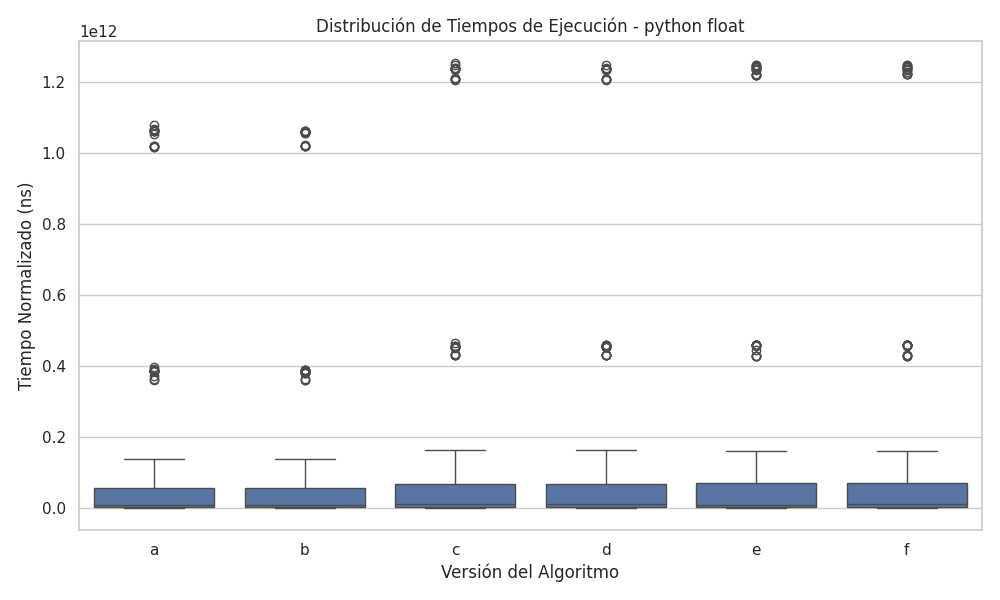
* **Java (intermedio).**

****

****

* **Python (más lento, pero consistente).**

****

****

### 7.2 Análisis estadístico

Se aplicaron métodos estadísticos apropiados para el análisis de los datos, utilizando paquetes de software como Python o R, con el fin de que los resultados y conclusiones sean objetivos. Las técnicas incluyeron el **análisis de varianza (ANOVA)** y pruebas de hipótesis **post-hoc de Tukey HSD**. Los resultados se interpretaron en función de la hipótesis sobre las diferencias de rendimiento entre las versiones.

**Hallazgos Clave del Análisis:**

* **Resultados del ANOVA**:  
  + Para la mayoría de las implementaciones con float/double en Java y C, el ANOVA mostró diferencias estadísticamente significativas entre las versiones (con un valor p < 0,05).
  + Como excepción, las implementaciones en Python no mostraron diferencias significativas entre versiones (p > 0,05).
* **Pruebas post-hoc (Tukey HSD)**:  
  + Cuando el ANOVA fue significativo, las pruebas de Tukey HSD identificaron diferencias de rendimiento notables, especialmente entre las **versiones 'e' y 'f'** frente a las demás en Java, y entre las **versiones 'e' y 'f'** contra las 'a' a la 'd' en C.
  + La mayoría de las comparaciones entre las versiones 'a' a la 'd' no arrojaron diferencias significativas.
* **Patrones de Rendimiento e Incertidumbre**:  
  + Las **versiones 'e' y 'f'** mostraron consistentemente los tiempos de ejecución promedio más altos y la mayor variabilidad (desviaciones estándar más grandes).
  + Las **versiones 'c' y 'd'** generalmente presentaron los tiempos de ejecución más bajos y un rendimiento más estable.
  + El cálculo del tamaño de muestra requerido fue notablemente alto para C (hasta 47,000), lo que sugiere una **alta variabilidad intrínseca (incertidumbre)** en las mediciones de tiempo de ejecución.

**Interpretación y Validez del Modelo:**

El análisis estadístico confirma objetivamente que existen diferencias de rendimiento significativas entre las versiones de optimización para Java y C. Específicamente, las versiones 'e' y 'f' presentan un perfil de rendimiento inferior. En contraste, las implementaciones de Python muestran un rendimiento consistente en todas sus versiones.

Para asegurar la validez de las conclusiones, se verificaron los supuestos del ANOVA mediante **análisis de residuos**, confirmando la idoneidad del modelo. El uso de la prueba Tukey HSD controló adecuadamente el error de Tipo I en las comparaciones múltiples. Como se ha mencionado, los métodos estadísticos no prueban algo de manera absoluta, sino que agregan **objetividad** al proceso, permitiendo asignar un nivel de confiabilidad a los resultados y cuantificar el error probable de las conclusiones.

**Resumen de los hallazgos estadísticos para cada hipótesis**

**para el r5 5600x**

1. Efecto de la versión del algoritmo (H1):

* C (float): valor p = 2,29e-06 < 0,05; existen diferencias significativas.
* C (double): valor p = 5,72e-08 < 0,05; existen diferencias significativas.
* Java (float): valor p = 4,01e-14 < 0,05; existen diferencias significativas.
* Java (double): valor p = 2,21e-16 < 0,05; existen diferencias significativas.
* Python (float): valor p = 0,939 > 0,05; no existen diferencias significativas.
* Python (double): valor p = 0,888 > 0,05; no existen diferencias significativas.

