Reporte técnico: Proyecto final de Sistemas Operativos y Laboratorio

1. Información del Proyecto

- **Título del Proyecto:** Inversión de matrices de gran tamaño
- **Curso/Materia:** Sistemas Operativos
- **Integrantes:** Cristian Daniel Muñoz Botero, Jonathan Andrés Granda Orrego, Brandon Duque García, Jhon Alexander Botero Gomez.
- **Fecha de Entrega:** 10 de Julio de 2025

2. Introducción

2.1. Objetivo del Proyecto

Describa claramente el objetivo principal de tu proyecto. ¿Qué problema busca resolver o qué funcionalidad implementa? ¿Qué se espera demostrar con este proyecto?

Analizar y comparar algoritmos existentes para la inversión de matrices de gran tamaño, evaluando su eficiencia y escalabilidad en distintos entornos de hardware(GPU y CPU). El proyecto busca abordar el problema del alto costo computacional asociado a esta operación en aplicaciones de gran escala, identificando cuáles algoritmos hacen un uso más eficiente de los recursos disponibles (CPU, memoria, paralelismo, etc.). Se espera demostrar qué métodos ofrecen un mejor desempeño en términos de tiempo de ejecución, consumo de memoria y adaptabilidad a diferentes arquitecturas, con el fin de orientar las aplicaciones en contextos de procesamiento intensivo como lo es la inteligencia artificial.

2.2. Motivación y Justificación

Explique por qué eligió este proyecto o tema. ¿Cuál es su relevancia en el contexto de los sistemas operativos? ¿Por qué es interesante o importante?

Este proyecto fue elegido por su estrecha relación con conceptos fundamentales del curso de sistemas operativos, tanto en el ámbito teórico como en el práctico. La inversión de matrices de gran tamaño representa un desafío computacional que demanda un uso intensivo y eficiente de los recursos del sistema, lo cual permite poner en práctica conocimientos clave del funcionamiento interno de un sistema operativo y elementos de hardware.

En primer lugar, este problema exige una gestión cuidadosa de la memoria RAM, ya que implica trabajar con grandes volúmenes de datos que deben almacenarse y manipularse sin saturar el sistema, y que sea de manera eficiente. Además, los algoritmos utilizados suelen apoyarse en bibliotecas numéricas optimizadas, las cuales aprovechan al máximo las capacidades de la arquitectura del hardware, reflejando cómo el sistema operativo gestiona el acceso eficiente a dichos recursos.
Por otro lado, el proyecto involucra directamente la gestión de procesos e hilos, ya que los algoritmos de inversión pueden beneficiarse significativamente del paralelismo, estrategia sumamente importante en este caso particular de la inversión de matrices. Esto permite estudiar cómo los sistemas operativos asignan tareas a múltiples núcleos, sincronizan hilos y equilibran la carga de trabajo entre ellos. De esta forma, el análisis del desempeño de estos algoritmos también permite comprender cómo influyen los mecanismos de planificación y concurrencia en la eficiencia computacional.
Finalmente, este proyecto es interesante porque conecta un problema matemático complejo con aspectos prácticos de la computación a bajo nivel, permitiendo explorar cómo los sistemas operativos pueden facilitar o limitar el rendimiento en escenarios de alta demanda. Además, ofrece la oportunidad de experimentar con técnicas de computación paralela, cada vez más relevantes en la era del procesamiento masivo de datos.

2.3. Alcance del Proyecto

Defina los límites del proyecto. ¿Qué funcionalidades o aspectos específicos aborda? ¿Qué queda fuera del alcance de esta implementación?

Este proyecto, en primer lugar, aborda el concepto de paralelismo dado que implementamos un algoritmo que invierte una matriz calculando fila por fila usando la matriz original. Esto permite que se pueda crear un hilo para cada fila, optimizando el tiempo computacional que se demora en realizar esta operación.

El concepto de paralelismo se podrá ver reflejado en el algoritmo de inversión de matrices LU, el cual implementamos sin paralelismo y con paralelismo para comparar los resultados. Adicionalmente haremos uso de CUDA con el objetivo de explorar su potencial específicamente para los algoritmos en Python.

3. Marco Teórico / Conceptos Fundamentales

Explique brevemente los conceptos relevantes relacionados con el proyecto (Tecnologías empleadas, para que se emplean, etc)

Este proyecto se enfoca en el análisis y comparación de algoritmos para la inversión de matrices de gran tamaño, lo cual requiere el entendimiento de varios conceptos técnicos fundamentales relacionados con la computación científica, la programación paralela y los sistemas operativos.

Lenguaje C: El lenguaje C es fundamental en el desarrollo de aplicaciones científicas y de alto rendimiento, ya que permite el manejo directo de memoria, punteros y estructuras de datos complejas. Estas características lo hacen ideal para implementar algoritmos numéricos como la inversión de matrices, especialmente en casos donde hay que controlar a detalle la CPU, los hilos de ejecución y la eficiencia computacional. Además C es compatible con bibliotecas de computación paralela como OpenMP y MPI que permiten distribuir tareas entre múltiples núcleos o incluso múltiples máquinas.

Inversión de matrices: La inversión de una matriz busca obtener otra matriz que, al multiplicarse por la original, produzca la matriz identidad. Este proceso es esencial en áreas como la resolución de sistemas lineales, aprendizaje automático, simulaciones físicas y análisis estadístico. Sin embargo, cuando se trata de matrices de gran tamaño, la operación se vuelve intensiva en cuanto a recursos computacionales, y resulta muy costosa.

CUDA (Compute Unified Device Architecture):CUDA es una plataforma de computación paralela desarrollada por NVIDIA que permite utilizar la GPU (Unidad de Procesamiento Gráfico) para tareas de propósito general. A diferencia de la CPU, que está optimizada para tareas secuenciales, la GPU cuenta con miles de núcleos capaces de ejecutar operaciones en paralelo, lo que la convierte en una herramienta poderosa para la inversión de matrices y otras operaciones numéricas pesadas.

CuPy: CuPy es una biblioteca de Python que proporciona una interfaz similar a NumPy, pero ejecuta los cálculos en la GPU utilizando CUDA. Es altamente eficiente para operaciones matriciales y vectoriales, lo que la hace ideal para trabajar con grandes volúmenes de datos y acelerar significativamente los tiempos de cómputo en comparación con implementaciones en CPU.

