UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

IMPLEMENTACIÓN DE NÚCLEOS COMPUTACIONALES PARA REDES NEURONALES CON JAX

Trabajo Fin de Máster

Máster Universitario en Computación en la Nube y de Altas Prestaciones / Cloud and High-Performance Computing

AUTOR/A: Solaz Vivó, Celia

Tutor/a: Alonso Jordá, Pedro

Cotutor/a: Castelló Gimeno, Adrián

CURSO ACADÉMICO: 2023/2024

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

MÁSTER UNIVERSITARIO EN COMPUTACIÓN EN LA NUBE Y DE ALTAS PRESTACIONES

TRABAJO FIN DE MÁSTER

IMPLEMENTACIÓN DE NÚCLEOS COMPUTACIONALES PARA REDES NEURONALES CON JAX

AUTORA: CELIA SOLAZ VIVÓ

TUTORES: ADRIÁN CASTELLÓ GIMENO
y PEDRO ALONSO JORDÁ

CONVOCATORIA: SEPTIEMBRE 2024

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

TRABAJO FIN DE MÁSTER

IMPLEMENTACIÓN DE NÚCLEOS COMPUTACIONALES PARA REDES NEURONALES CON JAX

AUTORA: CELIA SOLAZ VIVÓ

TUTORES: ADRIÁN CASTELLÓ GIMENO
y PEDRO ALONSO JORDÁ

Declaración de autoría:

Yo, Celia Solaz Vivó, declaro la autoría del Trabajo Fin de Máster titulado "Implementación de núcleos computacionales para redes neuronales con JAX" y que el citado trabajo no infringe las leyes en vigor sobre propiedad intelectual. El material no original que figura en este trabajo ha sido atribuido a sus legítimos autores.

12 de septiembre de 2024

Fdo: Celia Solaz Vivó

Resumen:

Este proyecto de investigaicón se realiza con un objetivo principal, que es el estudio de la portabilidad de la biblioteca *JAX* de *Python*, desarrollada por *Google*, en diferentes componentes *hardware* (*CPU, GPU y TPU*).

JAX es una herramienta de gran alcance que combina la familiaridad y simplicidad de la biblioteca de álgebra lineal *NumPy* con avanzadas capacidades de diferenciación automática y ejecución de alto rendimiento sobre aceleradores*hardware*.

Además del análisis de portabilidad de código, llevamos a cabo una comparativa de rendimiento con otras bibliotecas populares como son *NumPy*, *Cupy* y *Numba*, utilizando la multiplicación de matrices como caso de estudio. La comparativa se ha realizado en términos de operaciones en coma flotante por segundo (*GFLOPS*).

Las pruebas de rendimiento se llevan a cabo en *Notebooks* de *Google Colab* y *Kaggle* aprovechando, de esta forma, recursos *hardware* diversificados. Se obtiene y analiza el rendimiento máximo alcanzable para cada componente *hardware*.

Palabras clave: *JAX*, *NumPy*, *Cupy*, *Numba*,*Kaggle*, *Google Colab*, portabilidad, redes neuronales.

Abstract:

This research is conducted with a primary objective: to study the portability of the *JAX* library for *Python*, developed by Google, across different hardware components (*CPU, GPU, and TPU*).

JAX is a powerful tool that combines the familiarity and simplicity of NumPy with advanced capabilities for automatic differentiation and high-performance hardware-accelerated execution.

In addition to the code portability study, we conduct a performance comparison with other popular libraries such as *NumPy*, *Numba* and *CuPy*, using matrix multiplication as the case study. The comparison is performed using billions of floating point operations per second (*GFLOPS*).

The performance tests will be carried out in *Google Colab* and *Kaggle* notebooks, thereby leveraging the diversified hardware resources. The maximum achievable performance, in seconds, for each hardware component is obtained and analyzed.

Keywords: *JAX, NumPy, CuPy, Numba, Kaggle, Google Colab, portability, and neural networks.*

Agradecimientos:

En primer lugar, quiero agradecer a Pedro y Adrián, mis tutores del trabajo fin de Máster, por todo lo que han hecho por mí. Gracias por su predisposición a ayudar siempre que tenía cualquier duda o contratiempo, por buscar huecos donde no los había para poder atenderme y aconsejarme, por apoyarme en todo lo posible este tiempo. Este proyecto jamás hubiese sido posible.

Gracias a mi familia por estar ahí. Desde el primer día, en la decisión de que camino seguir con respecto al ámbito académico, con la selección del máster hasta en mis peores y mejores momentos tanto personales como profesionales. Cabe destacar que no soy fácil de tratar cuando estoy en mis peores momentos, pero ellos saben cómo mejorarlos y apoyarme para aprender de ellos.

Gracias a todas las grandes amistades que me han apoyado, tanto nuevas como viejas. Primeramente, a Andrea y Laya que me conocen desde hace más de 7 años, que han sabido escucharme en todo momento, han podido sacarme una sonrisa en los malos momentos y apoyarme en mis locuras.

Finalmente, a esas 8 personas maravillosas, Victoria, José, Luca, Alejandro, Aitor, José Luis, Dani y Eddy; que se han cruzado en mi camino este último año y han hecho posible volver a sacar a la niña interior que tenía escondida desde hace un tiempo. Logrando allanar con sus sonrisas y palabras esta nueva etapa personal. Espero que esta amistas siga a lo largo de los años, aunque estemos en países diferentes.

Índice general

Índice de Figuras

1 Introducción

1.1 Introducción

En la actual era digital, estamos inmersos en un crecimiento exponencial en el volumen de datos generados, consumidos y almacenados. Este fenómeno, se conoce como el crecimiento de datos, que ha sido impulsado por la expansión de dispositivos conectados, plataformas en línea y la digitalización de procesos en diversos sectores.

Este aumento de datos presenta desafíos significativos para el procesamiento rápido y eficiente, especialmente en el contexto del aprendizaje automático y las redes neuronales, donde la capacidad para analizar y extraer información útil de grandes conjuntos de datos es crucial.

En respuesta a estos desafíos, las técnicas de paralelización y las bibliotecas creadas en los diferentes lenguajes de programación han surgido como una de las soluciones más fundamentales para la mejora de la eficiencia computacional y del rendimiento de los sistemas.

La paralelización implica distribuir tareas computacionales complejas entre múltiples unidades de procesamiento, como pueden ser núcleos de *CPU* o unidades de procesamiento gráfico (*GPU*). Dicha distribución permite la realización cálculos simultáneos, acelerando el tiempo de procesamiento y mejorando la escalabilidad.

En este trabajo, nos centramos en implementar los núcleos de computo para redes neuronales utilizando la biblioteca *JAX*. Esta biblioteca se encuentra en el lenguaje de programación *Python* desarrollada por *Google Research* para facilitar la construcción y el entrenamiento de los modelos de aprendizaje automático, resaltando su capacidad en la ejecución operacional eficiente en *hardware* acelerado como son *GPU* y *TPU*.

El objetivo principal es la exploración de la paralelización eficiente de operaciones matemáticas complejas, como llegan a ser las multiplicaciones de matrices. A lo largo del documento, se analizarán y compararán diversas bibliotecas de *Python* especializadas en las operaciones matemáticas estudiadas, evaluando su impacto en la velocidad de cálculo y rendimiento en *GFLOPS*.

Mediante este estudio, se pretende proporcionar un marco sólido para comprender y aprovechar al máximo las capacidades de paralelización de *JAX* en entornos de redes

neuronales. Al hacerlo, se contribuye en el avance y la optimización del procesamiento de grandes volúmenes de datos, abriendo nuevas posibilidades para aplicaciones avanzadas en inteligencia artificial y el aprendizaje automático.

1.2 Motivación

Con este estudio, se pretende dar paso al uso de la biblioteca *JAX* para centrarnos en dos problemas cruciales en el campo del aprendizaje automático y las redes neuronales, que son el crecimiento exponencial de datos y la necesidad de mejorar la eficiencia computacional en todos los dispositivos.

El crecimiento masivo de datos, ha creado un desafío para el procesamiento rápido y eficiente de información. Abriendo las puertas a la creación de nuevas oportunidades sin precedentes para extraer conocimiento valioso, que requieren métodos avanzados para manejarlos y analizarlos.

Por otro lado, la paralelización ofrece una solución para aumentar el rendimiento computacional al distribuir tareas entre múltiples unidades de procesamiento, como son *TPUs* o *GPUs*. Esta técnica no solo acelera el procesado de dato, sino que también mejora la escalabilidad de las aplicaciones, permitiendo la construcción de modelos más complejos y procesarlos en un menor tiempo.

Usaremos *JAX* como herramienta principal en nuestro caso de estudio para evaluar su capacidad en el cálculo de operaciones matemáticas que se realizan en las diferentes capas de las redes neuronales, como es la multiplicación de matrices entre otros.

Por lo tanto, buscaremos explorar cómo la implementación de núcleos computacionales con *JAX* puede optimizar el rendimiento de las redes neuronales mediante técnicas de paralelización avanzadas. Al hacerlo, esperamos contribuir en el desarrollo de soluciones más eficientes y escalables para el procesado de datos en el contexto del aprendizaje automático.

Por último, queremos darle importancia a la portabilidad del código entre diferentes plataformas de *hardware*. La biblioteca *JAX* ofrece esta capacidad de portabilidad de manera excepcional. Permite la ejecución de cálculos en diferentes tipos de *hardware* sin necesidad de modificar el código, lo que facilita la transición entre distintos entornos de hardware (como *CPU, GPU* y *TPU*) sin realizar cambios significativos en el código fuente.

1.3 Objetivos

Este trabajo gira en torno a una premisa la cual formula que con el diverso conocimiento de las bibliotecas más conocidas en *Python* para el cálculo de operaciones matemáticas y el uso de diferentes operaciones matemáticas con diferentes tamaños de experimento, demostrar si la biblioteca creada por *Google Research*, *JAX*, es buena opción para mejorar y acelerar el rendimiento de las redes neuronales. A partir de esta premisa, hemos establecido los siguientes objetivos:

1. **Exploración de Bibliotecas Matemáticas en *Python:*** instalación y estudio de las principales bibliotecas de *Python* para el cálculo de operaciones matemáticas, como son *Numpy* o *Cupy*.

2. **Exploración de la Biblioteca *JAX*:** instalación y conocimiento de la estructura de *JAX* para realizar la comparación con otras bibliotecas.

3. **Pruebas con Diferentes Tamaños de Experimento:** realizar experimentos con diferentes tamaños y operaciones matemáticas para medir el impacto de *JAX* en términos de velocidad y eficiencia computacional.

4. **Evaluación del Rendimiento:** Comparar los resultados obtenidos en términos de rendimiento en *GFLOPS*.

5. **Análisis de Resultados:** Identificar en que casos *JAX* demuestra un rendimiento superior.

El resultado esperado de esto es comprobar que la biblioteca *JAX* es adecuada para la implementación en el cálculo de operaciones matemáticas, contribuyendo en el procesamiento y análisis de grandes volúmenes de datos.

1.4 Organización de la memoria

En la memoria comenzaremos con la explicación teórica (Sección 2 Marco teórico) de los distintos términos y tecnologías así como sus aplicaciones y relaciones entre sí.

Se realizará una exposición clara de las palabras clave mencionadas con anterioridad conjuntamente a su vinculación con el estudio de la implementación de núcleos computacionales para redes neuronales con *JAX*, e investigaciones previas que nos han ayudado a dar forma a nuestro trabajo (Sección 3 Estado del arte).

A continuación, se detallará le metodología empleada (Sección 4 Metodología y Sección 5 Implementación), incluyendo el diseño de los experimentos utilizando *JAX*, *Cupy* y *NumPy*, y la configuración del entorno de pruebas. Esta sección proporcionará una visión completa de los pasos seguidos para llevar a cabo el estudio.

Seguidamente, se presentarán los resultados obtenidas y su análisis (Sección 6 Experimentos). Se incluirán comparativas de rendimiento entre *JAX* y otras bibliotecas, acompañadas de gráficos que facilitaran la interpretación de los resultados obtenidos. La discusión de los resultados permitirá entender mejor las ventajas y limitaciones de *JAX* en el contexto del aprendizaje automático.

Finalmente, la memoria cerrará con la conclusión (Sección 7 Conclusiones y trabajo futuro) sobre los datos obtenidos y cómo repercute a la afirmación sobre la cual se basa este estudio, añadiendo una introducción a posibles investigaciones que podrían tener lugar a raíz de este proyecto de investigación.

2 Marco teórico

2.1 NumPy

NumPy [4] es una biblioteca fundamental para el cálculo de operaciones matemáticas en el ámbito de la computación científica en *Python*. Proporciona soporte para matrices y arreglos multidimensionales, junto con una colección de funciones matemáticas para operar con los datos. *NumPy* tiene algunas características principales que presentaremos a continuación:

- **Matrices y Vectores:** *NumPy* proporciona el objeto *'ndarray'* que permite almacenar y manipular datos de manera eficiente.
- **Funciones Matemáticas:** incluye funciones universales para operaciones comunes de álgebra lineal.
- **Eficiencia:** las operaciones están optimizadas para que sean rápidas y eficientes.
- **Interoperabilidad:** se integra bien con otras bibliotecas de *Python*, como pueden ser *Pandas* o *SciPy*.

2.1.1. Instalación

A continuación, explicaremos cómo instalamos *NumPy* en los entornos de *Google Colab*/*Kaggle* y *Anaconda*.

Google Colab* o *Kaggle

Para realizar la instalación de la biblioteca *NumPy* en *Google Colab* es necesario tener una cuenta en *Google* y acceder a la plataforma *Google Drive*. Para realizar la instalación en *Kaggle* también debemos crear una cuenta.