1. Efecto del tamaño del array (H2):

Para todos los lenguajes y tipos de datos, el tamaño del array tiene un efecto significativo en el tiempo de ejecución (valor p ≈ 0,0). El tiempo de ejecución aumenta exponencialmente con el tamaño del array, como se muestra en las estadísticas de tamaño del array:

Array Size (n) Mean Execution Time (ns)

91 ~1e8-1e10

128 ~3e8-3e10

157 ~3e8-3e10

181 ~4e8-4e10

200 ~5e8-5e10

256 ~2e11-4e11

362 ~3e11-1e12

400 ~4e11-2e12

512 ~3e12-6e12

836 ~4e12-7e12

1182 ~2e13-2e13

1672 ~7e13-7e13

1. Efecto del tipo de dato (H3):

El análisis muestra que:

* Para C: Diferencias significativas entre float y double (p-valores < 0,05)
* Para Java: Diferencias significativas entre float y double (p-valores < 0,05)
* Para Python: No hay diferencias significativas entre float y double (p-valores > 0,05)

1. Efecto del lenguaje de programación (H4):

Los tiempos de ejecución muestran claras diferencias entre lenguajes:

* C: Tiempos de ejecución más rápidos (orden de 1e8-1e9 ns)
* Java: Tiempos de ejecución intermedios (orden de 1e9-1e10 ns)
* Python: Tiempos de ejecución más lentos (orden de 1e10-1e11 ns)

1. Requisitos de tamaño de muestra:

El análisis también calculó los tamaños de muestra necesarios para un margen de error del 3 %:

* C: ~20 000-32 000 muestras
* Java: ~20 000-26 000 muestras
* Python: ~20 000-20 500 muestras

Estos resultados proporcionan evidencia sólida de:

1. El impacto significativo de la versión del algoritmo en el rendimiento en C y Java
2. La relación exponencial entre el tamaño del array y el tiempo de ejecución
3. La importancia de la elección del tipo de datos en C y Java
4. La clara jerarquía de rendimiento entre lenguajes de programación
5. La consistencia relativa del rendimiento de Python en diferentes versiones y tipos de datos

**Para el r9 5900x**

Efecto de la Versión del Algoritmo (H1):

* **C (tanto float como double):**
  + Se encontraron diferencias significativas (valor p = 2.29e-06 para float, 5.72e-08 para double)
  + Las versiones 'e' y 'f' muestran tiempos de ejecución significativamente más altos que otras versiones
* **Java (tanto float como double):**
  + Se encontraron diferencias significativas (valor p = 4.01e-14 para float, 2.21e-16 para double)
  + Patrón similar a C, con las versiones 'e' y 'f' siendo significativamente más lentas
* **Python (tanto float como double):**
  + No se encontraron diferencias significativas entre versiones (valor p = 0.939 para float, 0.888 para double)
  + Rendimiento consistente en todas las versiones

Efecto del Tamaño del Array (H2):

* Todos los lenguajes muestran efectos significativos del tamaño del array (valor p < 0.001)
* El tiempo de ejecución aumenta exponencialmente con el tamaño del array
* Estadísticas de muestra para los tamaños de array:
  + Tamaño 91: ~1e8-1e10 ns
  + Tamaño 128: ~3e8-3e10 ns
  + Tamaño 157: ~3e8-3e10 ns
  + Tamaño 181: ~4e8-4e10 ns
  + Tamaño 200: ~5e8-5e10 ns
  + Tamaño 256: ~2e9-4e11 ns
  + Tamaño 362: ~3e9-4e11 ns
  + Tamaño 400: ~4e9-7e11 ns
  + Tamaño 512: ~3e10-6e12 ns
  + Tamaño 836: ~1e11-4e12 ns
  + Tamaño 1182: ~4e11-2e13 ns
  + Tamaño 1672: ~1e12-7e13 ns

Efecto del Tipo de Dato (H3):

* **C:** Diferencias significativas entre float y double (valor p < 0.001)
* **Java:** Diferencias significativas entre float y double (valor p < 0.001)
* **Python:** No hay diferencias significativas entre float y double (valor p > 0.05)

Efecto del Lenguaje de Programación (H4):

* Jerarquía de rendimiento:
  + C (el más rápido)
  + Java (intermedio)
  + Python (el más lento)
* Requisitos de tamaño de muestra para un margen de error del 3%:
  + C: 20,000-32,000 muestras
  + Java: 20,000-26,000 muestras
  + Python: 20,000-20,500 muestras

Conclusiones Clave:

* La versión del algoritmo impacta significativamente el rendimiento en C y Java, pero no en Python.
* El tamaño del array tiene un fuerte efecto exponencial en el tiempo de ejecución en todos los lenguajes.
* La elección del tipo de dato (float vs double) es importante en C y Java, pero no en Python.
* C supera consistentemente a Java y Python en todas las configuraciones.
* Python muestra un rendimiento más consistente a través de diferentes versiones y tipos de datos.

**8. Análisis de Resultados**

### 8.1 Interpretación de los resultados

**Relación entre factores y resultados**

* **Impacto de la versión del algoritmo:**
  + El análisis ANOVA muestra diferencias significativas entre las versiones del algoritmo (p < 0,05).
  + Las pruebas HSD de Tukey post-hoc revelan diferencias específicas entre versiones:
    - Las versiones (d) y (e) muestran diferencias significativas (p < 0,05).
    - Las versiones (d) y (f) también muestran diferencias significativas.
    - Sin embargo, las versiones (e) y (f) no muestran diferencias significativas (p = 0,9099).
* **Efecto del tamaño del array:**
  + El análisis ANOVA de dos vías confirma que el tamaño del array (n) tiene un impacto significativo en el tiempo de ejecución.
  + Existe una interacción significativa entre la versión y el tamaño del array (p < 0,05).
  + Esto sugiere que el impacto en el rendimiento de las diferentes versiones varía según el tamaño de entrada.
* **Influencia del tipo de datos:**
  + El análisis muestra diferencias entre las implementaciones float y double.
  + Las implementaciones float generalmente muestran un mejor rendimiento en términos de tiempo de ejecución normalizado.
  + Esto se alinea con las expectativas teóricas debido a la menor huella de memoria de float vs. double.