PyTorch: PyTorch es un framework de código abierto ampliamente utilizado en machine learning y deep learning, que también permite realizar operaciones numéricas avanzadas sobre tensores. PyTorch puede aprovechar la GPU mediante CUDA, lo que permite realizar operaciones matriciales (como la inversión) de forma paralela y acelerada. Además, su flexibilidad lo hace útil no solo en entrenamiento de modelos, sino también en tareas de computación científica general.

Paralelismo y gestión de hilos: Los algoritmos para invertir matrices grandes pueden beneficiarse del paralelismo, es decir, dividir los cálculos en múltiples tareas que se ejecutan simultáneamente. PyTorch y CuPy abstraen gran parte de esta gestión de hilos, pero es importante comprender cómo el sistema operativo distribuye estas tareas entre núcleos de CPU y GPU, cómo sincroniza procesos concurrentes y cómo maneja la memoria compartida.

Gestión de memoria: Las operaciones sobre matrices grandes requieren un manejo eficiente de la memoria, tanto en la CPU como en la GPU. CUDA gestiona memoria global, compartida y local, y tanto CuPy como PyTorch permiten controlar explícitamente la transferencia de datos entre CPU y GPU, lo cual es clave para evitar cuellos de botella.

4. Diseño e Implementación

4.1. Diseño de la Solución

Describa cómo diseñó su proyecto. Incluya diagramas (de bloques, de flujo, de estados) para ilustrar la arquitectura o el funcionamiento interno. Explique las decisiones de diseño clave que tomo y por qué.

El diseño del proyecto se basó en una estructura modular que permite comparar el rendimiento de distintos algoritmos de inversión de matrices, se tomaron métricas temporales para ver la comparación de estos en distintos entornos de computación, implementados tanto en C como en Python, y ejecutados con y sin técnicas de paralelismo. El objetivo es evaluar la eficiencia y escalabilidad de cada enfoque, considerando el uso de recursos del sistema (CPU y Memoria) y el tiempo de ejecución .Además de realizar comparaciones y acercamientos al procesamiento con GPU, para realizar comparativas con núcleos de procesamiento gráfico.

Diseño de algoritmos

Diagramas de flujo:

Para los 3 algoritmos se realizaron los respectivos diagramas de flujo con los que se implementaron los códigos en C, lenguaje con el que se puede alcanzar mayor nivel de explicabilidad al momento de leer el algoritmo debido a su bajo nivel. Los diagramas de flujo de cada uno de los 3 algoritmos se encuentran en el repositorio del proyecto, en la carpeta 'diagramas-flujo' y se pueden observar en el siguiente link:

- Diagrama de flujo - Inversión con LU
- Diagrama de flujo - Inversión con reducción gaussiana
- Diagrama de flujo - Inversión con descomposición de Cholesky

Seguido de esto, se comenzó una búsqueda de las partes paralelizables de cada algoritmo. Buscando la manera de implementar varios hilos que calcularán la inversa de alguna parte de la matriz, ya sea un bloque, ya sea una fila o columna sin que haya problemas con cálculos anteriores o condiciones de carrera.

Para el algoritmo de inversión con descomposición LU se pudo observar que calculaba la inversa por columnas. Es decir, descomponía la matriz original en L y U y calculaba con esto la inversa de alguna columna de la matriz. Esto nos permitió realizar en C la implementación del paralelismo. Se crean hilos, los cuales cada uno calcula una columna de la matriz según haya disponibilidad. Haciendo que se calculen columnas de la matriz inversa al mismo tiempo.

Para el algoritmo de inversión de matrices con reducción gaussiana podemos paralelizar la parte verde del diagrama de flujo de este algoritmo. El ciclo donde se calcula la inversa podemos ver que itera sobre filas. Se calculan filas diferentes por lo que no se producen condiciones de carrera y se mejora el rendimiento del algoritmo.

Para el algoritmo de inversión de matrices con descomposición de Cholesky pudimos observar que la parte paralelizable es el ciclo principal. La región morada del diagrama de flujo para este algoritmo. Observamos que por cada iteración que se realiza en el ciclo principal se calcula una columna de la inversa, por lo que cada hilo podría calcular una columna de la inversa al mismo tiempo que se calculan otras columnas.

Implementación de algoritmos

- En **C y Python**, se implementaron algoritmos clásicos como la eliminación de Gauss-Jordan, la descomposición LU y descomposición de Cholesky utilizando control manual de memoria y estructuras optimizadas.
- En el lenguaje C se hace uso de la librería **Pthread** para el aprovechamiento de hilos y mejorar el proceso de inversión mediante la paralelización, se quiere aprovechar esta técnica para comprobar la mejora de rendimiento bajo la misma metodología que inicialmente plantean los algoritmos. También se usó la librería **fopenmp** para implementar la paralelización del algoritmo de inversión por descomposición de Cholesky.
- En **Python**, se utilizaron **CuPy** y **PyTorch** para aprovechar la computación paralela en GPU, replicando algoritmos similares y evaluando su rendimiento frente a la versión en CPU (NumPy, que ya optimiza el proceso de Inversión).

Módulo de comparación

Se construye para cada esquema una función de medición que registra el **tiempo de ejecución**, el **uso de memoria**, y/o otros parámetros de rendimiento. Esto incluye pruebas con matrices de distintos tamaños para observar la escalabilidad y el comportamiento cuando aumentamos el número de entradas en la estructura.

Visualización y análisis de resultados

Se generan gráficos comparativos para observar diferencias entre ejecuciones en CPU y GPU, con y sin paralelismo, permitiendo identificar los escenarios más eficientes para cada tipo de algoritmo, y el tamaño en el cual cada algoritmo se comporta mejor.

Decisiones de diseño clave

1. Uso de C para control fino de memoria y procesos: se eligió C por su cercanía al hardware, lo que permite una gestión detallada de la memoria RAM y una implementación eficiente del paralelismo mediante hilos (por ejemplo el uso de la librería **pthreads**).
2. Uso de Python con CuPy y PyTorch para GPU: Python se seleccionó por su facilidad de prototipado y sus bibliotecas modernas, que permiten explotar el paralelismo en GPU con muy poco código adicional (usar entornos gráficos como T4 que ofrece Google).
3. Comparación de versiones con y sin paralelismo: se decidió evaluar cada algoritmo bajo dos condiciones: una ejecución secuencial y una ejecución paralela, para analizar el impacto real del paralelismo en el desempeño, dado que es un tema importante dentro del curso.