Una vez estamos dentro ya podemos crear un nuevo *notebook* de *Google Colab* o *Kaggle*, como se observa en las Figuras 1 y 2:

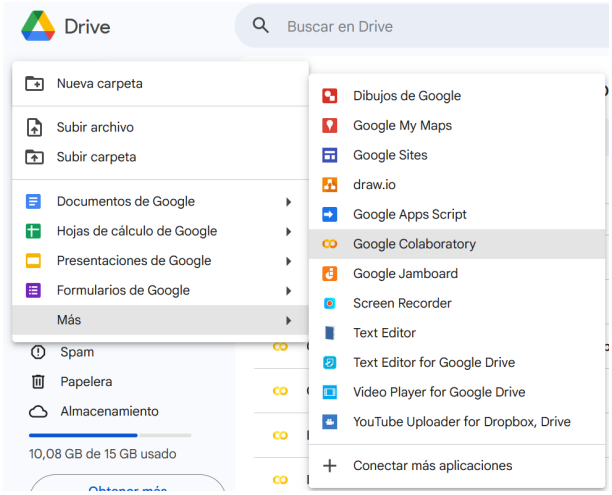

Figura 1: Creación de un *notebook* de *Google Colab* en *Google Drive*.

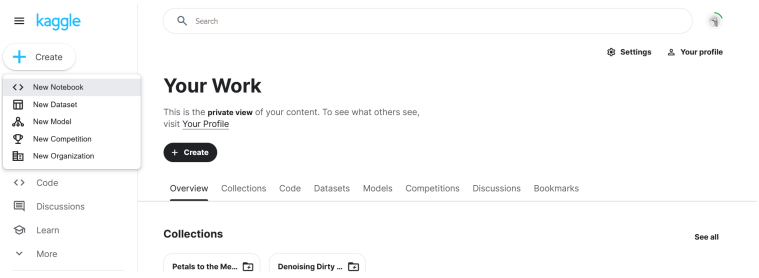

Figura 2: Creación de un *notebook* en *Kaggle*.

Ahora que tenemos el *notebook* creado, damos paso a escribir el código necesario para realizar la instalación de la biblioteca *NumPy* en una nueva celda de código. El código necesario para la instalación es *"!pip install numpy"* (Figura 3):

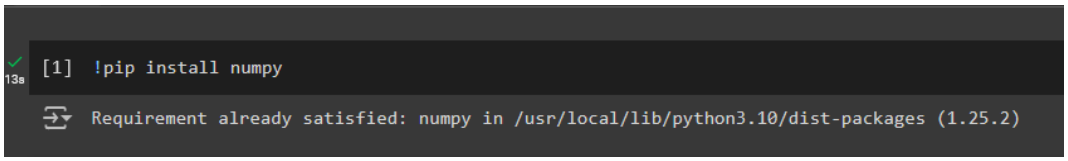

Figura 3: Instalación de la biblioteca *NumPy* en *Google Colab*.

Por último, cargaremos la biblioteca y comprobaremos el correcto funcionamiento de la biblioteca. Para la comprobación, hemos decidido realizar la multiplicación de 2 pequeñas matrices de tamaño 2×2 (Figura 4):

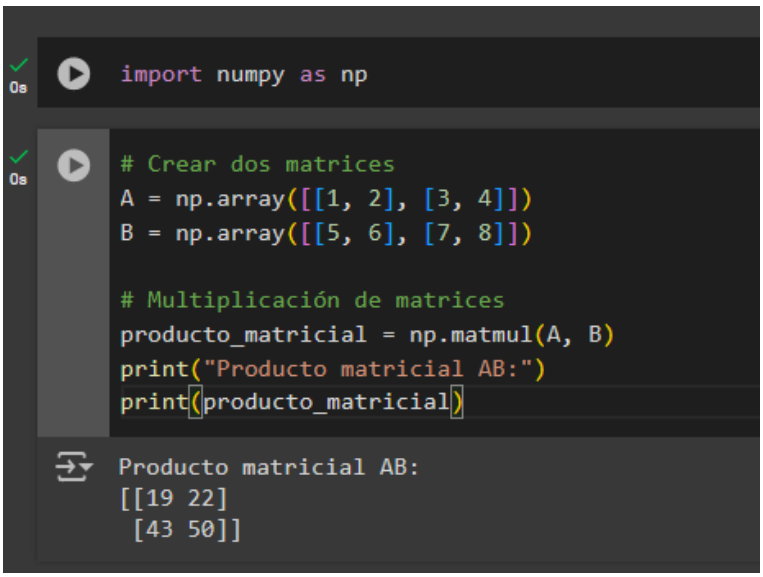

Figura 4: Carga de la biblioteca *Numpy* y multiplicación de matrices.

Anaconda

Antes de empezar con el tutorial de la instalación de la biblioteca *NumPy* en *Anaconda* tenemos que saber qué es y cómo se debe instalar este último.

Anaconda es una distribución de *Python* que incluye una amplia variedad de paquetes y herramientas útiles para el desarrollo y el campo de la ciencia de datos, como pueden ser *R* o *Spyder*.

Figura 5: Logo de la distribución *Anaconda*.

Una vez sabemos qué es y para qué sirve *Anaconda*, podemos pasar a explicar cómo debemos realizar su instalación.

En primer lugar, debemos descargar el instalador *Anaconda* adecuado para nuestro sistema operativo, ya sea *Linux, macOS o Windows* [8], como se muestra en la Figura 6.

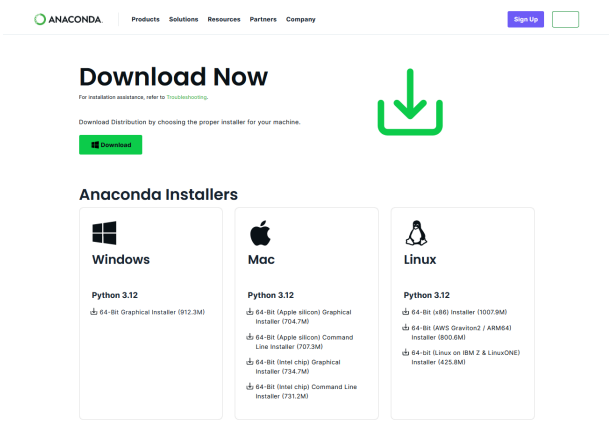

Figura 6: Página web para descargar el instalador *Anaconda*.

Una vez hemos descargado el instalador, debemos ejecutarlo. Durante la instalación, aceptamos los términos de la licencia y seleccionamos las opciones predeterminadas a menos que tengas requisitos específicos (Figura 7).

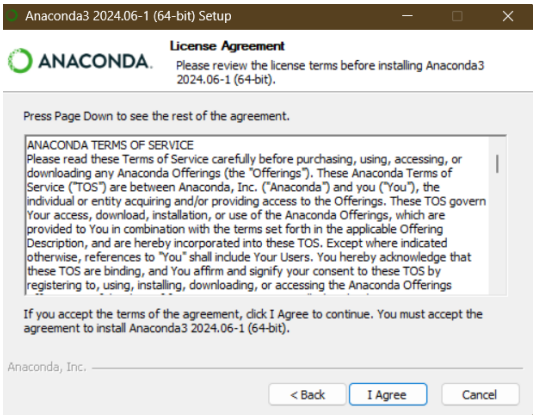

Figura 7: Términos de la licencia de *Anaconda*.

Completada la instalación, abrimos *Anaconda Navigator*. Para empezar, podemos crear un nuevo entorno virtual para mantener los proyectos organizados y evitar conflictos de paquetes. Para ello, debemos desplazarnos en la pestaña "*Environments*" y hacer clic en "*Create*". Le definimos un nombre al entorno y seleccionamos la versión de *Python* que mejor se ajuste a nuestras necesidades (Figura 8). Además, se puede instalar paquetes adicionales que se encuentran en la lista disponible o instalarlos más tarde desde la terminal.

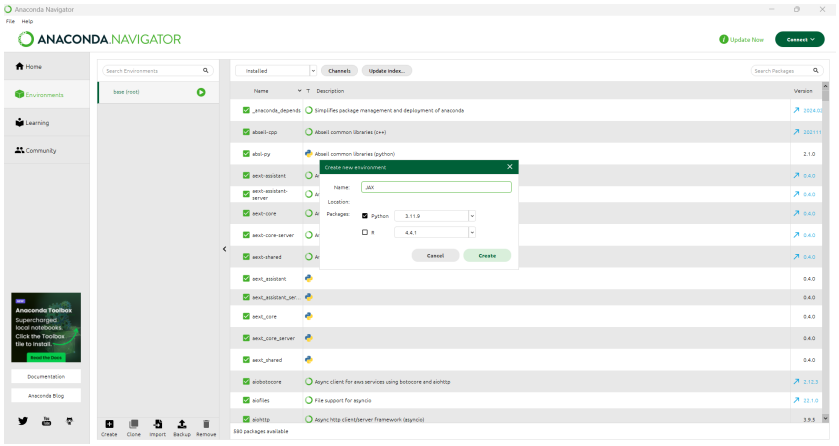

Figura 8: Creación de un entorno en *Anaconda*.

Seguidamente debemos situarnos en la pestaña "*Home*" para instalar *Spyder*. Abrimos *Spyder* y desde la terminal integrada, abrimos una nueva consola y escribimos "*!pip install numpy*" (Figura 9).

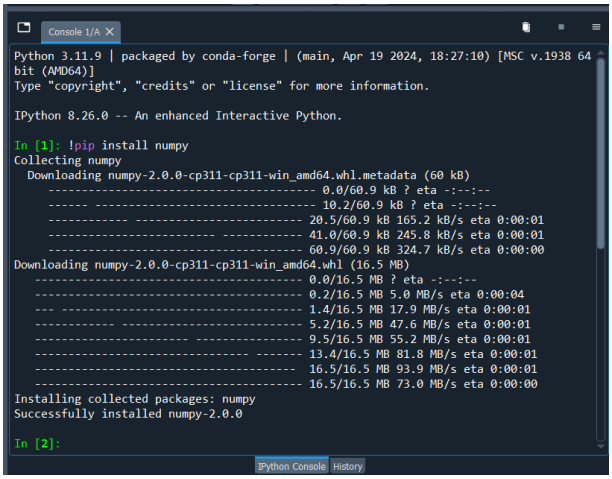

Figura 9: Instalación de *NumPy* en *Spyder*.

Por último, comprobamos si se ha instalado correctamente, para ello realizamos la misma prueba que hemos realizado en *Google Colab* (Figura 10).

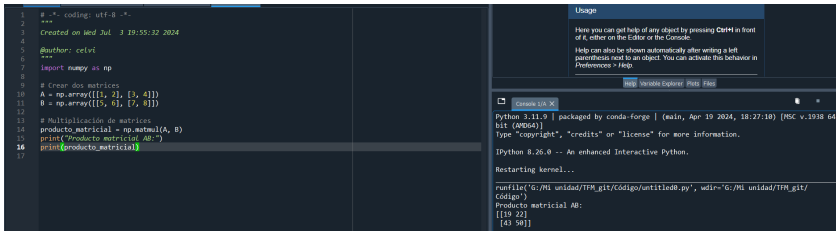

Figura 10: Multiplicación de matrices en *Spyder* con *NumPy*.

2.2 Cupy

Cupy [7] es una biblioteca diseñada para realizar cálculos numéricos en *GPU* mediante *CUDA*. La interfaz es similar a la de *NumPy*, permitiendo a los usuarios migrar los códigos de *NumPy* a *CuPy*. Algunas de las características más importantes de esta biblioteca son:

- **Compatibilidad con *NumPy*:** *CuPy* tiene una API casi idéntica a la de *NumPy*, lo que facilita la migración de código de *NumPy* a *CuPy*.
- **Aceleración por *GPU*:** Utiliza *CUDA* para ejecutar operaciones en *GPU*, logrando mejoras significativas en rendimiento.
- **Soporte Extenso de Funciones:** Incluye operaciones aritméticas, lógicas o manejo de arrays dispersos.
- **Interoperabilidad con Bibliotecas *CUDA*:** Integración con *cuBLAS, cuDNN, cuSPARSE* y *cuFFT* para optimizar operaciones específicas.
- **API de Funciones Universales:** Soporta la creación y uso de funciones para vectorizar operaciones personalizadas en *GPU*.
- **Memoria Unificada:** Facilita la transferencia de datos entre *CPU* y *GPU* con funciones.

2.2.1. Instalación

En este apartado, vamos a explicar cómo instalamos *CuPy* en el entorno *Google Colab*, dado que podemos seleccionar el tipo de acelerador de *Hardware* y en este caso, solo necesitamos ejecutar en *GPU*.

Los primeros pasos a seguir quedan explicados en la Sección 2.1 NumPy. Una vez tenemos creado el *notebook* podemos pasar a instalar la biblioteca *CuPy* y comprobar su funcionamiento.

Primeramente, vamos a instalar la biblioteca. Para ello seguiremos los siguientes pasos (Figura 11):

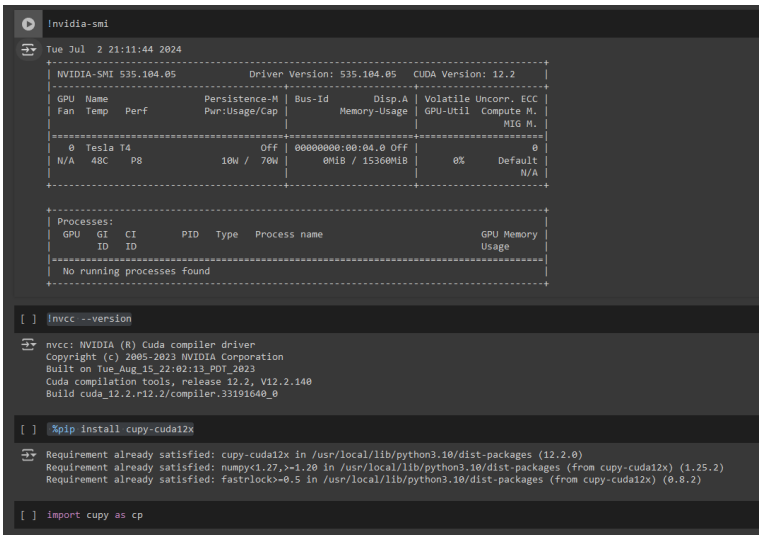

Figura 11: Instalación de biblioteca *CuPy* en *Google Colab*.