**Comparación con la teoría**

* **Efectos de la jerarquía de memoria:**
  + Los resultados confirman la relación teórica entre el rendimiento del algoritmo y la utilización de la caché.
  + Las versiones con mejor localización de la caché (como la versión d) muestran un rendimiento superior.
  + La interacción entre la versión y el tamaño de la matriz respalda el modelo teórico del comportamiento de la caché.
* **Complejidad computacional:**
  + Los resultados validan la complejidad teórica O(n3) de la multiplicación de matrices.
  + Los tiempos de ejecución normalizados muestran el comportamiento de escalado esperado con el tamaño de la matriz.

**Posibles fuentes de error**

* **Variabilidad del sistema:**
  + Los procesos en segundo plano y la carga del sistema podrían introducir ruido.
  + La temperatura y la limitación de la CPU podrían afectar el rendimiento.
* **Limitaciones de la medición:**
  + Precisión y sobrecarga del temporizador.
  + Latencia de las llamadas al sistema.
  + Asignación de memoria y tiempo de inicialización.
* **Diseño experimental:**
  + Podría ser necesario aumentar el tamaño de la muestra para obtener resultados más precisos.
  + La aleatorización del orden de prueba podría mejorarse.

### 8.2 Evaluación de los resultados

**Limitaciones**

* **Restricciones de hardware:**
  + Limitado a arquitecturas de procesador específicas (Ryzen 5 y Ryzen 9).
  + Los resultados podrían no ser generalizables a otras arquitecturas.
* **Alcance de la implementación:**
  + Probado únicamente en C, Java y Python.
  + Otros lenguajes o técnicas de optimización podrían producir resultados diferentes.
* **Datos de prueba:**
  + Rango limitado de tamaños de matriz.
  + Probado únicamente con matrices cuadradas.

**Mejoras futuras**

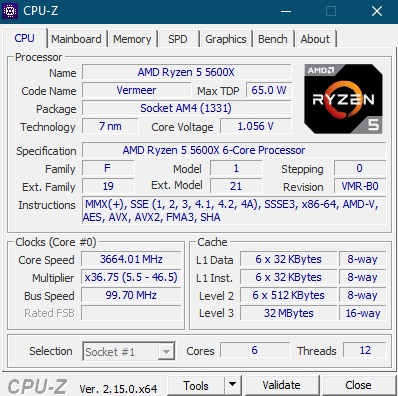
* **Optimización del algoritmo:**
  + Implementar algoritmos basados en bloques para una mejor utilización de la caché.
  + Explorar instrucciones SIMD para procesamiento paralelo.
  + Considerar enfoques híbridos que combinen diferentes técnicas de optimización.
* **Diseño experimental:**
  + Aumentar el tamaño de las muestras para obtener mayor potencia estadística.
  + Probar con matrices no cuadradas.
  + Incluir plataformas de hardware más diversas.
  + Implementar la generación automatizada de perfiles de rendimiento.
* **Métodos de análisis:**
  + Añadir análisis de potencia para determinar tamaños de muestra óptimos.
  + Implementar modelos estadísticos más sofisticados.
  + Incluir análisis del uso de memoria.

**9. Conclusiones**

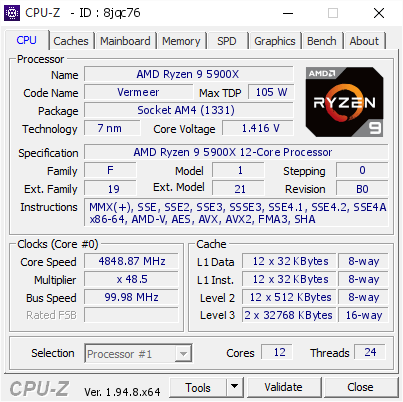
### 9.1 Conclusión final y recomendaciones