4. Tamaño progresivo de matrices: se probó la inversión con matrices de diferentes dimensiones (por ejemplo, 10x10, 100x100, 200x200, 300x300, etc.) para evaluar cómo se comportan los algoritmos a mayor escala.

4.2. Tecnologías y Herramientas

Enumere los lenguajes de programación, herramientas, entornos de desarrollo, o cualquier otra tecnología que utilizó para implementar su proyecto.

Lenguajes de programación:

- C
- Python

Herramientas:

- gcc
- python3
- Virtual environments de Python
- CUDA
- Pytorch
- NVCC

Entornos de desarrollo:

- Kali Linux
- Arch Linux
- Windows
- Visual Studio Code
- Nvim
- Google Collaboratory

4.3. Detalles de Implementación

Describa los aspectos más importantes de la implementación, es decir, hable sobre detalles técnicos sin ahondar mucho. ¿Cómo codificó las funcionalidades clave? Explique los aspectos básicos sobre archivos de configuración, las estructuras de datos o algoritmos principales que utilizo según aplique en la implementación de su proyecto. (Evite pegar todo el código aquí, para ello referencias a secciones en los anexos).

Se decidió en todos los algoritmos leer las matrices con las que se trabaja desde archivos **.csv** como se puede observar en esta carpeta del repositorio. Las matrices que no necesariamente son definidas positivas pero sí son invertibles, las cuales son las que usamos en los algoritmos de eliminación Gaussiana y descomposición LU tienen el nombre **matriz_NxN_invertible.csv**. Siendo N las dimensiones de la matriz. (**Todas las matrices deben ser cuadradas**).

Para el algoritmo de inversión con descomposición de Cholesky usamos un tipo de matriz diferente las cuales son cuadradas, invertibles y definidas positivas. Se encuentran en esta carpeta del repositorio.

Se codificaron entonces 12 algoritmos base (más implementaciones CUDA)

```
├── cholesky
│   ├── cholesky.c
│   ├── cholesky_paralelo.c
│   ├── cholesky_paralelo.py
│   └── cholesky.py
├── cuda_implementaciones
│   ├── inversion_manual_impl.ipynb
│   ├── matrix_inversión_cuda.ipynb
├── gaussiana
│   ├── gaussiana.c
│   ├── gaussiana.py
│   ├── parallel_gaussiana.c
│   └── parallel_gaussiana.py
├── lu
│   ├── lu_matrix_inversion.c
│   ├── lu_matrix_inversion.ipynb
│   ├── parallel_lu_matrix_inversion.c
│   └── parallel_lu_matrix_inversion.ipynb
```

Estos algoritmos se codificaron en base a los diagramas de flujo mencionados anteriormente:

- Diagrama de flujo - Inversión con LU
- Diagrama de flujo - Inversión con reducción gaussiana
- Diagrama de flujo - Inversión con descomposición de Cholesky

En los que se puede observar mayor fidelidad es en el código C dado su bajo nivel. En Python se aprovecharon facilidades de sintaxis y por la simplicidad del propio lenguaje cambia un poco la implementación.

Para cada algoritmo paralelo, se analizó cuidadosamente el diagrama de flujo correspondiente con el objetivo de identificar cuáles eran las porciones de código habilitadas para aplicar paralelismo de manera segura. En general se pudo observar que los algoritmos de LU y Descomposición de Cholesky invierten columna por columna. Operación la cual se puede paralelizar de manera sencilla para que cada hilo calcule una columna diferente de la matriz. Esto nos asegura que no hay condiciones de carrera.

Por ejemplo, comparando el algoritmo de **inversión con LU** de manera secuencial y de manera paralela se nota una diferencia clara. Esta diferencia radica en la función para invertir matriz. Esta función secuencial (línea 57) permite invertir la matriz aplicando la sustitución hacia atrás y hacia adelante en todas las columnas, una a la vez. La función paralela (línea 111 y 129) se divide en 2. Una función para invertir columna y otra función que se aprovecha de esta para

crear hilos para cada columna y realizar el procedimiento de manera paralela para cada columna. Este es el gran cambio del algoritmo. Las demás funciones, como la función de sustitución hacia adelante o hacia atrás queda de la misma manera.

Para el algoritmo de inversión por **eliminación gaussiana,** por ejemplo, al comparar el método de inversión de matrices mediante Gauss-Jordan entre la versión secuencial y la versión paralela en su código de python, se evidencia una diferencia notable, no solo en el código sino también en el rendimiento. La función secuencial (línea 24) ejecuta el algoritmo recorriendo fila por fila, normalizando el pivote y eliminando los elementos de su columna de manera tradicional, sin ningún tipo de paralelismo. Por el contrario, la versión paralela (línea 27) utiliza Numba con prange para procesar varias filas al mismo tiempo, paralelizando la eliminación de elementos en las columnas. Sin embargo, los resultados obtenidos en las pruebas muestran un panorama curioso: mientras la versión secuencial resuelve matrices pequeñas en milisegundos (por ejemplo, matrices de 10x10 en aproximadamente 0.0004 segundos y de 500x500 en cerca de 0.8 segundos), la versión paralela presenta tiempos muchos mayores incluso en esos tamaños pequeños (alrededor de 7 segundos constantes, sin grandes variaciones). Incluso con matrices grandes, como las de 5000x5000, la versión secuencial tarda en promedio cerca de 98 segundos, mientras que la versión paralela logra reducirlo a 62 segundos, mostrando su ventaja solo en escalas muy grandes. Este fenómeno se debe a la sobrecarga inicial que implica la compilación y paralelización con Numba, la cual solo se amortiza en matrices gigantescas. Las demás funciones, como la lectura de archivos o la construcción de la matriz aumentada, permanecen prácticamente iguales en ambas versiones. Aquí el cambio crucial es la paralelización del bucle interno, cuyo impacto depende fuertemente del tamaño del problema.

En el caso de la inversión por descomposición de Cholesky es muy parecida a la descomposición LU. Lo que tenemos es una función secuencial (línea 87) para invertir cada columna de la matriz, una detrás de otra. Representado en el ciclo for. Ahora para la función paralela (línea 88) se utiliza la librería openmp, con el objetivo de paralelizar el for que invierte las columnas. Esto permite hacer que cada iteración del for se haga al mismo tiempo que las otras.