1. **Verificación de la presencia de *GPU* y sus *drivers*:** !nvidia-smi
2. **Verificamos la versión de *CUDA* instalada:** !nvcc −−version
3. **Instalar *CuPy* con soporte para la versión específica de *CUDA*:** %pip install cupy-cuda12x
4. **Importar y usar *CuPy*:** import cupy as cp

Ahora nos queda el último paso, la comprobación del correcto funcionamiento de la biblioteca. Para ello, hemos decidido realizar la multiplicación de 2 pequeñas matrices de tamaño 2×2 (Figura 12):

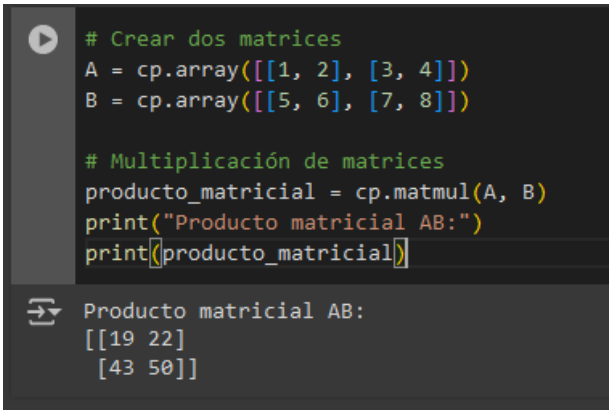

Figura 12: Multiplicación de matrices con *CuPy* en *Google Colab*.

2.3 Numba

Numba [3] es una biblioteca para compilar *just-in-time* (*JIT*) de funciones *Python* y *NumPy* para ejecución rápida en *CPU* o *GPU*. Las características más importantes de *Numba* son:

- **Compilación *Just-in-Time (JIT)*:** *Numba* compila las funciones de *Python* en código máquina.
- **Soporte para *CPU* y *GPU*:** *Numba* puede compilar las funciones tanto en *CPU* como en *GPU*.
- **Optimización:** Utilización de *"@jit"* para emplear *JIT* en funciones especificas. De esta forma, controlamos qué funciones se optimizan.
- **Soporte para *NumPy* y arrays multidimensionales:** *Numba* es compatible con las operaciones de *NumPy* y puedes optimizar las funciones que trabajan con *arrays* multidimensionales.
- **Paralelización automática:** *Numba* puede automatizar parte del proceso para ejecutar bucles en paralelo en *CPU* o *GPU*. De esta forma, reducimos el tiempo de ejecución.
- **Facilidad de integración:** Es fácil integrar *Numba* en los diferentes proyectos de *Python*, dado que no requiere grandes cambios en la estructura del código.

2.3.1. Instalación

La instalación de *Numba* se realizara en dos entornos diferentes, *Google Colab* y *Anaconda*.

Google Colab

Los primeros pasos a seguir quedan explicados en la Sección 2.1.1. Instalación. Una vez tenemos creado el *notebook* podemos pasar a instalar la biblioteca *Numba* y comprobar su funcionamiento.

En primer lugar, es necesario instalar la biblioteca *Numba*. A tal efecto, procederemos de la siguiente forma (Figura 13):

1. **Instalar *Numba*:** !pip install numba
2. **Importar y usar *Numba*:** import numba

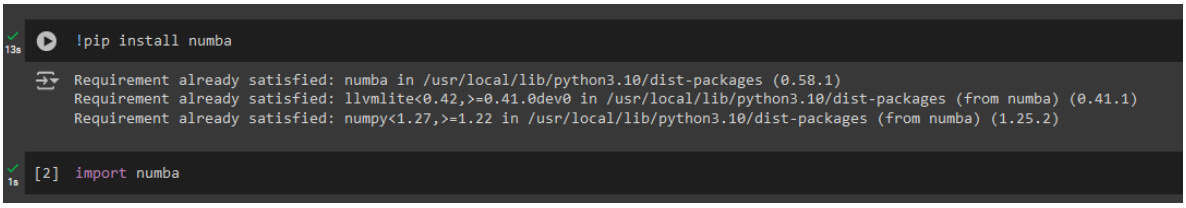

Figura 13: Instalación de la biblioteca *Numba* en *Google Colab*.

Finalmente, queda confirmar que la biblioteca está funcionando bien. Para lograrlo, hemos decidido multiplicar dos matrices pequeñas de 2×2 (Figura 14):

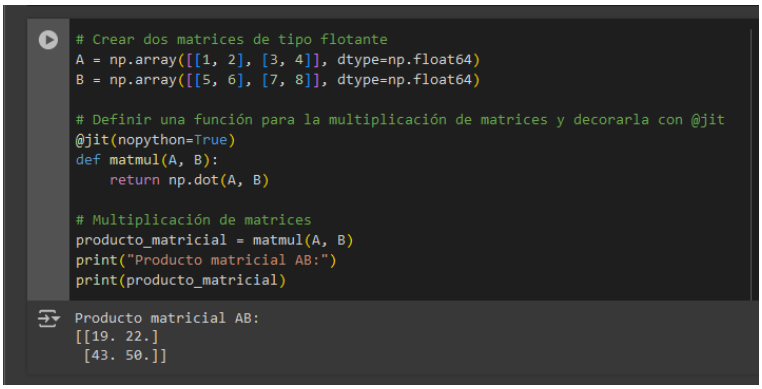

Figura 14: Multiplicación de matrices con *Numba* en *Google Colab*.

Anaconda

El detalle de los primeros pasos está en la Sección 2.1 NumPy, en la sección de *Anaconda*. Necesitamos tener abierto el programa *Spyder* para realizar la instalación de la biblioteca *Numba*.

Desde la terminal integrada, abrimos una nueva consola y escribimos "!pip install numba" (Figura 15):

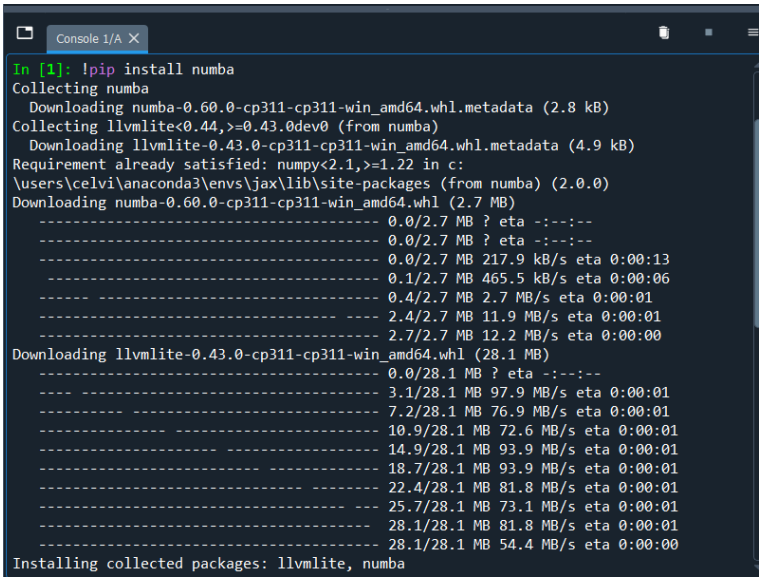

Figura 15: Instalación de la biblioteca *Numba* en *Anaconda*.

Para acabar, verificamos la correcta instalación a través de la multiplicación de dos matrices de tamaño 2×2 (Figura 16):

Figura 16: Multiplicación de matrices con *Numba* en *Anaconda*.

2.4 JAX

JAX [2] es una biblioteca de *Python*, desarrollada por *Google* que combina la flexibilidad de *NumPy* con la potencia de los compiladores JIT para la aceleración de cálculos numéricos. Esta diseñado principalmente para ser más eficiente y escalable. Las características más importantes de *JAX* son:

- **Compilación *Just-In-time (JIT)*:** Usa "@jit" para compilar funciones en código máquina y mejorar el rendimiento.
- **Compatibilidad con *NumPy*:** API similar a *NumPy*, de esta forma permite cambiar de forma sencilla el código existente.
- **Soporte para *GPU* y *TPU*:** Ejecución de cálculos en *CPU*, *GPU* y *TPU* sin cambiar nuestro código.
- **Transformaciones Funcionales:** Funciones como *'grad' o 'vmap'* para realizar diferenciación de forma automática y vectorizar.
- **Primitivas de Bajo Nivel:** Los usuarios pueden definir sus propias primitivas a bajo nivel para operaciones personalizadas.

2.4.1. Instalación

Procederemos a instalar *JAX* en tres entornos *Anaconda* y *Kaggle/Google Colab*.

Anaconda

La explicación de los primeros pasos está en la Sección 2.1 NumPy, en la sección de *Anaconda*. Necesitamos tener el programa *Spyder* abierto para instalar la biblioteca *JAX*.

Desde la terminal integrada, abrimos una nueva consola y seguimos los siguientes pasos (Figuras 17 y 18):

1. **Instalar primeramente la biblioteca *jaxlib*:** pip install jaxlib
2. **Instalar la biblioteca *JAX*:** pip install jax

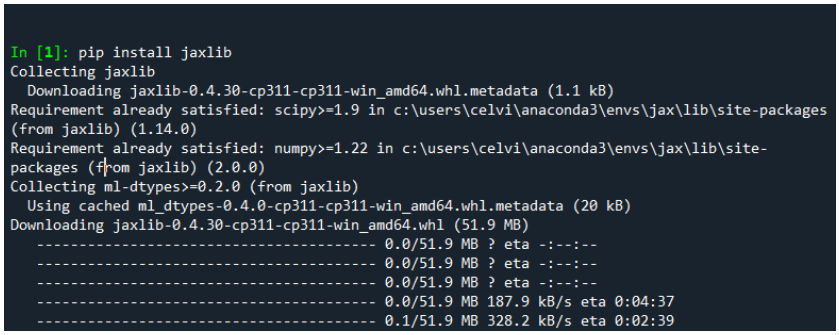

Figura 17: Instalación de la biblioteca *jaxlib* en *Anaconda*.

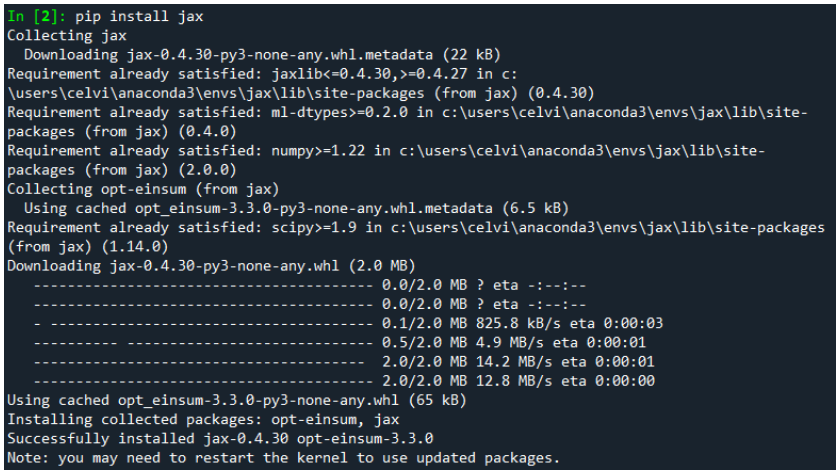

Figura 18: Instalación de la biblioteca *JAX* en *Anaconda*.

Se debe seguir el orden indicado, dado que en caso de no hacerlo no será posible utilizar de forma correcta la biblioteca *JAX* y se deberá volver a realizar la instalación desde cero.

Por último, ejecutamos la comprobación correspondiente con la misma prueba que hemos realizado con anterioridad, es decir, multiplicar dos matrices de 2×2 (Figura 19):

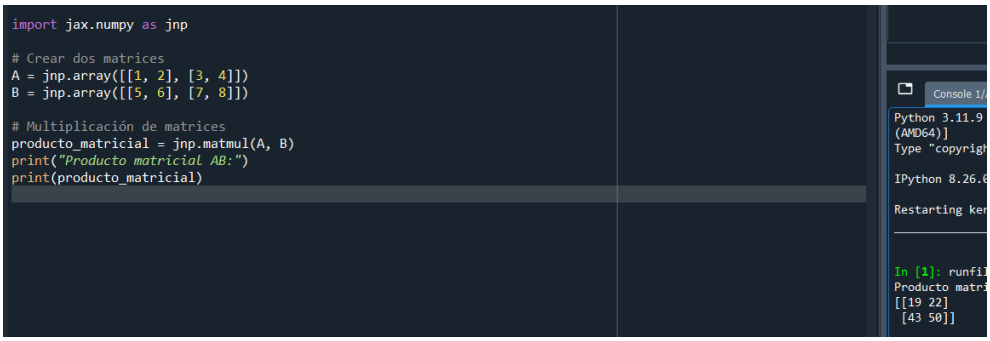

Figura 19: Multiplicación de matrices con *JAX* en *Anaconda*.

Google Colab/Kaggle

Los primeros pasos a seguir quedan explicados en la Sección 2.1 NumPy. Una vez tenemos creado el *notebook* podemos pasar a instalar la biblioteca *JAX* y comprobar su funcionamiento.

El primer paso es la instalación de la biblioteca *JAX*. A tal efecto, ejecutaremos el comando *"!pip install jax jaxlib"* (Figura 20):

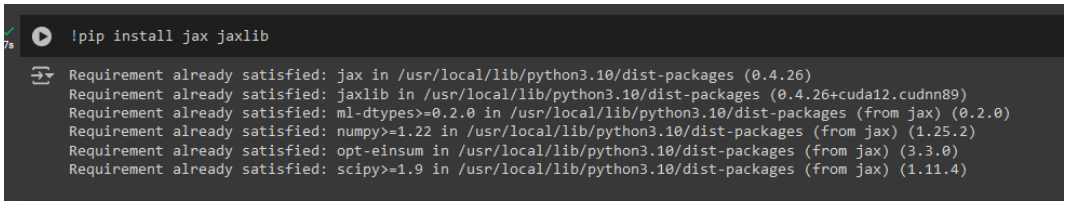

Figura 20: Instalación de la biblioteca *JAX* en *Google Colab/Kaggle*.