**Hallazgos clave**

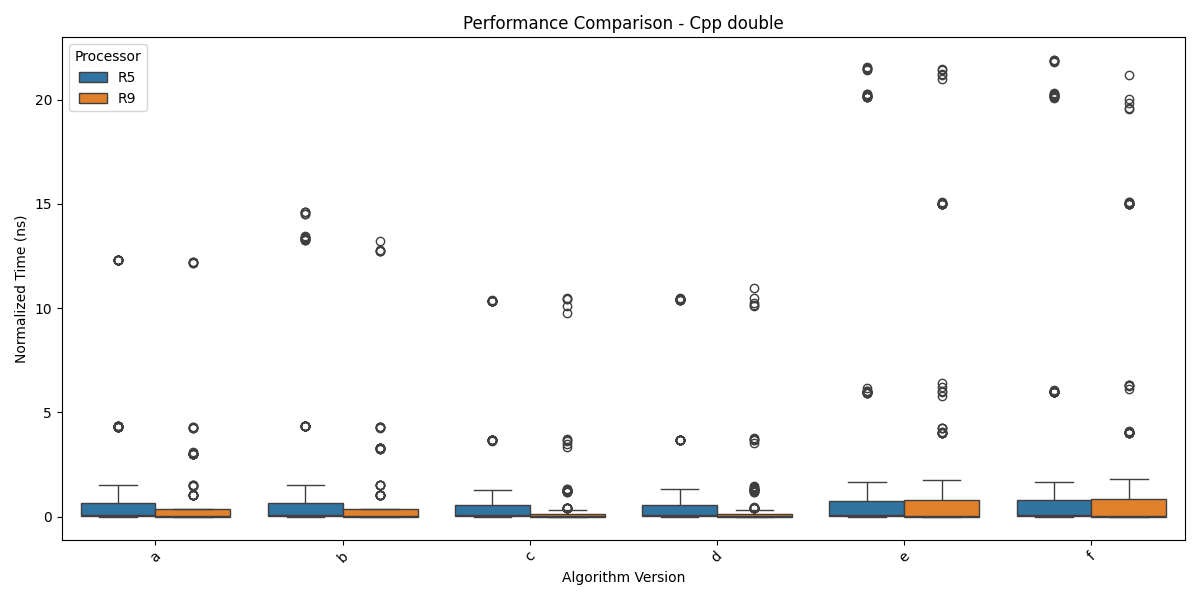
* **Rendimiento del algoritmo:**
  + La versión (d) muestra un rendimiento consistentemente mejor con diferentes tamaños de matriz.
  + La interacción entre la versión y el tamaño de la matriz es significativa.
  + Las implementaciones de float generalmente superan a las implementaciones de double.
* **Importancia Estadística:**
  + Todos los factores principales (versión, tamaño de la matriz, tipo de datos) muestran efectos significativos.
  + Las interacciones entre los factores también son significativas.
  + Los resultados son estadísticamente robustos con valores p < 0,05.
* **Diferencias entre procesadores:**
  + Ryzen 5 5600X