Por último el caso de los algoritmos con procesamiento en CUDA, La implementación se estructuró en torno a una arquitectura modular basada en clases, donde cada clase representa un método específico de inversión de matrices (LU, Gauss-Jordan, Cholesky) y una tecnología subyacente (CuPy o PyTorch). Las funcionalidades clave fueron codificadas con el objetivo de permitir pruebas comparativas bajo condiciones controladas.

Para la inversión de matrices se implementaron algoritmos manuales directamente en GPU usando:

- **CuPy**, que permite trabajar con arrays tipo NumPy pero ejecutados en CUDA.
- **PyTorch**, aprovechando su soporte para operaciones tensores en GPU.

Cada clase contiene métodos como `cupy_manual_lu_inverse()` o `torch_gauss_jordan_inverse()`, donde se codificaron paso a paso los algoritmos de factorización (por ejemplo, la sustitución hacia adelante y atrás en LU). Estos métodos trabajan sobre matrices cuadradas leídas desde archivos CSV (ver sección de anexos).

Se utilizaron estructuras básicas como arrays de NumPy para lectura desde disco, que luego se convertían a tensores o arrays de CuPy para ser procesados en GPU. La lectura de datos fue centralizada en una función `leer_matriz_csv()`, lo cual facilitó el manejo flexible de diferentes tamaños de entrada.

Para comparar el rendimiento, se implementó una función de benchmarking (`medir_tiempos_por_tamaño`) que ejecuta los algoritmos sobre matrices de diferentes tamaños, sincroniza manualmente la GPU antes y después de la operación (para obtener mediciones precisas), y registra los tiempos. Los resultados fueron graficados usando `seaborn` y también exportados a un DataFrame para su análisis tabular (ver sección de Resultados).

La organización del código permite reutilizar componentes, como la lógica de inversión, los métodos de sincronización GPU y la medición de tiempo, manteniendo una separación clara entre lógica de prueba y lógica matemática.

5. Pruebas y Evaluación

5.1. Metodología de Pruebas

Describe cómo probaste tu proyecto. ¿Qué enfoque de pruebas utilizaste? ¿Creaste casos de prueba específicos?

El proyecto fue probado mediante un enfoque de **pruebas funcionales y de rendimiento,** con énfasis en la correctitud matemática comparando el producto de las matrices $A \cdot A^{-1}$.
Para ello se diseñan pruebas controladas sobre matrices cuadradas invertibles de diferentes tamaños (10x10 hasta 500x500).

Específicamente se hace uso de dos tipos de matrices:

- **Matrices generales invertibles,** generadas previamente y luego agrupadas como archivos .csv
- **Matrices definidas positivas (dpi),** estas matrices almacenadas de la misma manera se usan para pruebas específicas en el algoritmo de Cholesky

Los tamaños de matriz evaluados incluyeron 100, 200, 300, 400 y 5000. Para cada tamaño, se ejecutaron todos los algoritmos disponibles (LU, Gauss-Jordan, Cholesky) en sus versiones Cupy y PyTorch.

Medición de desempeño

Se implementaron un módulo para la medición con precisión del tiempo de ejecución en cada algoritmo. Los tiempos estructurados tipo `DataFrame` y representados gráficamente usando `seaborn`, lo que permitió comparar el rendimiento de los métodos en función del tamaño de la matriz y la tecnología utilizada.

5.2. Casos de Prueba y Resultados

Presente los casos de prueba más importantes que ejecutó y los resultados obtenidos. Puede usar para ello una tabla como la siguiente:

ID Caso de prueba	Descripción del caso de prueba	Resultado esperado	Resultado obtenido	Éxito/Fallo
CP-001	Invertir matrices bien condicionadas Usando Cupy y PyTorch	Matriz invertida correctamente, el resultado presenta un error bajo	Respuesta válida, la matriz se invierte y se toman los tiempos	Éxito
CP-002	Invertir la matriz mal condicionada usando PyTorch	Esperar que el sistema maneje estabilidad y se capture la excepción	Se produce un error, y la excepción no se había manejado	Fallo
CP-003	Ejecutar Cholesky con matriz cuadrada no simétrica positiva	Se lanza una excepción indicando que la matriz no es positiva definida	En efecto, Cholesky maneja las excepciones	Éxito
CP-004	Realizar gráfica de tiempos usando todos los algoritmos y tamaños	Se espera recoger todos los tiempos al invertir matrices almacenadas en archivos	Dado que las matrices fueron generadas de manera general, hay error cuando se ejecuta Cholesky	Fallo
CP-005	Gaussiana secuencial C	Matriz invertida correctamente con error bajo.	Respuesta válida, la matriz se invierte y se toman los tiempos	Éxito
CP-006	Gaussiana paralelo C	Matriz invertida correctamente con error bajo, mejores tiempos que secuencial.	Matriz invertida correctamente con error bajo, aunque los tiempos al invertir matrices menores a 500x500 son mayores que en su versión secuencial.	Fallo
CP-007	Gaussiana secuencial Python	Matriz invertida correctamente con error bajo.	Matriz invertida correctamente con error bajo, se toman los tiempos.	Éxito
CP-008	Gaussiana paralelo Python	Matriz invertida correctamente con error bajo.	Matriz invertida correctamente con error bajo, aunque los tiempos al	Fallo

			invertir matrices menores a 500x500 son mayores que en su versión secuencial.L	
CP-009	Cholesky secuencial C	Matriz correctamente invertida, con tiempos pequeños y mejores que en su versión de python.	Matriz invertida correctamente pero con tiempos elevados y similares a los tiempos de su versión de python.	Fallo
CP-010	Cholesky paralelo C	Matriz correctamente invertida con mejores tiempos que en su versión secuencial.	Matriz invertida correctamente, con tiempos significativamente mejores que en su versión secuencial.	Éxito
CP-011	Cholesky secuencial Python	Matriz inversa correctamente con error bajo	Matriz invertida correcto con tiempos un poco altos en algoritmos	Éxito
CP-012	Cholesky paralelo Python	Mejor que el secuencial al ya que se reparten los cálculos en hilos que se ejecutan al mismo tiempo para los cálculos.	en matrices que no son demasiado grandes puede empeorar en la creación de múltiples hilos	Fallo