Acabamos con la comprobación de la correcta instalación de la biblioteca, para ello multiplicamos dos matrices 2×2 (Figura 21):

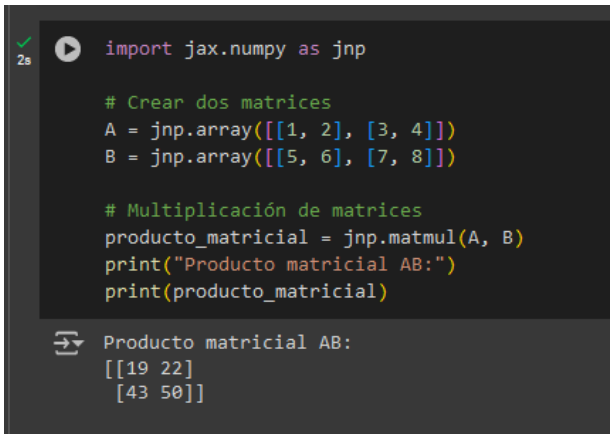

Figura 21: Multiplicación de matrices con *JAX* en *Google Colab/Kaggle*.

3 Estado del arte

Hoy en día encontramos que el campo de las redes neuronales ha experimentado avances significativos gracias a diferentes factores. Algunos de los factores más importantes son el desarrollo de *frameworks* y herramientas que permiten la implementación eficiente y escalable de modelos complejos.

Uno de los *frameworks* emergentes es *JAX*, que ha ganado popularidad los últimos años debido a su semejanza con *Numpy* y su alto rendimiento computacional.

3.1 Implementación de núcleos computacionales

Los núcleos computacionales, también conocido como unidades fundamentales de procesamiento dentro de *CPUs*, *GPUs* o *TPUs*; desempeñan un papel crucial en la ejecución eficiente de los algoritmos en las redes neuronales. En el campo de la implementación de redes neuronales con *JAX*, la optimización y la distribución correcta de los núcleos son elementos críticos para maximizar el rendimiento y la velocidad de entrenamiento.

3.2 Innovaciones en *JAX*

JAX es conocido principalmente por su gran compatibilidad y cálculos puros, que simplifican el proceso de desarrollo de modelos de aprendizaje profundo. Estas ventajas permiten a los usuarios implementar y experimentar con arquitecturas de redes neuronales más avanzadas de una forma eficiente.

3.3 Desafíos actuales y investigaciones previas

A pesar de los avances, aun existen desafíos en la implementación de núcleos computacionales que debemos tener en cuenta, como puede ser la distribución de la carga computacional entre los diferentes tipos de núcleos (*CPU*, *GPU* y *TPU*) o la optimización del código.

En nuestro caso, nos centraremos en este último caso, es decir, la optimización del código a partir de la comparación de diversas bibliotecas de *Python*.

A continuación, se van a detallar una serie de investigaciones en el ámbito de la comparación de bibliotecas para obtener el mejor rendimiento en el menor tiempo para el cálculo de operaciones matemáticas en los diferentes núcleos computacionales.

3.3.1 Modelo 2D de propagación de incendios forestales en *Python*: de *NumPy* a *CuPy*

El primer artículo [11], se centra en la prevención de incendios forestales en Chile. Para ello se pretende buscar las mejores herramientas de simulación.

Con el objetivo de cumplir su hipótesis inicial plantean usar la metodología descrita a continuación.

1. **Modelo Matemático:** Se usa un modelo lineal, que separa las dependencias espaciales y temporales, de esta forma se logra obtener una aproximación independiente entre clases numéricas.

2. **Implementación en *CPU* y *GPU*:** Se implementa *NumPy* en *CPU*, y *CuPy* en *GPU*. Llegando a demostrar una mejora en el rendimiento en comparación con las implementaciones en *CPU*.

3. **Evaluación del Algoritmo:** Se realizan análisis de estabilidad y convergencia usando diferentes métodos numéricos, como son *Euler* o *RK4*.

Se concluye que la implementación en *CuPy* demuestra ser una herramienta valiosa para mejorar el rendimiento sin añadir complejidad al usar *frameworks* más especializados como *CUDA* o *OpenCL*.

3.3.2 Influencia de la cantidad de datos, tipo de datos y paquetes de implementación en la codificación de la *GPU*

En el segundo artículo [14], se estudia cómo afectan los tipos y la cantidad de datos, así como los paquetes implementados al rendimiento del procesamiento en *GPU*.

Con este fin se plantea la siguiente metodología:

1. **Cantidad de datos:** Se prueban diferentes tamaños de conjuntos de datos para evaluar cómo afecta al rendimiento de la *GPU*.

2. **Tipo de Datos:** Se experimenta con diversos tipos de datos para determinar su impacto en el rendimiento de la *GPU*.

3. **Paquetes de Implementación:** Se analizan varios paquetes de *software* disponibles para la programación en *GPU*, incluyendo bibliotecas populares y entornos de desarrollo específicos, como son *Numba* y *CuPy*.

Se concluye que para maximizar el rendimiento en *GPU*, es fundamental considerar de forma precavida la cantidad y el tipo de datos a utilizar, más adecuada para las necesidades específicas del proyecto.

Por otra parte, tenemos la comparación de los paquetes de implementación. En este caso, se concluye que ambos paquetes tienen ventajas significativas para la computación en *GPU*, pero que en su mayoría *Numba* tiende a ser más rápido que *CuPy* en el manejo de grandes cantidades de datos.

3.3.3 Comparación de compiladores *Python*

La tercera publicación se compone de dos secciones. La primera sección [12] aborda la comparación de diversos compiladores *Python* que mejoran el rendimiento en el campo de la investigación científica en finanzas cuantitativas.

La segunda sección [13] profundiza en el rendimiento de varios compiladores *JIT* en el campo de la regresión lineal y otras operaciones de *Machine Learning*.

Conjuntamente se quiere estudiar varios compiladores *JIT* y *frameworks* que destacan por mejorar el rendimiento de *Python* en aplicaciones críticas de computación científica.

La metodología implementada es la observada a continuación:

1. **Selección de Bibliotecas y *Frameworks*:** Se seleccionan herramientas clave como *Cython*, *Numba*, *JAX*, *Tensorflow* y *PyTorch* basadas en su popularidad y capacidades técnicas.

2. **Diseño de Experimentos:** Formulación de casos de prueba que cubren una variedad de escenarios, como son una regresión lineal. De esta forma, implementa las operaciones utilizando cada biblioteca o *framework* estudiado.

3. **Ejecución y Evaluación de Pruebas:** Medición del tiempo de ejecución y eficiencia energética para cada implementación.

Antes de llegar a las conclusiones, vamos a analizar los resultados obtenidos en esta publicación. Analizaremos las librerías o *frameworks* más relevantes.

- ***Cython*:** Mejora al integrar el código *Python* con extensiones en C.
- ***Numba*:** Optimización de forma automática mediante compilación *JIT*, adecuado para acelerar funciones numéricas sin reescribir código.
- ***JAX*:** Alto rendimiento para *CPU* y *GPU*, optimizando de esta forma los cálculos de álgebra lineal.
- ***TensorFlow* y *PyTorch*:** *Frameworks* robustos que tienen soporte para *GPU* y técnicas avanzadas de optimización en *JIT*.

Por último, concluye que la elección de compilador o *framework* depende de las necesidades del proyecto. Pero, *Cython* y *Numba* son excelentes para *CPU*. Mientras que *JAX*, *Tensorflow* y *PyTprch* son buenas para *GPU*.

3.3.4 Evaluación comparativa de *JAX* vs. *NumPy*

La cuarta publicación [10], trata de comparar el rendimiento de las bibliotecas *JAX* y *NumPy* en el campo del cálculo científico.

A continuación se detalla el proceso metodológico seguido en este estudio:

1. **Definición del Problema y Objetivos:** Comparación del rendimiento entre *JAX* y *NumPy*.

2. **Selección de Casos de Prueba:** Los casos de pruebas seleccionados son la multiplicación de grandes matrices y cálculos de gradientes.

3. **Configuración del Entorno de Pruebas:** Se usa *Google Colab* para ejecutar pruebas con *CPU, TPU* y *GPU*.

4. **Medición del Rendimiento:** Evaluamos el tiempo de ejecución y eficiencia en diferentes *hardware*. Además, de la comparación de resultados de *JAX* con y sin el compilador *JIT*.

JAX es mejor que *NumPy* en muchos aspectos, especialmente en el caso del aprendizaje profundo y la computación en *hardware* acelerado. Pero, encontramos que *NumPy* es adecuado para tareas simples. Concluyendo que la elección entre *JAX* y *NumPy* debe basarse en el tamaño del problema y el entorno de ejecución disponible.

3.4 Análisis y comparaciones

Después de haber visto varios artículos y publicaciones que comparan bibliotecas y *frameworks* de *Python* para la mejora del rendimiento en el cálculo de operaciones en los núcleos computacionales, se puede apreciar que un elemento clave es el tamaño del conjunto de datos y el entorno de ejecución.

La biblioteca que más se estudia es *NumPy* dado que es la más conocida y empleada en el campo de la computación científica, en concreto, las redes neuronales.

Para finalizar, podemos destacar que el entorno más empleado es *GPU*, y la mejor biblioteca en dicho entorno es *CuPy*, dado que se obtiene un menor tiempo y por lo tanto, un mejor rendimiento. Esto se debe gracias al tipo de transferencia de datos y el uso de bibliotecas aceleradas, como son *cuBLAS* o *cuDNN*.

4 Metodología

Para realizar este proyecto, se han seguido una serie de pasos que se detallan a continuación (Figura 22):

Figura 22: Pasos a seguir.

4.1 Requisitos

Se ha hecho una revisión del estado del arte en Implementación de Núcleos Computacionales para Redes Neuronales con *JAX* (Sección 3 Estado del arte) con el objetivo de identificar elementos comunes y tendencias clave.

Además, se ha explorado una variedad de bibliotecas y entornos de desarrollo para facilitar la implementación y experimentación con diferentes núcleos computacionales (Sección 2 Marco teórico). Después de evaluar varias opciones, se ha decidido utilizar *Google Colab*, *Kaggle* y *Anaconda*. Estas tres plataformas fueron seleccionadas por su facilidad de uso y, en el caso de *Kaggle* y *Google Colab*, por la capacidad de seleccionar el núcleo computacional deseado para ejecutar los experimentos.

Para la implementación y comparación de librerías en *CPU* hacemos uso del entorno *Anaconda*. En el caso de *GPU* usamos *Google Colab*. Y por último, para la implementación en *TPU* empleamos el entorno *Kaggle*.

La selección de un entorno diferente a *Google Colab* en el caso de *TPU* depende de un factor principal que describiremos a continuación.

Kaggle ofrece un periodo de duración de sesiones más largo y menos restricciones en la duración continua de uso de *TPUs* en comparación con *Google Colab*, lo que permite sesiones más estables y prolongadas sin interrupciones.

Asimismo, tanto *Kaggle* como *Google Colab* permiten la ejecución de solo un *notebook* a la vez. Por lo tanto, distribuir la carga entre estas plataformas para diferentes núcleos computacionales resulta en una elección más eficiente y manejable.

4.2 Selección de bibliotecas

Luego de una exhaustiva búsqueda y comparación de las diferentes bibliotecas de *Python* utilizadas en el ámbito del cómputo científico y la aceleración de hardware (Sección 2 Marco teórico), tales como *NumPy*, *SymPy* o *CuPy*; nos hemos decantado por las bibliotecas más empleadas y desarrolladas. Estas son *NumPy*, *CuPy*, *Numba*, y *JAX*, siendo esta última la librería que queremos estudiar en detalle.

Una vez expuestas y estudiadas las principales ventajas de las bibliotecas seleccionadas, realizamos una pequeña comparativa para analizar sus diferencias y similitudes:

- *NumPy, CuPy, Numba* y *JAX* ofrecen capacidades complementarias para el cómputo científico y la aceleración de *hardware*.
- *NumPy* proporciona un excelente rendimiento para realizar operaciones matemáticas en *CPU*.
- *CuPy*, en cambio, está diseñada para ejecutarse en *GPU*, ofreciendo una interfaz similar a *NumPy* pero optimizada para el cómputo paralelo.
- *Numba* ofrece una solución flexible para compilar y optimizar código en *CPU* y *GPU* gracias al compilar *LLVM* [5].
- Por último, *JAX* destaca por sus avanzadas capacidades en diferenciación automática y la ejecución en *hardware* acelerado, incluyendo *CPU, GPU* y *TPU*.

En conclusión, la combinación y la comparación de estas bibliotecas permite seleccionar la mejor opción dependiendo de las necesidades específicas, como pueden ser el núcleo de computo. De esta manera, optimizamos el rendimiento y la eficiencia de la aplicación científica.

4.3 Estudio de bibliotecas

Una vez seleccionadas las bibliotecas que vamos a estudiar, es fundamental conocer su funcionamiento detallado. A continuación, se realizó un análisis en profundidad de cada una, destacando sus características, ventajas y ejemplos de uso prácticos, enfocándonos principalmente en las funciones clave utilizadas en las capas de las redes neuronales, como la multiplicación de matrices.

Algunas de las funciones más importantes de las bibliotecas estudiadas son: la definición de *arrays*, creación de matrices aleatorias, y la multiplicación y suma de matrices.

4.4 Implementación

La implementación del sistema (Sección 5 Implementación) se ha llevado a cabo siguiendo una serie de pasos estructurados para asegurarnos de la correcta integración y funcionalidad de las bibliotecas empleadas. Este proceso incluye la definición de funciones clave, creación y multiplicación de matrices.

Los pasos que hemos seguido en este punto son:

1. **Preparación del Entorno:** Antes de iniciar con la implementación, debemos configurar el entorno donde vamos a proceder con el desarrollo, cómo puede ser la instalación y carga de las bibliotecas o la elección del núcleo computacional.

2. **Definición de Funciones y Estructuras de Datos:** Definimos la estructura de datos que necesitemos, matrices y *arrays*, utilizando las bibliotecas seleccionadas.

3. **Implementación de Operaciones Matemáticas:** La principal operación matemática básica, la multiplicación, se implementó utilizando las funciones propias de cada biblioteca.