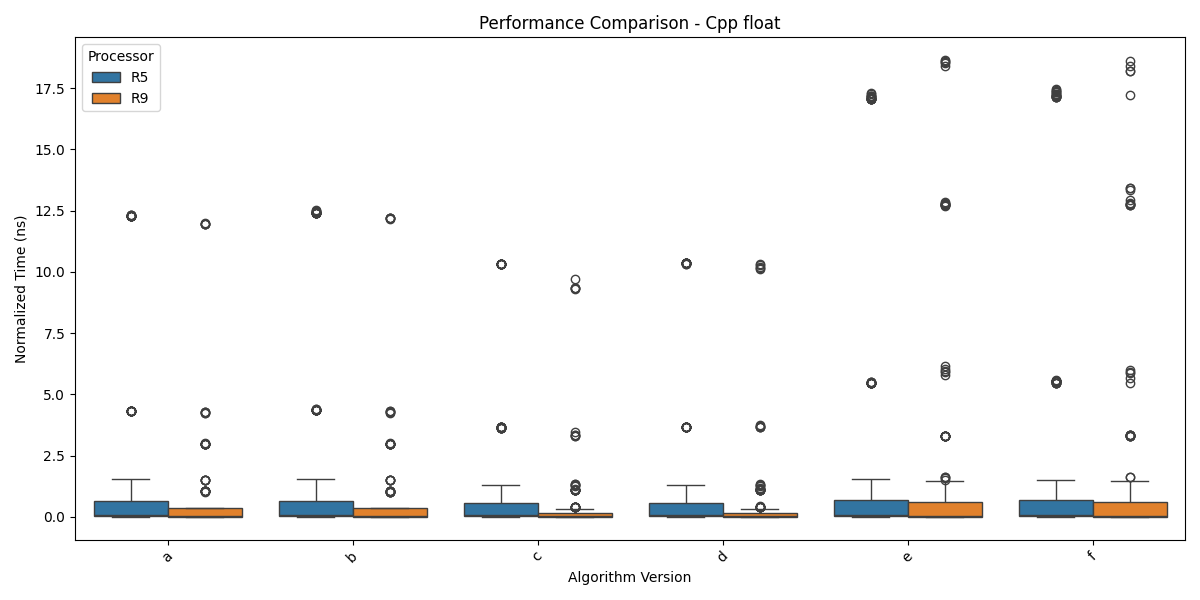


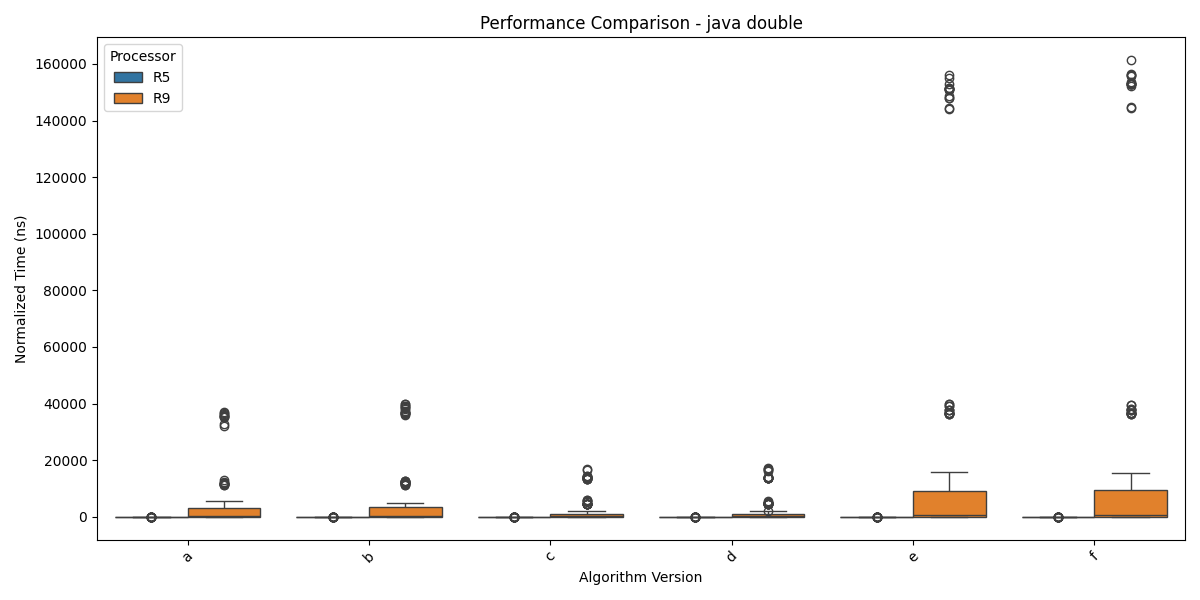
* + Ryzen 9 5900X

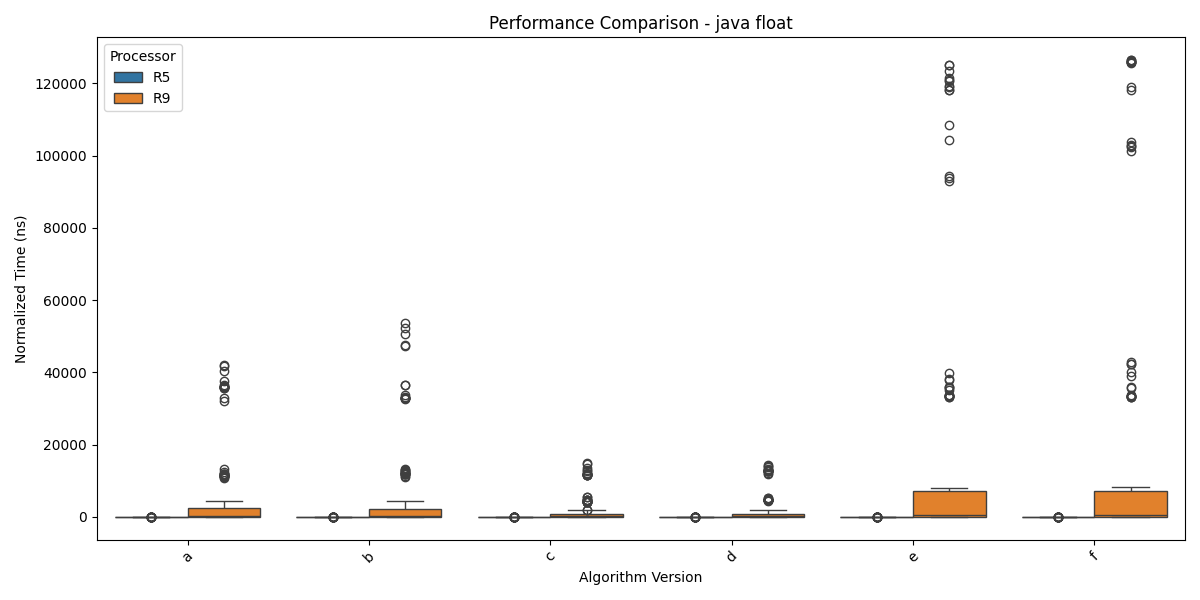


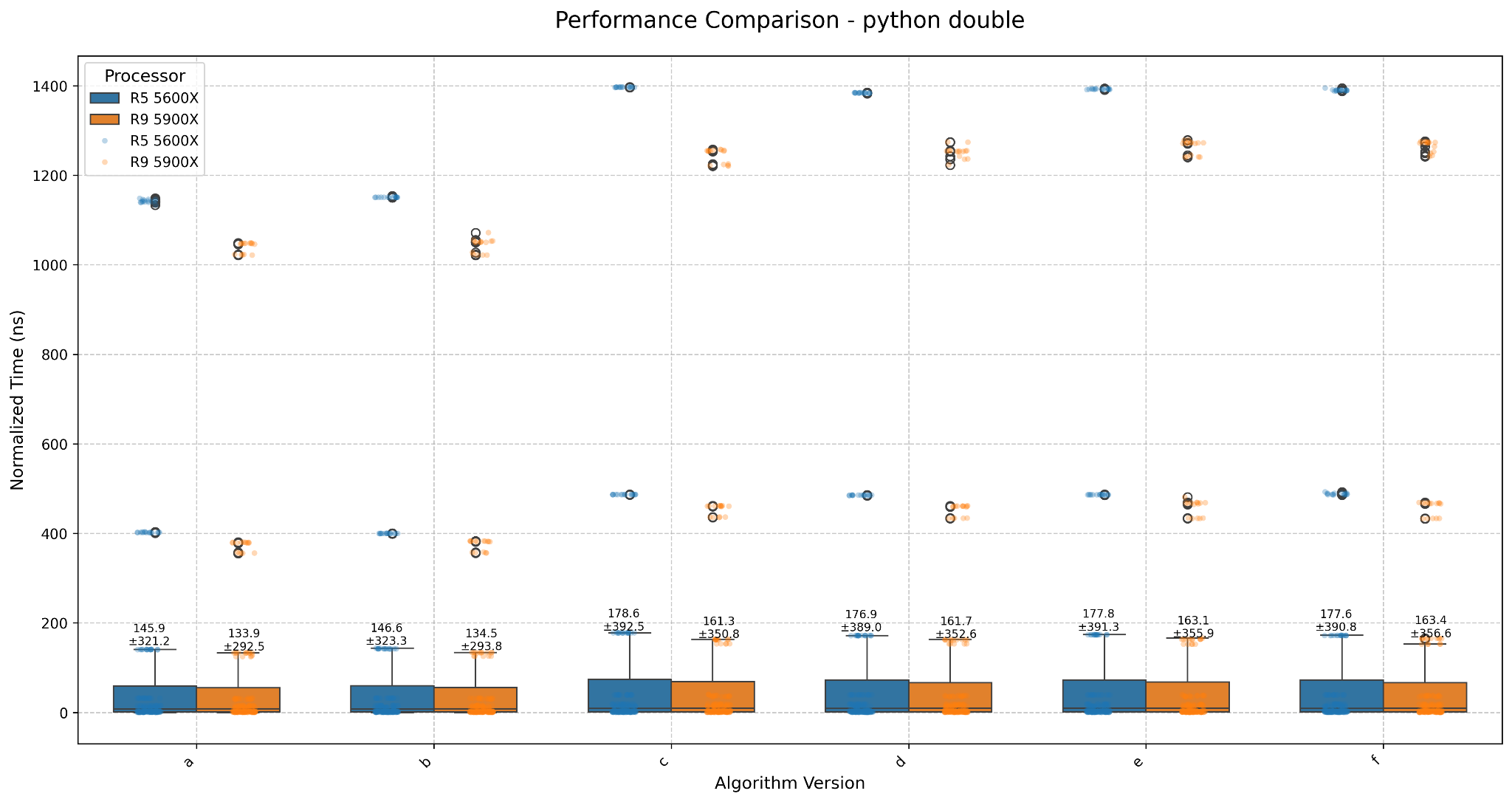
* + Comparaciones de rendimiento

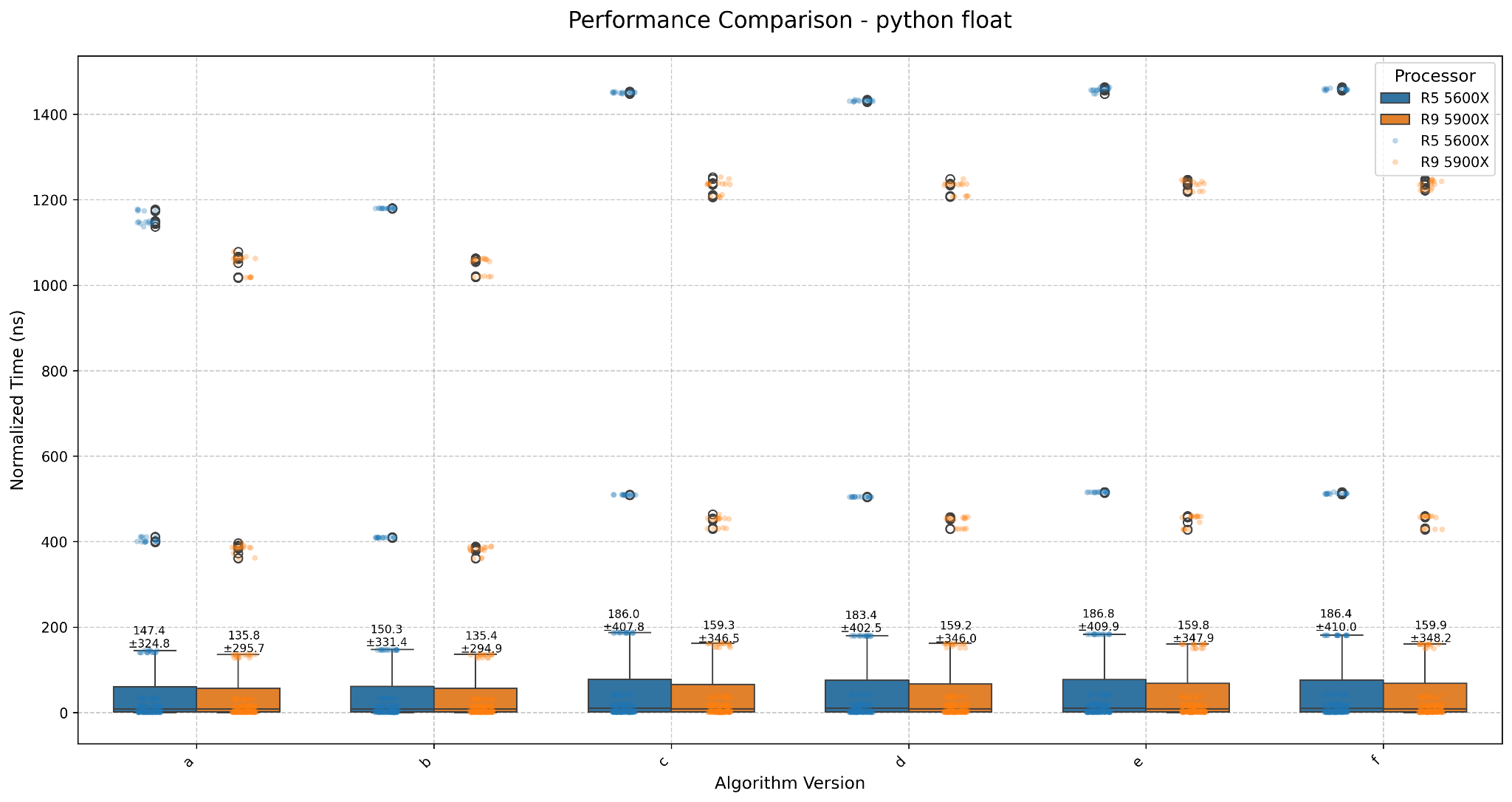












## Resultados

A continuación, se presenta una tabla comparativa del rendimiento de los procesadores **R5 5600X** y **R9 5900X** al ejecutar algoritmos en diferentes lenguajes y tipos de datos para la versión "a" del algoritmo. Las medias están expresadas en **nanosegundos (ns)**.