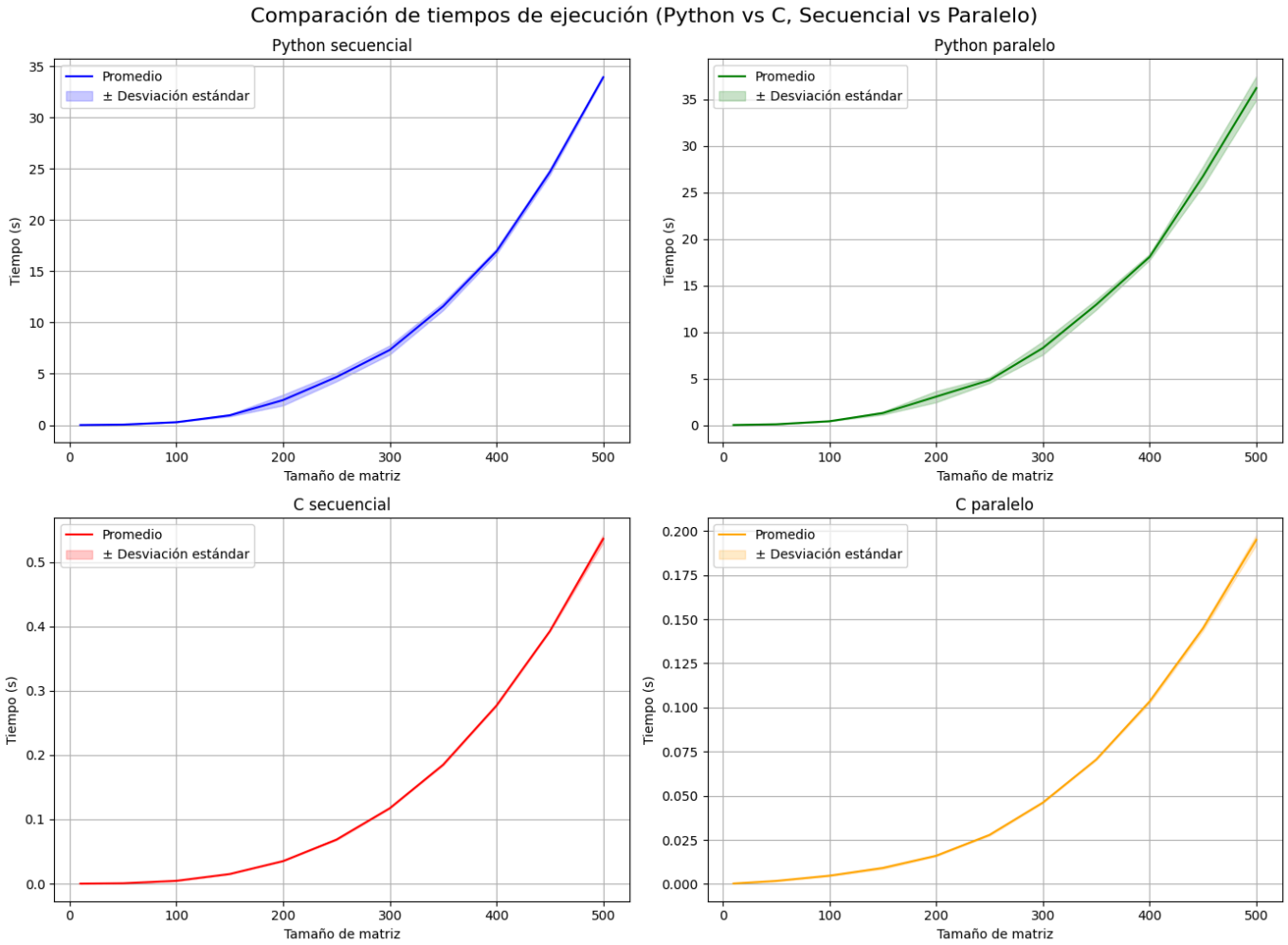
5.3. Evaluación del Rendimiento (si aplica)

Si su proyecto tiene aspectos de rendimiento (ej: tiempos de ejecución, uso de CPU/memoria), presente y analiza las métricas obtenidas.

El proyecto cuenta con métricas de rendimiento a nivel de tiempos de ejecución dependiendo el tamaño de la matriz que se esté invirtiendo.

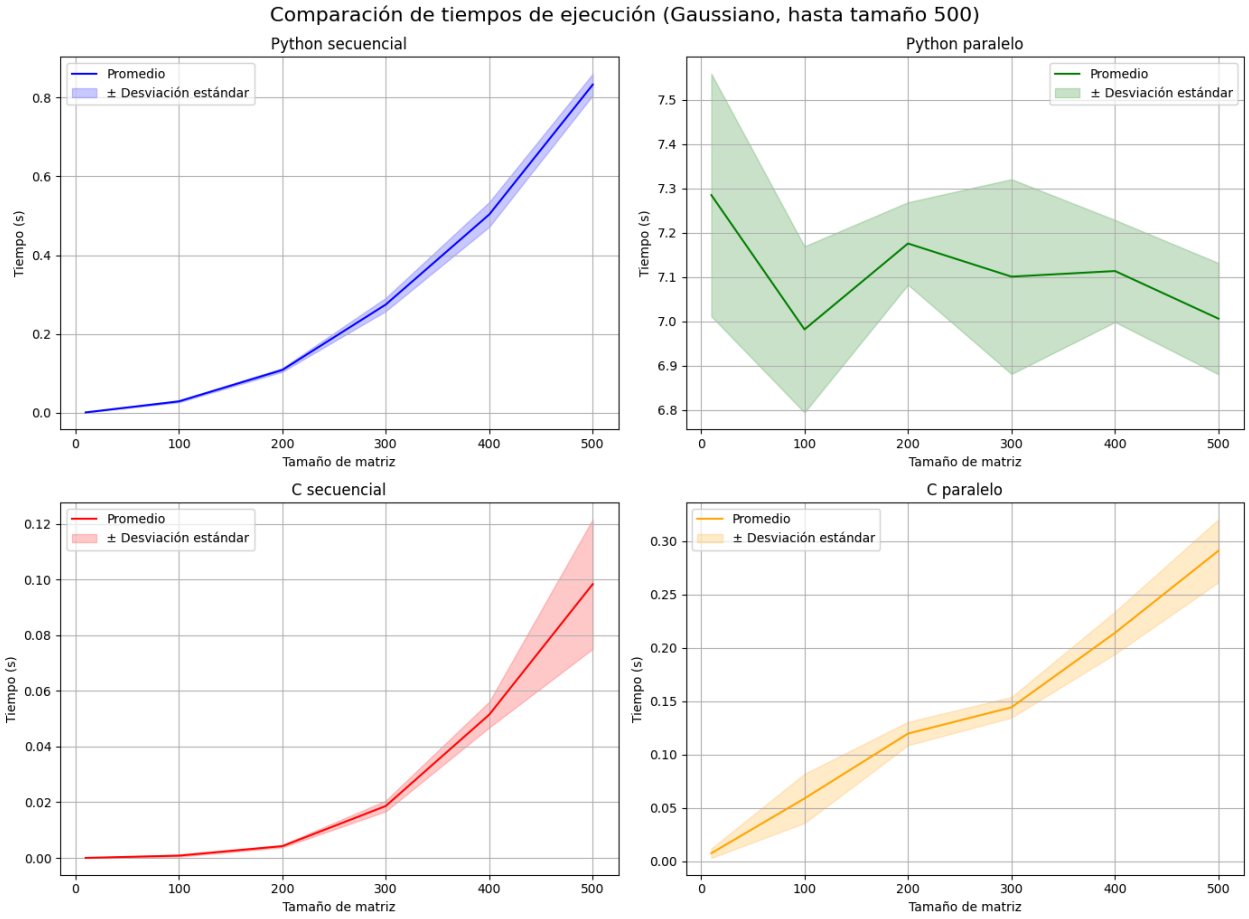
Las gráficas obtenidas para los 3 algoritmos base (Sin CUDA) son las siguientes:

LU



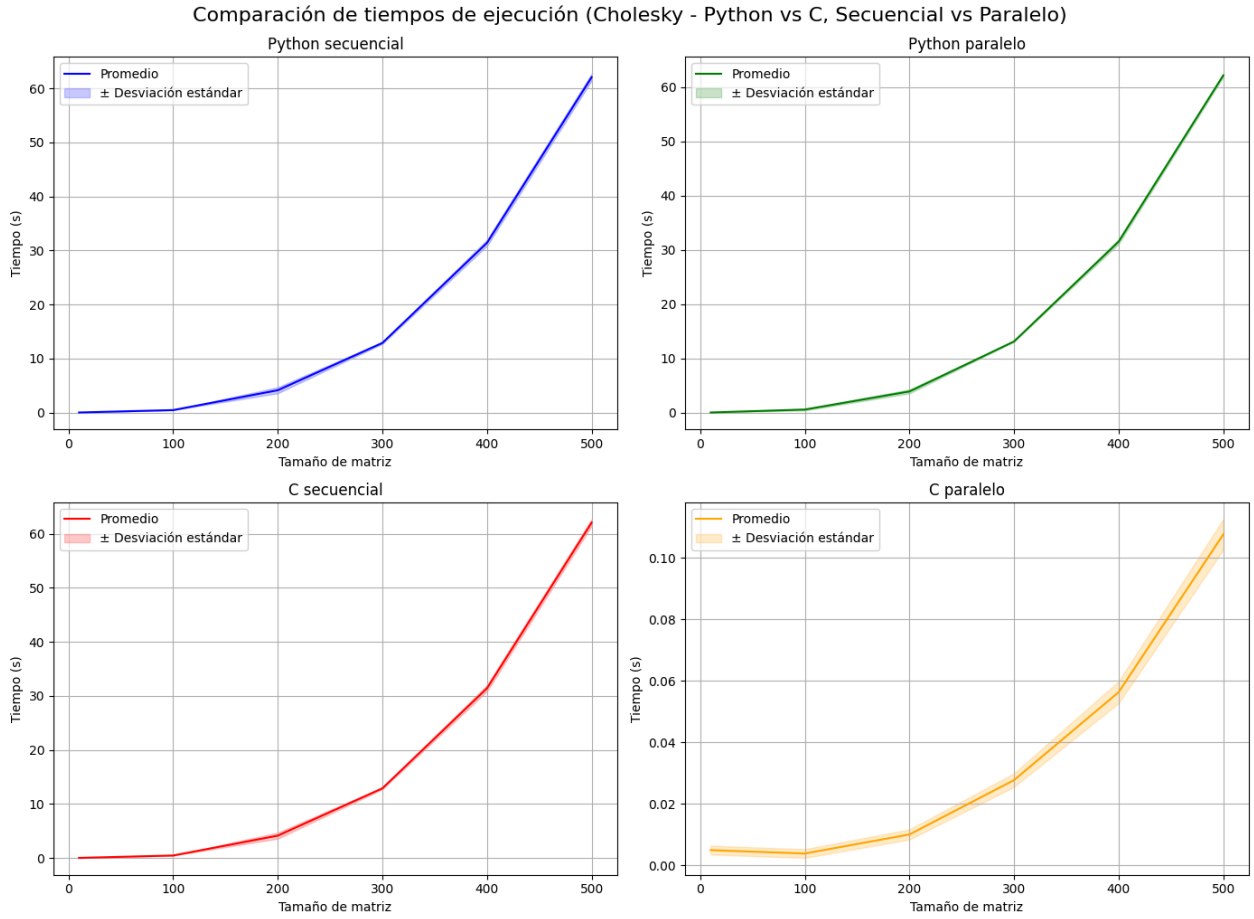
	Tamaño matriz	Python sec (s)	±Python sec	Python par (s)	±Python par	C sec (s)	±C sec	C par (s)	±C par
0	10	0.001383	0.000123	0.004663	0.001497	0.000007	0.000001	0.000313	0.000070
1	100	0.277820	0.010883	0.411448	0.014940	0.004400	0.000064	0.004757	0.000233
2	200	2.436070	0.514011	3.055704	0.594555	0.035061	0.000204	0.016063	0.000333
3	300	7.323826	0.434860	8.283672	0.708493	0.117264	0.000389	0.046181	0.000288
4	400	16.932820	0.263757	18.068277	0.305661	0.276883	0.000945	0.103423	0.000862
5	500	33.915898	0.081215	36.206505	1.265797	0.536134	0.005957	0.194988	0.002733

Reducción Gaussiana:



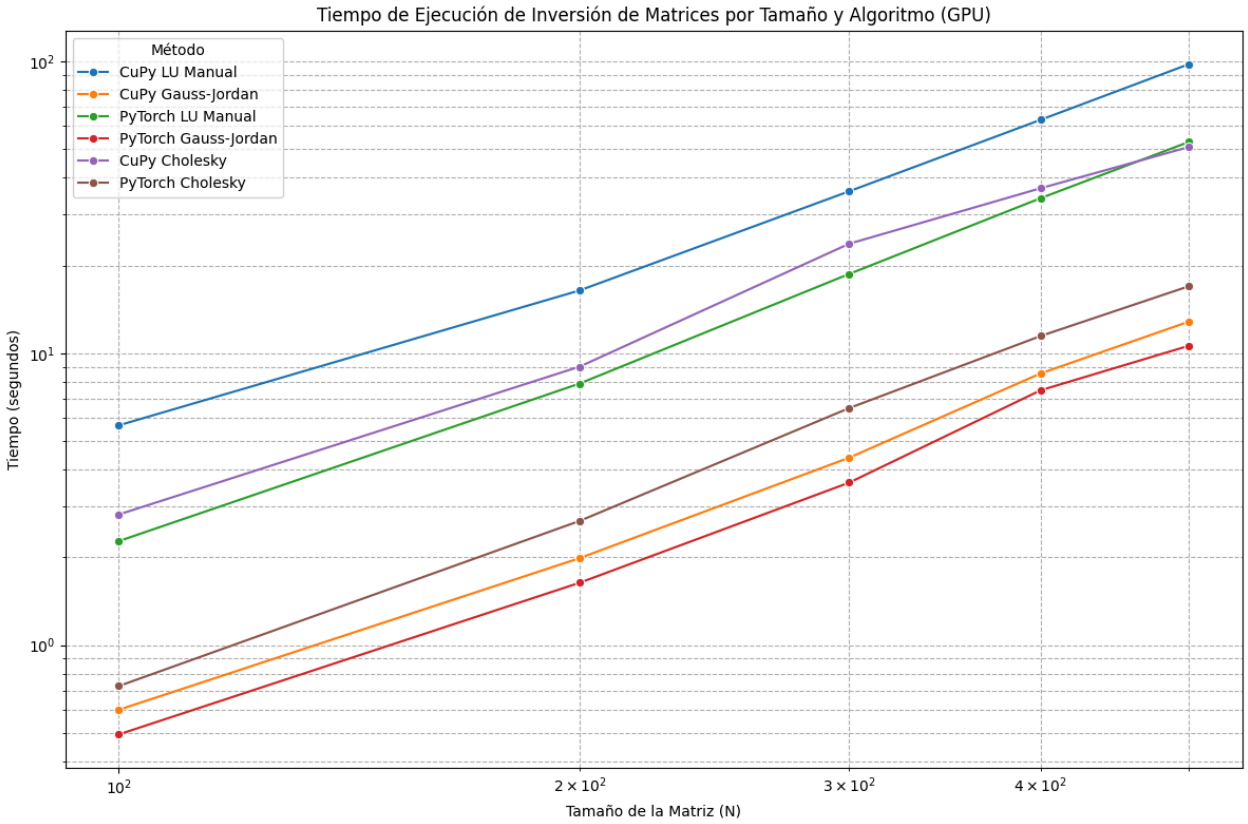
	Tamaño matriz	Python sec (s)	±Python sec	Python par (s)	±Python par	C sec (s)	±C sec	C par (s)	±C par
0	10	0.000539	0.000178	7.284962	0.273720	0.000003	0.000001	0.007529	0.004381
1	100	0.028529	0.002507	6.981542	0.187512	0.000807	0.000365	0.058793	0.022976
2	200	0.108252	0.004587	7.175553	0.093060	0.004208	0.000452	0.119572	0.010901
3	300	0.274572	0.016778	7.100759	0.219884	0.018648	0.001945	0.144204	0.009588
4	400	0.503445	0.031403	7.113406	0.115229	0.051497	0.004630	0.213951	0.019766
5	500	0.833275	0.027274	7.005846	0.125609	0.098401	0.023281	0.290930	0.029460

Descomposición de Cholesky:



	Tamaño matriz	Python sec (s)	±Python sec	Python par (s)	±Python par	C sec (s)	±C sec	C par (s)	±C par
0	10	0.00518	0.001070	0.00836	0.001484	0.00518	0.001070	0.004916	0.001427
1	100	0.44566	0.009338	0.54290	0.160103	0.44566	0.009338	0.003813	0.001416
2	200	4.11884	0.555413	3.87974	0.356430	4.11884	0.555413	0.010007	0.001637
3	300	12.87718	0.176355	13.09572	0.045521	12.87718	0.176355	0.027684	0.002188
4	400	31.40020	0.566703	31.49776	0.428556	31.40020	0.566703	0.056273	0.003539
5	500	62.06222	0.626677	62.14066	0.293462	62.06222	0.626677	0.107551	0.004962

Tiempos de ejecución algoritmos usando CUDA



Consolidado de tiempos de manera tabular

Tamaño	CuPy LU Manual	CuPy Gauss-Jordan	CuPy Cholesky	PyTorch LU Manual	PyTorch Gauss-Jordan	PyTorch Cholesky
100	5,663764	0,600719	2,803164	2,272295	0,494634	0,72423
200	16,419461	1,987367	9,000503	7,877842	1,636727	2,66525
300	35,89374	4,390265	23,734057	18,703206	3,606114	6,49157
400	63,060311	8,530182	36,758364	33,986567	7,468879	11,4598
500	97,844172	12,840536	50,892854	52,987177	10,61358	16,9757

5.4. Problemas Encontrados y Soluciones

Describa los problemas o errores significativos encontrados durante el desarrollo y las pruebas, y cómo los resolvió.