4. **Optimización del Código:** La optimización del código, se implementó para la biblioteca *JAX* y *Numba*, utilizando el compilador *JIT*, lo que permite acelerar las operaciones mediante la compilación del código *Python* a código máquina en tiempo de ejecución.

5. **Validación y Verificación:** se realizan pruebas exhaustivas para verificar el rendimiento y la exactitud de las implementaciones. Se comparan los resultados obtenidos en el rendimiento para cada una de las bibliotecas empleadas y configuraciones de *hardware*.

4.5 Testeo

El proceso de testeo nos ayuda a verificar el correcto funcionamiento del sistema de funciones. Así como asegurar que los resultados sean los esperados y precisos. A continuación, se detallan las etapas y métodos de testeo empleados.

4.5.1. Tipos de testeos realizados

Las tipos de pruebas realizadas para cubrir los diferentes aspectos del sistema son:

- **Pruebas Unitarias:** Validación de las funciones de forma individual.
- **Pruebas de Rendimiento:** Evaluación de eficiencia y escalabilidad.
- **Pruebas de Validación:** Comprobación de la exactitud de los resultados comparándolos con referencias conocidas.

4.6 Resultados

Por último, se estudiaron los resultados obtenidos (Sección 6 Experimentos) tras la implementación y pruebas de las bibliotecas estudiadas, *NumPy*, *JAX*, *Numba* y *CuPy*; para operaciones matemáticas. Se realizaron diversas pruebas para evaluar el rendimiento y la precisión de cada una de las bibliotecas en diferentes escenarios.

4.6.1 Precisión

Para estudiar la precisión de los cálculos, se comparan los resultados de la multiplicación de matrices obtenidos con cada biblioteca.

Además, se emplea el uso de *float16* y *float32* para evaluar cómo la precisión de los cálculos puede variar con diferentes representaciones en coma flotante. Las operaciones de multiplicación de matrices se llevaron a cabo con ambas precisiones, y los resultados se compararon en términos de tiempo y rendimiento.

4.6.2 Rendimiento

Medimos el tiempo de ejecución para cada una de las multiplicaciones de matrices empleando cada biblioteca. Las pruebas se realizaron en un entorno con capacidad *GPU*, *CPU* y *TPU*.

4.6.3 Escalabilidad

Se evaluó la escalabilidad de cada biblioteca incrementando el tamaño de las matrices, para ello empleamos 20 conjuntos diferentes de tamaño de matrices que definimos en *MNK*. Estos tamaños son los resultantes de realizar una transformación a los datos de entrada de las capas convolucionales del modelo ResNet-50 v1.5.

Los tiempos de ejecución y el rendimiento se registraron y analizaron a través de un gráfico de barras para cada uno de los núcleos computacionales.

5 Implementación

5.1 Núcleos computacionales

Los núcleos computacionales empleados en nuestro estudio son *CPU*, *GPU* y *TPU*. A continuación, se describen el tipo usado y las características de cada una.

- La *CPU* es la más versátil de las tres opciones. Está diseñada para manejar una amplia variedad de tareas generales con gran eficiencia en multitarea y tiempos de respuesta rápidos.
- La *GPU* está optimizada de tal forma que es ideal para tareas que requieren la ejecución simultánea de numerosas operaciones. Permite realizar cálculos intensivos de manera eficiente, que resulta en una excelente opción para renderizado de gráficos y entrenar redes neuronales.
- La *TPU* es *hardware* especializado para acelerar tareas de inteligencia artificial y aprendizaje automático. Y orientado a optimizar las operaciones tensoriales, proporcionando un rendimiento superior en la ejecución de modelos de redes neuronales.

5.1.1 *CPU*

La *CPU* utilizada en nuestro estudio es un *Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz* [1]. Este procesador pertenece a la octava generación de procesadores *Intel* y está diseñada para ofrecer un equilibrio óptimo entre rendimiento y eficiencia energética.

Las características de la *CPU* son las siguientes:

- **Frecuencia de reloj:**
  - Frecuencia anunciada: 1.8000 GHz
- **Arquitectura:** X86_64 (AMD64)
- **Fabricante:** *GenuineIntel*
- **Número de núcleos:** 8

- **Memoria caché:**
  - *Caché* L2: MB (1048576 *bytes*)
  - *Caché* L3: MB (8388608 *bytes*)
- **Información técnica:**
  - Familia: 6
  - Modelo: 142
  - *Stepping:* 10

Esta *CPU* es eficiente para el manejo de una variedad de tareas generales y eficaz en entornos multitarea, proporcionando tiempos de respuesta rápidos y un rendimiento confiable para aplicaciones de uso diario y computo científico.

5.1.2 *GPU*

La *GPU* utilizada en el estudio es una *GPU NVIDIA Tesla T4* [6]. Esta tarjeta gráfica se basa en la arquitectura *Turing* de *NVIDIA* y está diseñada para ofrecer un alto rendimiento en tareas de computación y aprendizaje automático.

La *Tesla T4* proporciona hasta 16GB de memoria *GDDR6* y es conocida por su eficiencia energética, siendo ideal para entornos de procesamiento intensivo de datos y aplicaciones de inteligencia artificial.

Las características de la *GPU* son las siguientes:

- **Uso de energía:**
  - Máximo: 70W
- **Memoria de la *GPU*:**
  - Total: 15360MiB
- **Versión del controlador:** 535.104.05
- **Versión de *CUDA*:** 12.2

Esta *GPU* es ideal para entornos de computación de alto rendimiento, ofreciendo una capacidad de memoria alto y bajo consumo de energía en relación con su potencia de cálculo.

Es adecuada para ejecutar modelos de aprendizaje profundo, análisis de datos a gran escala y otras aplicaciones que requieren un procesamiento gráfico intensivo.

5.1.3 *TPU*

La *TPU* utilizada en el estudio es una *TPU v3-8* [9]. Esta unidad de procesamiento tensorial está diseñada para manejar tareas de aprendizaje profundo. Se conoce por su alto rendimiento y eficiencia.

La *TPU v3-8* cuenta con 4 chips, cada uno con 2 núcleos, sumando un total de 8 núcleos de *TPU*. Cada núcleo está equipado con *hardware* dedicado para multiplicación de matrices de 128×128, lo que acelera significativamente las cargas de trabajo de aprendizaje automático.

Las características de la *TPU* son las siguientes:

- **Memoria de la *TPU*:** 128GB de memoria de alta velocidad.
- ***Batch Size*:** 128 elementos por núcleo.
- ***Hardware*:**
  - 4 chips de *TPU*, 8 núcleos en total.
  - Multiplicadores de matrices de 128×128.
- **Soporte de *software*:**
  - *Tensorflow* 2.1 o superior.
  - Compatibilidad con *Keras* y bucles de entrenamiento personalizados.
  - Soporte para *TFRecordDataset* y optimización de canalización de datos.

Esta *TPU* es ideal para entornos de computación de alto rendimiento, ofreciendo una gran capacidad de memoria y procesamiento eficiente de grandes modelos de aprendizaje profundo.

Es ideal para la ejecución de modelos de redes neuronales convolucionales, análisis de datos a gran escala y otras aplicaciones que requieren un procesamiento tensorial intensivo.

5.2 Multiplicación de matrices

La multiplicación de matrices se ha implementado en las cuatro bibliotecas de *Python* que queremos comparar. Para determinar cuál de estas bibliotecas es mejor,

utilizamos el mismo código en diferentes entornos computacionales, distribuidos en tres plataformas: *Anaconda*, *Kaggle* y *Google Colab*.

Las principales partes del código son las siguientes:

- Carga de las bibliotecas.
- Definición de las dimensiones de las matrices, el número de repeticiones y las listas donde almacenamos los resultados obtenidos.
- Creación aleatoria de las matrices.
- Primera iteración para inicializar la carga de datos.
- Cálculo de las iteraciones subsecuentes.
- Medición del tiempo por iteración.
- Cálculo de *GFLOPS*.

5.2.1 Código

A continuación, exploraremos el código empleado de manera general, sin entrar en detalles específicos para cada una de las bibliotecas, excepto cuando exista alguna diferencia notable que deba ser destacada. Por esta razón, utilizaremos *xx* en sustitución del nombre de la biblioteca empleada y *yy* en sustitución del tipo de núcleo computacional (*GPU*, *CPU* o *TPU*).

Primeramente, una vez instalamos las bibliotecas (Sección 2 Marco teórico) pasamos a cargarlas, de esta forma podremos empezar a realizar el cálculo para estudiar el rendimiento de las bibliotecas.

```
import numpy as np
import jax.numpy as jnp
import time
from jax import jit
import jax
from jax import random
from numba import njit, prange
import cupy as cp
```

Cargamos las dimensiones de las matrices, el número de repeticiones y las listas donde vamos a almacenar los resultados. Donde s hace referencia al tamaño del lote, bs, y n hace referencia al número de repeticiones.

```
#Establecemos las dimensiones de las matrices.
bs = s
MNK=[
            [1000, 1000, 1000], #Capa de inicializacion.
            [12544*bs, 64, 147],
            [3136*bs, 64, 64],
            [3136*bs, 64, 576],
            [3136*bs, 256, 64],
            [3136*bs, 64, 256],
            [3136*bs, 128, 256],
            [784*bs, 128, 1152],
            [784*bs, 512, 128],
            [784*bs, 512, 256],
            [784*bs, 128, 512],
            [784*bs, 256, 512],
            [196*bs, 256, 2304],
            [196*bs, 1024, 256],
            [196*bs, 1024, 512],
            [196*bs, 256, 1024],
            [196*bs, 512, 1024],
            [49*bs, 512, 4608],
            [49*bs, 2048, 512],
            [49*bs, 2048, 1024],
            [49*bs, 512, 2048],]
```

El uso de una capa de inicialización se debe a varias razones, destacando principalmente tres, las cuales explicaremos a continuación.

1. **Compilación *JIT* en las bibliotecas *JAX* y *Numba*:** uso del compilador *JIT* Durante la primera ejecución, la función llamada se compila, lo que consume tiempo y recursos.

2. **Inicialización de Datos y Modelos:** La carga y preparación de los datos, modelos o recursos necesarios se realizan durante la primera iteración. Este proceso es costoso y ocurre solo una vez al inicio.

3. **Reserva de Memoria:** La memoria de datos y operaciones se asigna en la primera iteración.

Cabe destacar que en la biblioteca *CuPy*, aunque previamente hayamos utilizado esta para realizar operaciones, al volver a hacer uso de la misma se vuelve a inicializar ciertos componentes. Por lo tanto, hemos añadido una capa inicial que nos ayudará en este caso y que no se tomará en cuenta a la hora de comparar las bibliotecas. Existen varias razones por las cuales la primera iteración puede ser más costosa, como se enumeran a continuación:

1. **Asignación y Gestión de Memoria:** La memoria debe ser asignada y gestionada eficientemente, considerando las características de la arquitectura (*CPU, GPU o TPU*).

2. **Cachés y Optimización:** La optimización del uso de *cachés* y la ejecución es crucial en todas las arquitecturas de procesamiento.

3. **Lanzamiento del *Kernel*:** Ajustes iniciales en el lanzamiento del *kernel* o la función de procesamiento principal, asegurando la configuración óptima para la carga de trabajo.

4. **Optimización de Recursos:** La optimización de recursos específicos, adaptando técnicas como la paralelización o maximizar la utilización del *hardware*.

```
#Establacemos la cantidad de repeticiones.
repeticiones = n

#Definimos las listas donde almacenamos los tiempos y GFLOPS.
iters_time_xx_yy = []
gflops_xx_yy = []
```

El siguiente fragmento de código solo se emplea para las bibliotecas *Numba* y *JAX*, este último si usamos el compilador *JIT*. Esto se debe a que es necesario si queremos compilar y optimizar el código, aprovechando al máximo las capacidades de *hardware* y obteniendo mejoras significativas en rendimiento.

```
@compilador
def matmul(A, B):
  return xx.dot(A,B)
```

Donde figura *@compilador* debemos sustituirlo por *@jit* en el caso de usar la biblioteca *JAX*, o por *@njit* al utilizar la biblioteca *Numba*.

A continuación, se crean las dos matrices de forma aleatoria que vamos a multiplicar, A y B.

Para *Numba* y *NumPy*:

```
#Creamos las matrices.
for M, N, K in MNK:
    A = xx.random.rand(M, K)
    B = xx.random.rand(K, N)
```

En la biblioteca *Cupy*, es necesario especificar la precisión en coma flotante deseada al crear las matrices. Por defecto, las matrices se crean con una precisión en coma flotante de 64 bits, pero nosotros queremos utilizar una precisión de 32 bits.

Para *CuPy*:

```
#Creamos las matrices.
for M, N, K in MNK:
    A = xx.random.rand(M, K, dtype = xx.float32)
    B = xx.random.rand(K, N, dtype = xx.float32)
```

En el caso de *JAX* la creación de matrices tiene una pequeña variación, es necesario usar una clave pseudoaleatoria para asegurar el control de generación de números aleatorios y la reproducibilidad:

```
#Creamos las matrices.
key = random.PRNGKey(0)

for M, N, K in MNK:
  A = random.uniform(key,shape=(M, K))
  B = random.uniform(key,shape=(K, N))
```

Calculamos las iteraciones tantas veces como repeticiones tenemos. Si no usamos el compilador *JIT* emplearemos el siguiente código:

```
    start_time_bucle = time.time()
    for _ in range(repeticiones):
        C = xx.dot(A, B)
    end_time_bucle = time.time()
```

En caso de usar el compilar *JIT* sustituimos la tercera línea del código anterior por la función que hemos creado previamente para las bibliotecas *JAX* y *Numba*.

```
    start_time_bucle = time.time()
    for _ in range(repeticiones):
        C = matmul(A, B)
    end_time_bucle = time.time()
```

Por último, vamos a calcular el tiempo medio por iteración y los *GFLOPS* medios de cada conjunto de multiplicación de matrices.

```
    #Calculamos tiempo por iteracion.
    total_time = end_time_bucle - start_time_bucle
    iter_time = total_time / repeticiones
    iters_time_xx_yy.append(iter_time)

    #Calculamos GFLOPS.
    flops = 2 * M * N * K

    gflop_value = (flops / iter_time) / 1e9
    gflops_xx_yy.append(gflop_value)
```

Este código se repite para cada una de las bibliotecas en cada uno de los núcleos computacionales estudiados.