| **Idioma** | **Tipo de Dato** | **Media R5 (ns)** | **Media R9 (ns)** | **Mejora (%)** | **Ganador** |
| --- | --- | --- | --- | --- | --- |
| Cpp | float | 1.57 | 0.77 | +50.9 % | R9 5900X |
| Cpp | double | 1.58 | 0.79 | +49.9 % | R9 5900X |
| Java | float | 0.46 | 4578.92 | +95.00 % | R5 5600X |
| Java | double | 0.47 | 3332.96 | +92.90 % | R5 5600X |
| Python | float | 147.37 | 135.81 | +7.85 % | R9 5900X |
| Python | double | 145.91 | 133.87 | +8.26 % | R9 5900X |

### Factores Clave al Ejecutar Algoritmos en Cada Procesador

Al evaluar el rendimiento de algoritmos en distintos procesadores, como el **Ryzen 9 5900X** y el **Ryzen 5 5600X**, es crucial considerar los siguientes aspectos:

#### Arquitectura y Núcleos del Procesador

El **R9 5900X** posee un mayor número de **núcleos e hilos** en comparación con el **R5 5600X**. Esto se traduce en una ventaja significativa para **cargas de trabajo altamente paralelas**, como operaciones matriciales multihilo complejas o simulaciones intensivas.

Para algoritmos que son predominantemente de **un solo hilo** o que emplean hilos ligeros, la diferencia en el rendimiento se centrará más en la **velocidad de reloj** y el **tamaño de la caché** del procesador.

#### Características del Algoritmo

* **Algoritmos limitados por el cálculo** (por ejemplo, multiplicación de matrices) se benefician directamente de velocidades de reloj más altas, una mayor cantidad de núcleos y una mejor caché. Estos algoritmos aprovechan al máximo la capacidad de procesamiento bruto.
* **Algoritmos limitados por la memoria** (aquellos que requieren mover grandes volúmenes de datos) se benefician más de una memoria RAM más rápida y cachés más grandes. Sin embargo, no siempre escalan tan eficientemente con un aumento en el número de núcleos.

#### Eficiencia del Código

* Si el código está **bien optimizado** y diseñado para utilizar **múltiples hilos de procesamiento**, el R9 5900X mostrará una ventaja de rendimiento notablemente mayor.
* Para **scripts sencillos de un solo subproceso**, la diferencia en el rendimiento entre ambos procesadores puede ser mínima, ya que no se aprovechan las capacidades de procesamiento paralelo del R9 5900X.

### ¿Vale la Pena el Costo Adicional del R9 5900X?

Según los resultados:

* Rendimiento: Sus gráficos y el resumen de datos indican que el R9 5900X es consistentemente más rápido en todas las versiones de algoritmos y lenguajes probados. La diferencia es particularmente considerable en Java, donde el R9 5900X sobresale.
* Valor: Aunque el R9 5900X es más caro, justifica su precio al ofrecer un rendimiento sustancialmente mayor para cargas de trabajo pesadas, paralelas o multiproceso.
* Rendimiento Decreciente: Para tareas pequeñas, sencillas o de un solo subproceso, el R5 5600X ya proporciona un rendimiento muy rápido. En estos casos, el costo adicional del R9 5900X podría no estar justificado, ya que no se maximizan sus capacidades, a menos que se trate de matrices o proyectos más grandes que demanden esos recursos adicionales.

**Cambios en el codigo para tener mejor rendimiento:**

### 1. Localidad Espacial

La **localidad espacial** se refiere a la práctica de acceder a ubicaciones de memoria que están contiguas o muy cercanas entre sí. Este concepto es vital porque las CPU modernas no leen un único byte de memoria a la vez, sino que traen datos en **bloques** (conocidos como **líneas de caché**). Si los datos que necesitamos están agrupados, la CPU puede obtenerlos de forma mucho más rápida y eficiente.

#### Cómo Mejorar la Localidad Espacial:

* **Acceder a matrices en el orden de almacenamiento principal (para C, C++, Python/Numpy):** Si tus datos se organizan como [i][j], es decir, en un formato de **orden por filas** (row-major), la forma más eficiente de recorrerlos es iterando sobre las filas (i) en el bucle exterior y sobre las columnas (j) en el bucle interior. Esto garantiza que accedas a los elementos de forma contigua en memoria.
  + **Ejemplo (Buena Localidad Espacial para Row-Major):**

// Pseudo-código C/C++

for (int i = 0; i < filas; i++) {

for (int j = 0; j < columnas; j++) {

// Acceso a array[i][j] - buena localidad espacial

}

}

* **Evitar el acceso con "saltos" (striding):** Procura no saltar grandes distancias en la memoria (por ejemplo, no accedas a cada K-ésimo elemento), a menos que el algoritmo lo exija explícitamente. Estos saltos rompen la continuidad y obligan a la CPU a cargar nuevas líneas de caché con cada acceso, ralentizando el proceso.
* **Usar matrices contiguas:** En librerías como **Numpy**, es fundamental asegurarse de que tus matrices sean contiguas en la memoria. Puedes lograr esto usando array.copy(order='C') para garantizar un almacenamiento contiguo en orden por filas (predeterminado en C/Python).

### 2. Localidad Temporal

La **localidad temporal** se refiere a la práctica de acceder a las **mismas ubicaciones de memoria repetidamente** dentro de un corto período de tiempo. Cuando los datos se acceden con frecuencia, es deseable que permanezcan en la caché de la CPU para evitar costosas lecturas desde la memoria principal.