Durante el desarrollo de los métodos de inversión de matrices, tanto en sus versiones secuenciales como en sus versiones paralelas, y en los lenguajes C y Python, nos encontramos con varios problemas técnicos que exigieron soluciones creativas:

1. **Paralelización ineficiente en Python para matrices pequeñas:** En Python al aplicar paralelización con Numba o con hilos, notamos que para matrices pequeñas (por debajo de 500x500), los tiempos de ejecución eran peores en la versión paralela que en la secuencial. Esto se debía al costo inicial que genera la compilación de Numba o de la creación de múltiples hilos. Como **solución** decidimos incluir en nuestro análisis final, explicando que la versión paralela en Python sólo resulta beneficiosa en matrices grandes, y que para casos pequeños es mejor usar la versión secuencial.
2. **Sobrecarga por compilación con Numba en Python:** La primera ejecución de los códigos en Python paralelos con Numba siempre era mucho más lenta debido al proceso de compilación inicial. Como **solución** ejecutamos un "calentamiento" previo, realizando una corrida inicial con una matriz pequeña antes de medir los tiempos, para evitar que la compilación afecte los resultados.
3. **Lectura y escritura de matrices de gran tamaño:** En matrices de gran tamaño, el tiempo de lectura/escritura de los archivos CSV también se volvió significativo, afectando las mediciones de tiempo. Como **solución** separamos claramente el tiempo de ejecución del algoritmo del tiempo de lectura/escritura en los reportes, para enfocarnos solo en el rendimiento del algoritmo de inversión.
4. **Estructura de los códigos:** El mantener cuatro versiones distintas por método (secuencial y paralela en C y Python) generaba confusión al momento de depurar o actualizar o funciones compartidas. Como **solución** organizamos el proyecto en carpetas estructuradas por método de inversión, documentamos cada función, para facilitar el mantenimiento y la comprensión del código.
5. **Manejo de memoria dinámica en C:** Uno de los principales retos fue la correcta asignación y liberación de memoría dinámica en los programas de C, especialmente al trabajar con matrices aumentadas en el método de Gauss-Jordan y durante la inversión con LU. Como **solución** creamos funciones auxiliares específicas para alocar y liberar matrices, como **allocate_matrix()** o **alocar_matriz()**, para asegurar que no quedaran fugas de memoria. También revisamos cuidadosamente que todos los **malloc** y **free** estuvieran bien balanceados.
6. **Lectura y validación de archivos CSV en C:** Al leer los archivos CSV con matrices en C, nos encontramos con problemas como filas incompletas, columnas faltantes o errores por matrices no cuadradas. Como **solución** implementamos funciones robustas de lectura como leer_csv() y validaciones para garantizar que la matriz cargada fuera cuadrada, generando mensajes de error claros en caso de fallos.
7. **Divisiones por cero y singularidades numéricas:** Al procesar matrices mal condicionadas o de números muy pequeños, nos topamos con errores por divisiones cercanas a cero. Como **solución** añadimos comprobaciones de pivoteo parcial y umbrales mínimos **(fabs(A[i][i]) < 1e-10)** para detectar matrices singulares y abortar el proceso con mensajes de error amigables.

8. **Paralelización limitada en matrices pequeñas:** En las versiones paralelas en C, observamos que para matrices pequeñas el rendimiento era incluso peor que en las versiones secuenciales, debido al costo de crear y sincronizar múltiples hilos. Como **solución** incluimos en nuestras conclusiones que la paralelización solo es efectiva para matrices de tamaño moderado o grande, y que en matrices pequeñas es preferible mantener la versión secuencial.
9. **Incompatibilidad entre versiones CuPy, PyTorch y CUDA**
   Uno de los problemas que mas dificultad presento, fue la incompatibilidad entre la versión instalada de `cupy-cuda12x` y la versión de CUDA soportada por la instalación de PyTorch, esto provocaba que Cupy detectara correctamente la GPU, pero PyTorch no, o viceversa.
   Esto se debía a que CuPy permite elegir una versión explicita de CUDA (`cupy-cuda12x`, `cupy-cuda11x`, etc), mientras que PyTorch instala una versión fija de CUDA dentro del paquete `torch` y puede que no siempre sean compatibles.
10. **Fallos por matrices no invertibles o mal condicionadas**
    Algunas matrices cargadas desde archivos CSV no eran invertibles o eran numéricamente inestables, lo que se hizo (especialmente en la clase de `Inversión with Cholesky`) Fue aplicar técnicas de regularización que consiste en sumar un pequeño valor a la diagonal principal.

6. Conclusiones

Resuma los logros de tu proyecto. ¿Cumpliste los objetivos iniciales? Discuta lo que aprendió al realizar el proyecto, especialmente en relación con los conceptos de sistemas operativos del curso.Evalúe el éxito general de su proyecto.

- El proyecto cumplió con los objetivos iniciales. Se logró ejecutar, comparar y discriminar diferentes algoritmos corriendo sobre diferentes plataformas para invertir matrices y encontrar cual es el mejor en términos de rendimiento temporal. Se concluye que los algoritmos implementados usando CUDA tienen un rendimiento superior respecto a los algoritmos implementados sobre C y Python, incluso con paralelismo.
- Entre Python y C, se destaca C de gran manera mostrando muy buenos tiempos para llevar a cabo la inversión de matrices de 500x500.
- Se aprendió a realizar paralelismo en C con fopenmp, pthread y a manejar paralelismo en Python. Se aprendió lo que es el GIL y sus implicaciones en el paralelismo con Python.
- Se aprendió que en todos los casos, no siempre es mejor paralelizar los algoritmos, dado que en ocasiones, el tiempo de ejecución de
- El proyecto no solo cumplio su objetivo de comparar algoritmos de inversión de matrices en diferentes entornos de cómputo, sino que también permite profundizar en el uso práctico de aceleración por hardware mediante CUDA, a través de las bibliotecas CuPy y PyTorch
- La comprensión como el sistema operativo y el entorno de ejecución gestionan el acceso a la GPU, en el caso puntual fue el entendimiento del control de sincronización entre CPU y GPU `torch.cuda.synchronize()` o `cp.cuda.runtime.deviceSynchronize()` para la obtención de mediciones de tiempo mas precisas y confiables.

7. Referencias

Enumere todas las fuentes que consultaste (libros, artículos, sitios web, documentación de herramientas, etc.) utilizando un formato de citación consistente.

1. CuPy Developers. CuPy: A NumPy-compatible array library accelerated by CUDA. Disponible en: https://cupy.dev/ [Accedido: 29-jun-2025].
2. **Google Research.** *Using GPUs in Google Colab*. Google Colab Documentation, 2025. Disponible en: https://research.google.com/colaboratory/ [Accedido: 8-jul-2025].
3. International Organization for Standardization (ISO). *Programming Languages — C (ISO/IEC 9899:2018)*. Disponible en: https://www.open-std.org/jtc1/sc22/wg14/ [Accedido: 8-jul-2025].
4. A. M. H. *et al*., “Matrix: command-line tool for matrix inversion and multiplication using Pthreads,” GitHub, 2016. [Online]. Available: Matrix-C project