5.2.2 Gráficos

Para visualizar de forma más clara las distintas bibliotecas utilizadas en núcleos *GPU*, *CPU* y *TPU*, hemos decidido crear un gráfico de barras que presenta la media de los *GFLOPS* obtenidos en 20 multiplicaciones de matrices correspondientes a las diferentes capas convolucionales de una red neuronal.

A continuación, mostramos el código empleado para generar el gráfico en los diferentes experimentos, por ejemplo de dos bibliotecas.

Utilizaremos *xx* en sustitución del nombre de la biblioteca empleada y *yy* en sustitución del tipo de núcleo computacional (*CPU, GPU o TPU*).

```
#Crear los indices.
indices = np.arange(len(gflops_xx_yy_1)-1)
#Reducir el ancho de las barras.
bar_width = 0.2
#Opacidad de las barras.
opacity = 0.8
```

```
#Generar graficos de barras.
fig, axis = plt.subplots(figsize=(30, 10))

#Primer grafico de barras.
bars_xx_1 = plt.bar(indices, gflops_xx_yy_1[1:],
                    bar_width,
                    alpha=opacity,
                    color='gold',
                    label='Biblioteca_1')

#Segundo grafico de barras.
bars_xx_2 = plt.bar(indices + bar_width, gflops_xx_yy_2[1:],
                    bar_width,
                    alpha=opacity,
                    color='deepskyblue',
                    label='Biblioteca_2')

#Agregar titulo en los ejes y en el grafico.
plt.xlabel('Indice', fontsize=20)
plt.ylabel('GFLOPS', fontsize=20)
plt.title('titulo', fontsize=24)

#Centrar etiquetas y ajustar la escala.
plt.xticks(indices + 1.5 * bar_width, indices + 1, fontsize=16)
plt.yticks(fontsize=16)

#Crear las etiquetas de los valores dentro de las barras.
def add_labels(bars):
    for bar in bars:
        height = bar.get_height()
        #Ajustar etiquetas de las barras.
        plt.text(bar.get_x() + bar.get_width() / 2, height - 0.05,
                 f'{height:.2f}', ha='center', va='top', fontsize=14,
                    color='black', fontweight='bold', rotation=90)

#Agregar las etiquetas.
add_labels(bars_numpy)
add_labels(bars_numba)

#Agregar la leyenda y mostrar el grafico.
plt.tight_layout()
plt.legend(fontsize=16)
plt.show()
```

Podemos añadir tantos grupos de barras como sea necesario. Para hacerlo, utilizamos la función *plt.bar()* con los parámetros adecuadas, como son el color, el título y los datos.

5.3 Precisión en coma flotante

La precisión de los cálculos en coma flotante es crucial en los diversos campos científicos donde se requieren resultados altamente precisos. El uso de coma flotante se utiliza para representar una amplia gama de valores numéricos, pero puede tener errores de redondeo debido a la forma de almacenamiento de los números.

En nuestro caso, vamos a emplear dos tipos de precisión, la de 16 bits (*Half-precision*) y la de 32 bits (*Single-precision*).

La precisión de 16 bits se emplea para las aplicaciones donde el ahorro de memoria es crucial, como son las redes neuronales. En cambio, la precisión de 32 bits se usa en gráficos y simulaciones científicas.

Realizamos un estudio preliminar para evaluar la viabilidad de ejecutar cálculos utilizando diferentes bibliotecas y precisiones en coma flotante.

Seguidamente, analizaremos los ajustes necesarios para elegir la precisión en coma flotante que se utilizará en nuestro estudio. La parte del código que debemos editar es la creación de forma aleatoria de las matrices.

Utilizaremos *xx* en sustitución del nombre de la biblioteca empleada. Recordemos que las bibliotecas estudiadas son *NumPy, Numba, CuPy y JAX*.

Para *NumPy:*

```
#Creamos las matrices.
for M, N, K in MNK:
    A = xx.random.rand(M, K).astype(xx.float16)
    B = xx.random.rand(K, N).astype(xx.float16)
```

Para *JAX:*

```
#Creamos las matrices.
key = random.PRNGKey(0)

for M, N, K in MNK:
  A = random.uniform(key,shape=(M, K),dtype=xx.float16)
  B = random.uniform(key,shape=(K, N),dtype=xx.float16)
```

En *Numba* no es posible realizar este estudio a causa de la biblioteca que se emplea por detrás, que es *NumPy*.

Las implementaciones de las rutinas *numpy.linalg* en *Numba* solo admiten los tipos de punto flotante que se utilizan en las funciones *LAPACK* que proporcionan la funcionalidad central subyacente.

Como resultado, solo se admiten los tipos *float32, float64, complex64 y complex128*. Si por ejemplo, tenemos el tipo *float16*, se debe realizar una conversión de tipo apropiada a un tipo de punto flotante apropiado antes de usarlo en nuestro código. La razón de esta decisión es esencialmente evitar tener que replicar las elecciones de conversión de tipo hechas en *Numpy* y también alentar al usuario a elegir el tipo de punto flotante óptimo para la operación que está realizando.

En el caso de la biblioteca *Cupy* tampoco es posible la realización del estudio dado que la función *random.rand()* solo acepta las precisiones en coma flotante de 32 y 64 bits.

Para comparar el uso de coma flotante, hemos creado un gráfico de barras, como se detalla en la Sección 5.2.2 Gráficos.

6 Experimentos

6.1 *CPU*

En esta sección vamos a exponer los resultados de los experimentos realizados para *CPU*, con las diferentes bibliotecas estudiadas. Para *CPU* usamos las bibliotecas *NumPy, Numba y JAX*.

6.1.1 Precisión en coma flotante

Los experimentos realizados para seleccionar la mejor precisión en coma flotante utilizan los siguientes valores para los parámetros *bs* y *repeticiones*. Para *bs* usamos 1 y para *repeticiones*, 25.

La razón para utilizar un número tan bajo de repeticiones es el tiempo que consume cada repetición. Inicialmente, probamos con un valor de 100, pero tras esperar más de 4 horas sin que la ejecución finalizara, decidimos optar por un número menor. Esto sigue permitiéndonos evaluar si es preferible usar coma flotante de 16 bits en lugar de 32 bits.

Cabe destacar que el tiempo elevado consumido por la precisión en coma flotante de 16 bits se debe a que la biblioteca utilizada por *NumPy* no lo soporta, y por lo tanto, se emula por *software*.

El gráfico presentado a continuación hace referencia a la ejecución del código de multiplicación de matrices con la biblioteca *NumPy* en *CPU*, para 16 y 32 bits.

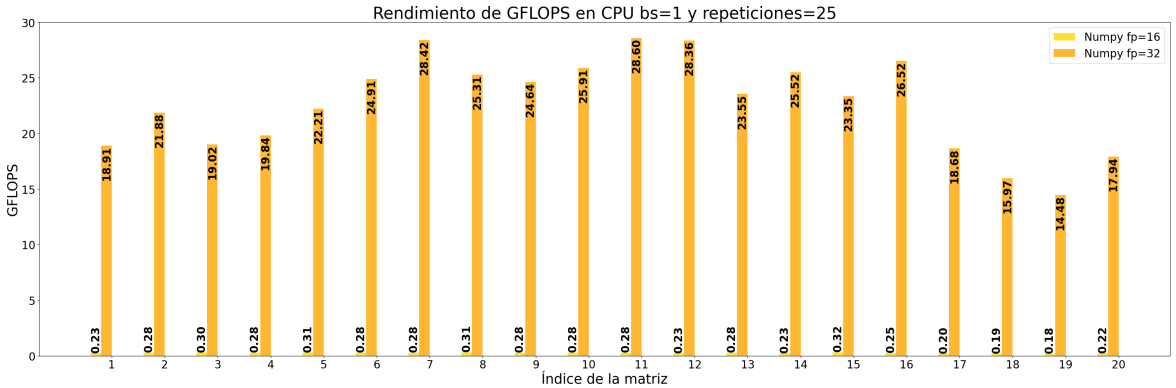

Figura 23: Resultados en *GFLOPS* para *NumPy* en *CPU*.

Al observar el gráfico de la Figura 23, notamos que el valor de *GFLOPS* es mayor para la precisión de 32 bits en comparación con la de 16 bits. Los *GFLOPS* son una medida del rendimiento computacional que calcula el número de operaciones en punto flotante por segundo, por lo tanto, un valor más alto indica un mejor rendimiento. Con esta información, concluimos que la precisión en coma flotante de 32 bits es la más adecuada para nuestros propósitos.

Ahora continuamos con el gráfico de la ejecución de la multiplicación de matrices con la biblioteca *JAX* sin *JIT* en *CPU*, para 16 y 32 bits.

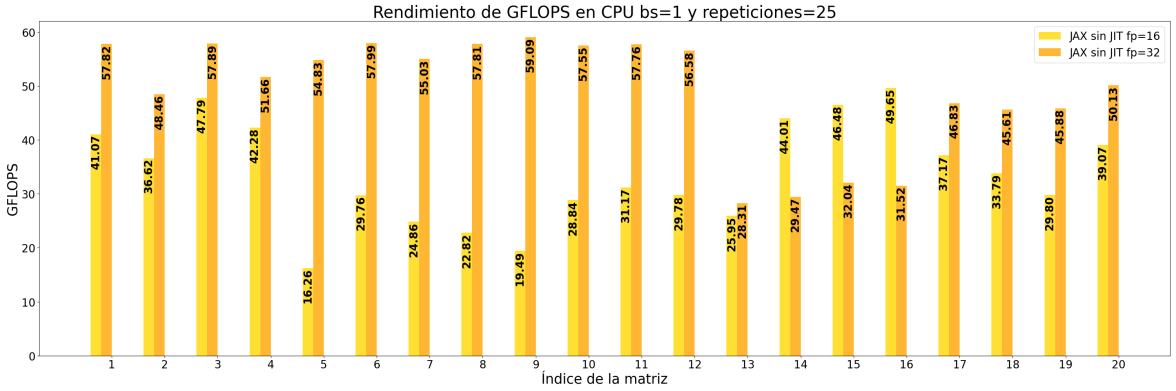

Figura 24: Resultados en *GFLOPS* para *JAX* sin *JIT* en *CPU*.

Al igual que en el caso anterior, en su mayoría los valores más altos de *GFLOPS* los obtenemos con la precisión en coma flotante de 32 bits.

Por último, tenemos el gráfico donde comparamos la precisión de 16 bits y de 32 bits para la multiplicación de matrices usando la librería *JAX* con *JIT* en *CPU*.

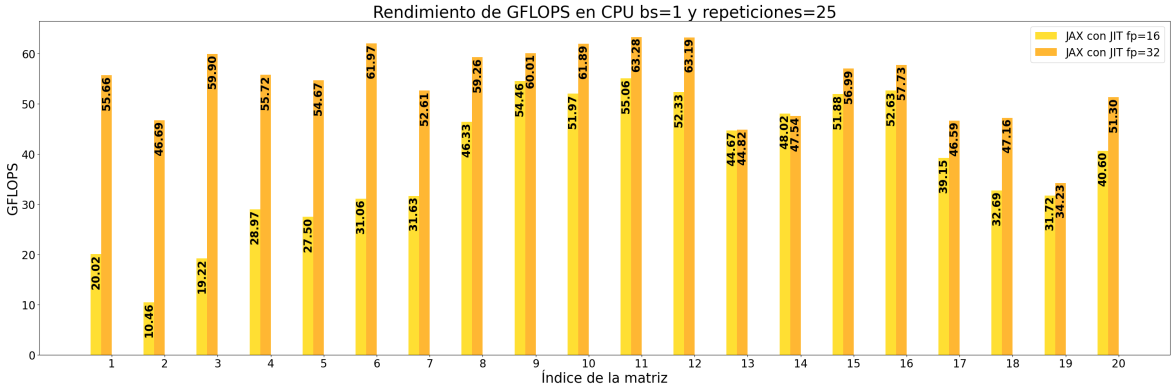

Figura 25: Resultados en *GFLOPS* para *JAX* con *JIT* en *CPU*.

En este tercer gráfico (Figura 25), observamos que predominantemente los valores más altos de *GFLOPS* se obtienen para la precisión en coma flotante de 32 bits. Tenemos

algunos casos puntuales donde la precisión en coma flotante de 16 bits obtiene un valor mayor, como puede ser la décima y la decimotercera multiplicación.

En conclusión, para las 3 bibliotecas estudiadas obtenemos el mismo resultado que es en su mayoría un mejor rendimiento usando la precisión en coma flotante de 32 bits.

6.1.2 Multiplicación de matrices

Los resultados que se presentarán en el gráfico (Figura 26) de este apartado corresponden a cuatro experimentos realizados en *CPU* con las bibliotecas *NumPy*, *Numba*, *JAX* sin *JIT* y *JAX* con *JIT*. Los experimentos se realizaron con los siguientes parámetros: $bs = 1$, $repeticiones = 1000$ y 32 bits de precisión en coma flotante.

Para determinar cuál de las bibliotecas es más eficiente usando *CPU*, compararemos el valor medio de *GFLOPS* obtenido en cada una de las multiplicaciones de matrices que hemos realizado, asignando un color diferente a cada biblioteca en el gráfico.