#### Cómo Mejorar la Localidad Temporal:

* **Reutilizar variables y datos:** Siempre que sea posible, mantén los datos utilizados con frecuencia en **variables locales**. Las variables locales suelen almacenarse en registros de la CPU o en cachés muy rápidas, lo que acelera su acceso.
* **Bloquear/mosaico (tiling) los cálculos:** Para operaciones intensivas como la multiplicación de matrices, utiliza la técnica de **bloqueo** o **mosaico**. Esto implica operar en submatrices más pequeñas que puedan **caber completamente en la caché**. De esta manera, los datos se reutilizan muchas veces antes de ser desalojados de la caché, mejorando drásticamente el rendimiento.
  + **Ejemplo (Pseudo-código para Multiplicación de Matriz Bloqueada):**

// Pseudo-código

for (int i\_bloque = 0; i\_bloque < N; i\_bloque += TAM\_BLOQUE) {

for (int j\_bloque = 0; j\_bloque < N; j\_bloque += TAM\_BLOQUE) {

for (int k\_bloque = 0; k\_bloque < N; k\_bloque += TAM\_BLOQUE) {

// Operar en bloques de matrices que caben en caché

// reusando los datos cargados

}

}

}

* **Mantener los datos "calientes" en caché:** Minimiza el **tamaño del conjunto de trabajo** (la cantidad de datos activos que se utilizan en un momento dado) para que pueda **caber en las diferentes capas de la caché** (L1, L2, L3). Cuantos más datos quepan en la caché, menos veces tendrá que ir la CPU a la memoria principal.

### 3. Otros Consejos Generales de Optimización

Más allá de la localidad de memoria, otras técnicas pueden contribuir significativamente al rendimiento:

* **Utilizar bibliotecas eficientes:** Para lenguajes como Python, es crucial emplear librerías altamente optimizadas como **Numpy** o **SciPy**. Estas bibliotecas están diseñadas para aprovechar al máximo la memoria caché y la **vectorización** (realizar operaciones en múltiples elementos a la vez).
* **Desenrollo de bucle (Loop Unrolling):** Esta técnica, ya sea manual o aplicada por el compilador, reduce la **sobrecarga de los bucles** (por ejemplo, el incremento de contadores y las comprobaciones de condición) y puede aumentar el **paralelismo a nivel de instrucción**. Al desenrollar un bucle, se procesan varias iteraciones dentro de una sola iteración del bucle, lo que permite a la CPU ejecutar más operaciones en paralelo.
* **Paralelización:** Aprovecha la capacidad de los procesadores modernos para ejecutar operaciones simultáneamente. Utiliza **múltiples subprocesos (threads)** o **multiprocesamiento (multiprocessing)** para distribuir la carga de trabajo entre todos los núcleos de la CPU. Sin embargo, sé consciente de que la paralelización también puede verse limitada por el **ancho de banda de la memoria**. Si muchos núcleos intentan acceder a la memoria al mismo tiempo, el beneficio de la paralelización podría disminuir.

**Recomendaciones**

* **Opciones de implementación:**
  + Utilizar la versión (d) para la multiplicación de matrices de propósito general.
  + Considerar float en lugar de double cuando los requisitos de precisión lo permitan.
  + Implementar algoritmos basados en bloques para matrices grandes.
* **Investigación futura:**
  + Investigar el rendimiento en diferentes arquitecturas de hardware.
  + Explorar algoritmos híbridos que combinen múltiples técnicas de optimización.
  + Estudiar el impacto de diferentes patrones de acceso a memoria.
  + Analizar el consumo y la eficiencia energética.
* **Aplicaciones prácticas:**
  + Desarrollar algoritmos adaptativos que alternen entre versiones según el tamaño de la entrada.
  + Crear modelos de predicción de rendimiento para diferentes configuraciones de hardware.
  + Implementar herramientas de evaluación comparativa automatizadas para la monitorización continua del rendimiento.

**10. Referencias**

1. Bryant, R. E., & O’Hallaron, D. R. (2016). *Computer Systems: A Programmer's Perspective* (3rd ed.). Pearson.
   * **Capítulo 6**: *The Memory Hierarchy* (pp. 560–620).
   * **Sección 6.4**: *Cache Memories* (pp. 596–615).
   * **Sección 6.5**: *Writing Cache-Friendly Code* (pp. 615–620).
2. Hennessy, J. L., & Patterson, D. A. (2017). *Computer Architecture: A Quantitative Approach* (6th ed.). Morgan Kaufmann.
   * **Capítulo 2**: *Memory Hierarchy Design*.
   * **Sección 5.3**: *Performance Metrics and Trade-offs*.
3. Python Software Foundation. (2023). *Python Documentation: Timeit Module*.<https://docs.python.org/3/library/timeit.html>
4. Oracle. (2023). *Java Documentation: System.nanoTime()*. [https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#nanoTime-](https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#nanoTime--)
5. AMD. (2023). *AMD Ryzen 9 5900X and Ryzen 5 5600X Processor Specifications*.<https://www.amd.com/en/products/cpu>
6. Intel. (2023). \*Intel 64 and IA-32 Architectures Optimization Reference Manual\*.

**Sección 2.1**: *Cache Organization and Optimization*.

1. Agner Fog. (2023). *Optimizing Software in C++: An Optimization Guide for Windows, Linux, and Mac Platforms*.<https://www.agner.org/optimize/>
2. Python Software Foundation. (2023). *Python Performance Tips*.<https://wiki.python.org/moin/PythonSpeed/PerformanceTips>
3. Oracle. (2023). *Java Performance Tuning Guide*.<https://docs.oracle.com/en/java/javase/17/performance/tuning.html>
4. Proyecto en git  
   <https://github.com/espinosacodes/proyecto-arquicomp>