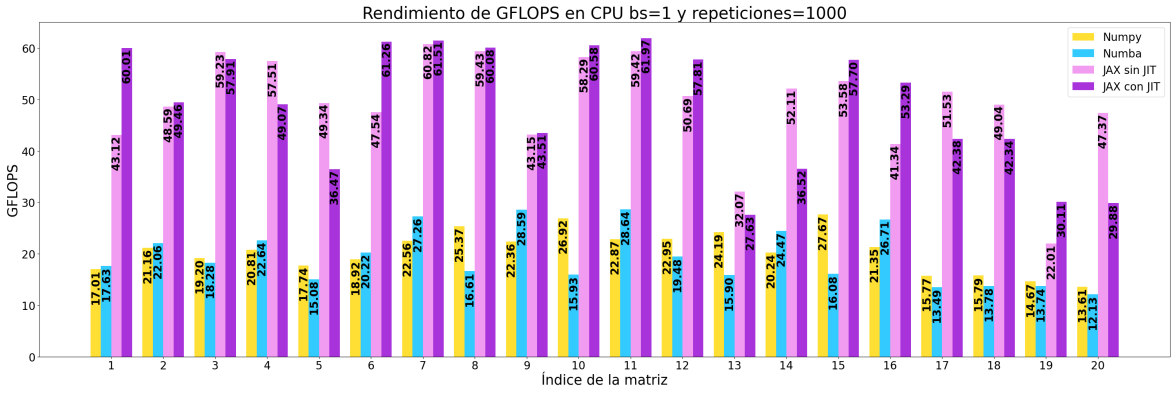

Figura 26: Resultados en *GFLOPS* para *CPU*.

Si observamos detenidamente el gráfico (Figura 26), notamos que una biblioteca destaca claramente por encima de las demás en términos de *GFLOPS* es *JAX*. Una vez descartadas las otras dos bibliotecas, debemos decidir si es mejor emplear el compilador *JIT* de *JAX* o no.

En la mayoría de los casos, *JAX* sin *JIT* obtiene valores superiores de *GFLOPS* en comparación con el uso del compilador *JIT*. Esto puede deberse a que el compilador *JIT* no está optimizando adecuadamente los recursos disponibles para nuestras tareas específicas.

6.2 *GPU*

La presente sección recopila los resultados de los experimentos realizados para *GPU*, con las diferentes bibliotecas estudiadas, *JAX*, *CuPy* y *Numba*.

6.2.1 Precisión en coma flotante

Los experimentos realizados para seleccionar la mejor precisión en coma flotante utilizan los siguientes valores para los parámetros *bs* y *repeticiones*. Para *bs* usamos 1 y para *repeticiones*, 25.

Como hemos comentado anteriormente (Sección 5.3 Precisión en coma flotante) no podemos realizar este experimento con la bibliotecas *Numba* y *Cupy*.

El primer gráfico expone los resultados de la ejecución de la multiplicación de matrices en *GPU* implementando la biblioteca *JAX* sin usar el compilador *JIT*, la medida estudiada es la empleada ya con anterioridad, *GFLOPS*.

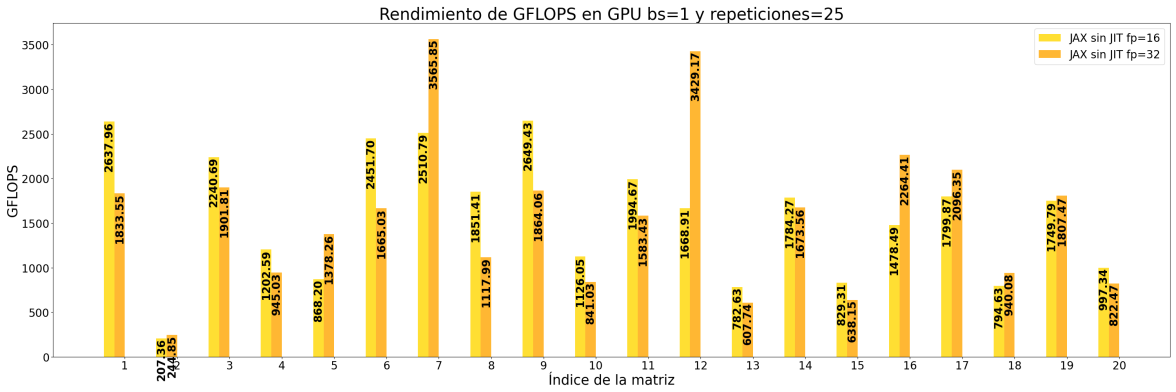

Figura 27: Resultados en *GFLOPS* para *JAX* sin *JIT* en *GPU*.

Si observamos el gráfico (Figura 27) detenidamente notamos que es mejor el uso de 16 bits. La precisión de 32 bits es mejor en el caso de las matrices de mayor dimensión, que son las situadas a la derecha. Por lo tanto, si tenemos un gran conjunto de datos nos interesa implementar una precisión en coma flotante de 32 bits, y en caso de matrices pequeñas es mejor aplicar 16 bits.

Por último, tenemos un gráfico que representa en *GFLOPS* el rendimiento de la librería *JAX* con el compilador *JIT* para una precisión en coma flotante de 16 y 32 bits.

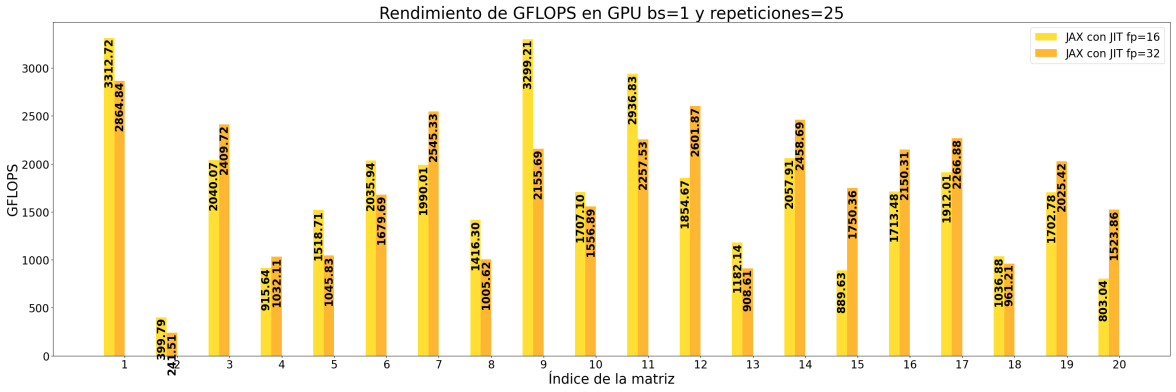

Figura 28: Resultados en *GFLOPS* para *JAX* con *JIT* en *GPU*.

En el gráfico de la Figura 28 no podemos tomar una decisión clara para seleccionar la mejor precisión. Dado que tenemos la misma cantidad de multiplicaciones que dan mejor rendimiento para 16 y 32 bits. Cabe destacar, que en las matrices de la derecha, las más grandes, la precisión en coma flotante de 32 bits es mejor.

En conclusión, si tenemos un conjunto de datos pequeño es mejor el uso de 16 bits. Y en caso de un conjunto de datos grande se recomienda el uso de 32 bits.

6.2.2 Multiplicación de matrices

Los resultados que se presentarán en este primer gráfico (Figura 29) corresponden a tres experimentos realizados en *GPU* para las bibliotecas *CuPy, JAX* sin *JIT, JAX* con *JIT* y *Numba*. Los experimentos se realizaron con los parámetros: *bs*=1, *repeticiones*=1000 y 32 bits de precisión en coma flotante.

Para determinar cuál de las bibliotecas es más eficiente usando *GPU*, compararemos el valor medio de *GFLOPS* obtenido en cada una de las multiplicaciones de matrices que hemos realizado, asignando un color diferente a cada biblioteca en el gráfico.

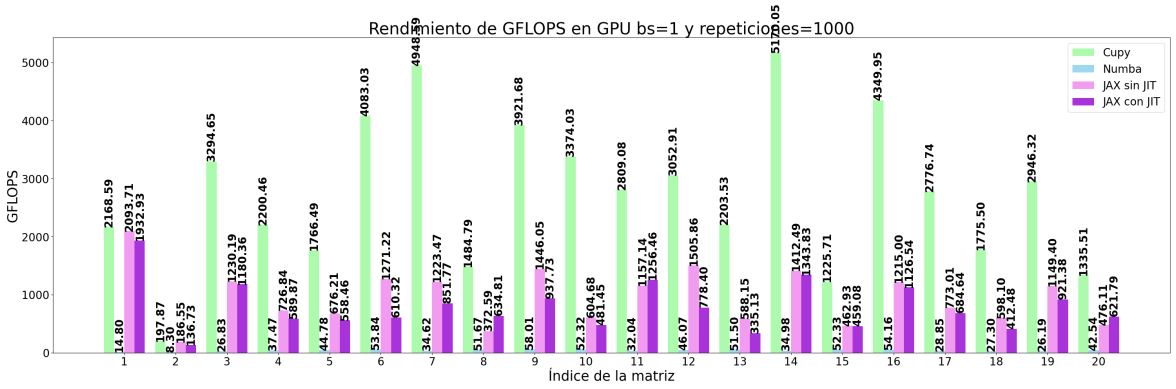

Figura 29: Resultados en *GFLOPS* para *GPU* para bs=1 y repeticiones=1000.

El gráfico anterior (Figura 29) nos ayudara a seleccionar las dos mejores bibliotecas para *GPU*. Presenta una clara decisión de descartar la biblioteca *Numba* debido a sus bajos valores de *GFLOPS*, los cuales oscilan entre 10 y 50 *GFLOPS*. En contraste, la biblioteca *CuPy* muestra valores significativamente superiores, situándose entre 800 y 5000 *GFLOPS*. Esta diferencia de rendimiento es considerable, lo que justifica la eliminación de *Numba* para tareas que requieren alta capacidad de procesamiento.

A continuación, probamos a cambiar el valor del parámetro *bs* (*batch size*) a 128 con el mismo número de repeticiones para las dos mejores bibliotecas, *JAX* y *Cupy*. También se probó a cambiar el número de *repeticiones*=1250 con un *bs*=128. De esta forma se pretendía poder tomar la decisión de si era mejor usar o no el compilador *JIT* para *JAX* y ver si mejoraba el rendimiento de *CuPy*.

Obtuvimos como resultado que la biblioteca *CuPy* mantenía un valor semejante al del previo experimento. Por otra parte, no pudimos tomar una decisión definitiva con el uso de *JIT* en *JAX*, dado que obtuvimos una distribución equitativa entre las multiplicaciones, es decir, teníamos el mismo número de resultados buenos para la precisión en coma flotante para 16 y 32 bits.

Por último, para lograr obtener una decisión final entre el uso de *JIT* en *JAX*, decidimos además de ajustar el parámetro *bs* a 128, aumentar el número de *repeticiones* a 2000. Una vez más, hemos realizado pruebas con las dos mejores bibliotecas identificadas previamente, *CuPy* y *JAX*.

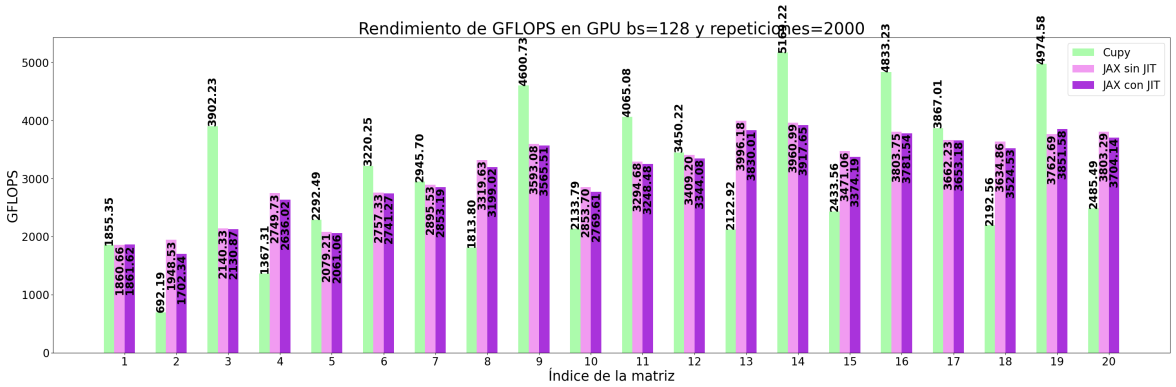

Figura 30: Resultados en *GFLOPS* para *GPU* para bs=128 y repeticiones=2000.

Este último gráfico de la sección (Figura 30) nos confirma que *JAX* es la biblioteca que más aprovecha las capacidades de una *GPU* en la mayoría de los casos, excepto en la 1, 3, 5, 6, 9, 11, 12, 14, 16, 17 y 19 multiplicación de matrices. Mientras que en *JAX* no podemos tomar una clara decisión de si usar o no el compilador *JIT*. No obstante, cabe destacar que, si debemos decantarnos por usar *JIT* o no, la mejor opción es hacer uso de él para grandes conjuntos de datos.

En conclusión, la mejor biblioteca para aprovechar las capacidades de una *GPU* es *CuPy* usando un número de repeticiones pequeño, mientras que para un número de repeticiones más grande en la mayoría de los casos es mejor usar *JAX*.

6.3 *TPU*

En esta sección vamos a exponer los resultados de los experimentos realizados para *TPU*. Usaremos solo la biblioteca *JAX*, dado que no hemos encontrado otra biblioteca con sus mismas características que podamos usar para compararla. Y la biblioteca *NumPy* no tiene soporte directo para *TPU*.

6.3.1 Precisión en coma flotante

Los experimentos realizados para seleccionar la mejor precisión en coma flotante utilizan los siguientes valores para los parámetros *bs* y *repeticiones*. Para *bs* usamos 1 y para *repeticiones*, 25.

El primer experimento se ha realizado con la biblioteca *JAX* sin emplear el compilador *JIT*, con 16 y 32 bits de precisión en coma flotante. Los resultados se muestran

en *GFLOPS*.

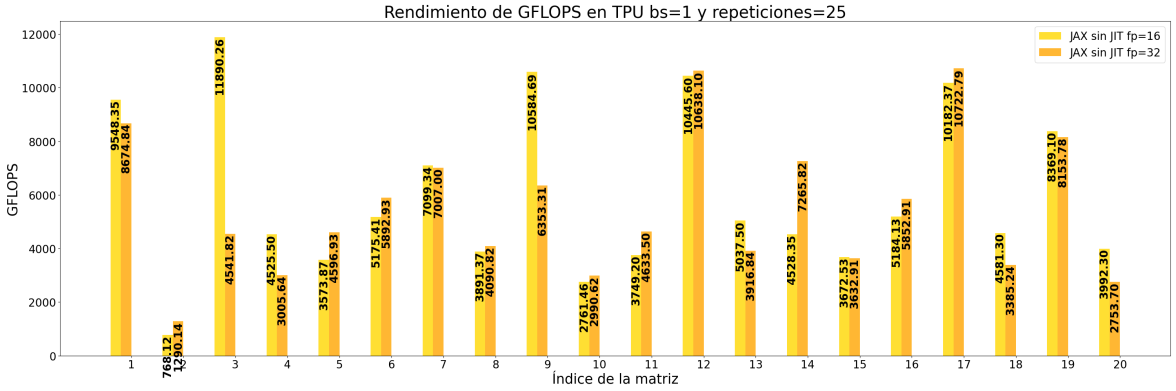

Figura 31: Resultados en *GFLOPS* para *JAX* sin *JIT* en *TPU*.

Los valores más altos de *GFLOPS* los obtenemos mayormente con la precisión en coma flotante de 32 bits.

Por último, tenemos el gráfico donde comparamos la precisión de 16 bits y de 32 bits para la multiplicación de matrices usando la librería *JAX* con *JIT* en *TPU*.

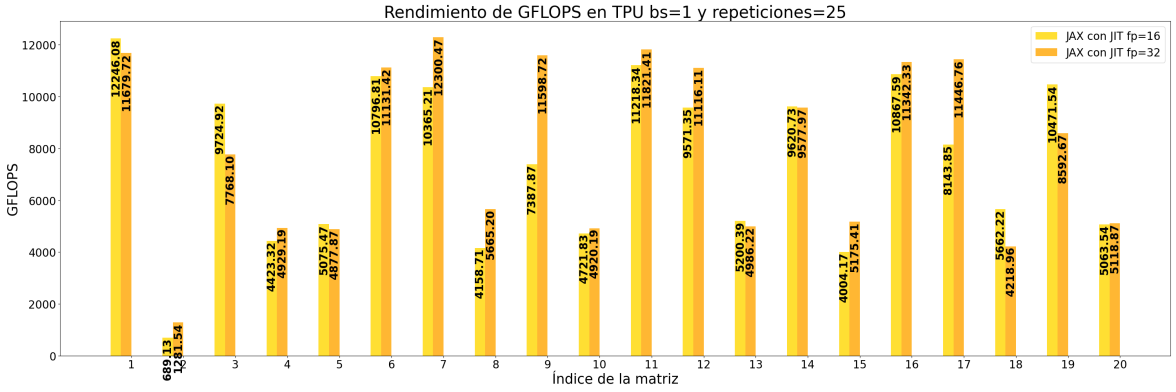

Figura 32: Resultados en *GFLOPS* para *JAX* con *JIT* en *TPU*.

En este segundo gráfico (Figura 32), observamos que los valores más altos de *GFLOPS* se obtienen en su mayoría para la precisión en coma flotante de 32 bits.

En conclusión, para la biblioteca estudiada obtenemos el mismo resultado que es un mejor rendimiento usando la precisión en coma flotante de 32 bits.

6.3.2 Multiplicación de matrices

Los resultados que se presentarán en el gráfico (Figura 33) de este apartado corresponden a tres experimentos realizados en *TPU* con las bibliotecas *NumPy*, *JAX* sin

JIT y *JAX* con *JIT*. Los experimentos se realizaron con los siguientes parámetros: *bs* = 1, *repeticiones* = 1000 y 32 bits de precisión en coma flotante.

Para determinar cuál de las bibliotecas es más eficiente usando *TPU*, compararemos el valor medio de *GFLOPS* obtenido en cada una de las multiplicaciones de matrices que hemos realizado, asignando un color diferente a cada biblioteca en el gráfico.

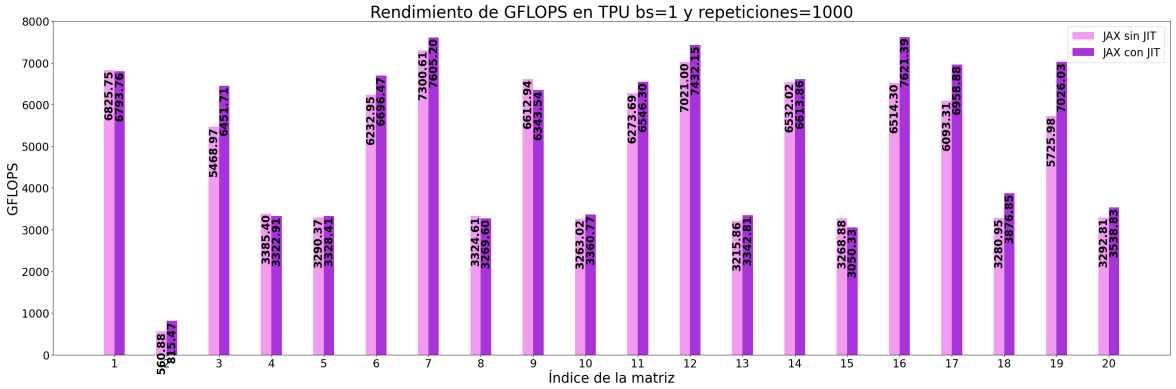

Figura 33: Resultados en *GFLOPS* para *TPU* para bs=1 y repeticiones=1000.

Si observamos detenidamente el gráfico (Figura 33), notamos que obtenemos valores más altos al usar el compilador *JIT* en *JAX*.

Para aprovechar mejor el uso de la *TPU*, hemos decidido cambiar ambos parámetros definidos en el código, *repeticiones* y *bs*. En este caso, hemos usado *repeticiones* = 1250 y *bs*=128. De esta forma, queremos asegurarnos de que estamos utilizando todas las capacidades de la *TPU* de manera efectiva.

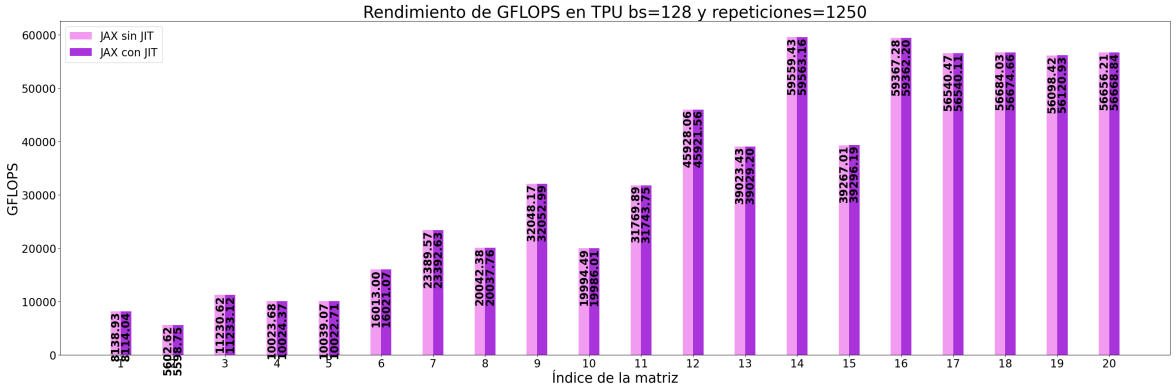

Figura 34: Resultados en *GFLOPS* para *TPU* para bs=128 y repeticiones=1000.

El gráfico (Figura 34) confirma que podríamos aprovechar un poco más las capacidades de la *TPU*. Observamos que los resultados son muy similares tanto si utilizamos

el compilador *JIT* como si no lo hacemos. Esto podría deberse al tamaño de bloque de la *TPU*, que es 128×128.

Este tamaño de bloque puede afectar el rendimiento y la eficiencia del procesamiento, especialmente en configuraciones donde los datos no están optimizados para dicho tamaño de bloque.

7 Conclusiones y trabajo futuro

En este apartado se concentran los resultados más importantes obtenidos durante la investigación y el desarrollo del proyecto.

Se presenta un resumen de las conclusiones surgidas de los experimentos realizados y se proponen diversas líneas de trabajo futuro que podrían continuar y expandir la investigación actual.

Estas conclusiones y propuestas tienen como objetivo proporcionar una guía para posibles mejoras y optimización del rendimiento de las bibliotecas analizadas, así como explorar nuevas áreas de aplicación y desarrollo.

7.1 Conclusiones

Para una mejorar clasificación de las conclusiones hemos decidido dividir las conclusiones en cuatro secciones diferentes que expondremos a continuación.

7.1.1 Eficiencia y rendimiento de las bibliotecas

- **NumPy:** Se ha demostrado que, a pesar de ser de las bibliotecas más usadas, el rendimiento en *GFLOPS* para *GPU* y *TPU* es inferior a otras bibliotecas estudiadas.
- **CuPy:** Esta biblioteca esta diseñada para ejecutar código en *GPU*. Ha demostrado un rendimiento significativamente mejor en álgebra lineal y en pruebas de precisión en coma flotante.
- **Numba:** Ofrece una mejora en el rendimiento al usar el compilar *JIT*. Aunque no alcanza el nivel de optimización de *CuPy* en *GPU*, es una opción viable para acelerar el rendimiento sin necesidad de modificar en exceso el código.
- **JAX:** Para ejecutar los cálculos tanto en *CPU* como en *GPU* con un alto rendimiento gracias a la optimización y uso eficiente de recursos. Además, en *TPU* se demuestra que es la mejor opción dado que es la más eficiente cuando se utilizan adecuadamente los parámetros de configuración, como el tamaño del lote.

En conclusión, el uso de la biblioteca *JAX* ha demostrado ser una gran aliada cuando queremos obtener un rendimiento superior a otras bibliotecas. Además, es versátil a la hora de usar diferentes núcleos computacionales, es decir, no es necesario realizar modificaciones en el código para ejecutarlo en *CPU, GPU* y *TPU*.

No debemos olvidar que la biblioteca *CuPy* es una buena opción para usar en *GPU*, pero la mejor es *JAX*. Mientras que para los núcleos computacionales *CPU* y *TPU* ha sido indudablemente la biblioteca *JAX*.

7.1.2 Precisión en coma flotante

Los experimentos realizados en las bibliotecas y núcleos computacionales estudiados, han demostrado que todas las bibliotecas que se obtiene un mejor rendimiento al usar 32 bits de precisión en coma flotante. Esto es teniendo en cuenta ligeras variaciones que dependen del hardware utilizado.

7.1.3 Portabilidad

Durante el desarrollo de este trabajo, se realizaron pruebas en entornos diversificados como *Google Colab y Kaggle*, aprovechando sus recursos de *hardware* variados (*CPU, GPU* y *TPU*). Estas pruebas demostraron que el código escrito en *JAX* podía trasladarse entre estos entornos sin requerir ajustes.

Finalmente, se puede concluir que *JAX* ha permitido cambiar de plataforma sin necesidad de modificar el código ni cambiar de biblioteca, logrando así uno de los objetivos clave de este proyecto: la portabilidad eficiente y efectiva del código.

7.1.4 Desafíos y límites

Algunas de las bibliotecas estudiadas, como puede ser *NumPy*, han demostrado no ser ideales para tareas que requieren un uso de *hardware* especializado. La optimización de código y la configuración de los parámetros son cruciales para maximizar el rendimiento.

7.2 Trabajo futuro

Una vez expuestas las conclusiones obtenidas a lo largo del estudio podemos pasar a identificar las áreas más prometedoras para futuras investigaciones y desarrollos. Dichas áreas de trabajo futuro están orientadas a la optimización del rendimiento de las bibliotecas utilizados, expandir el alcance de los experimentos para abordar desafíos más complejos. A continuación, detallamos algunas de las principales líneas de investigación donde se podría seguir trabajando en un futuro.

7.2.1 Exploración de nuevas bibliotecas

Investigar y evaluar nuevas bibliotecas que puedan ofrecer las mismas funciones y mejoras en términos de rendimiento y facilidad de uso, especialmente aquellas diseñadas para entornos de computación emergentes.

7.2.2 Ampliación del conjunto de datos

Realizar pruebas adicionales con conjuntos de datos más grandes y variados para evaluar cómo se comportan las diferentes bibliotecas en nuevos escenarios complejos y realistas.

7.2.3 Integración con otros sistemas

Investigar la integración de las bibliotecas estudiadas con otros sistemas y plataformas de aprendizaje automático y análisis de datos para crear *pipelines* más eficientes y escalables.

Con estas líneas de trabajo, podemos avanzar con la mejora del rendimiento y la eficiencia en el procesamiento de datos y aprendizaje automático, aprovechando al máximo las capacidades de las bibliotecas disponibles.

Estas conclusiones y propuestas de trabajo futuro se basan en los resultados obtenidos y el análisis plasmado en el documento, proporcionando una visión clara de los avances y pasos a seguir para continuar mejorando en esta investigación.

7.3 Objetivos de desarrollo sostenible

Nuestro proyecto tiene relación directa con varios de los Objetivos de desarrollo sostenible (ODS) propuestos por la ONU, contribuyendo de manera significativamente en áreas clave para un futuro más sostenible.

En primer lugar, cumplimos el *ODS 7: Energía asequible y no contaminante* que se refleja en la búsqueda de una eficiencia mayor en el uso de recursos tecnológicos.

Gracias a la mejora en el rendimiento de *hardware* (CPU, GPU y TPU), se optimiza el consumo energético, contribuyendo a la reducción de la huella de carbono asociada a los grandes centros de datos y al procesamiento de estos. De este modo, se fomenta el uso responsable y eficiente de la energía en un mundo cada vez más digital.

En segundo lugar, fomentamos la innovación tecnológica, alineándose con el *ODS 9: Industria, Innovación e Infraestructura*.

Promovemos la innovación en el campo de la computación de alto rendimiento. El estudio e implementación de herramientas como *JAX* ayudan a impulsar el desarrollo de infraestructuras más avanzadas, que sean capaces de soportar aplicaciones complejas, como son la inteligencia artificial. Dando como resultado un avance en la industria tecnológica para afrontar futuros desafíos de forma sostenible y eficiente.

Por último, nos implicamos en el ámbito educativo y formativo, contribuyendo al *ODS 4: Educación de calidad*.

Las herramientas y metodologías desarrolladas en esta investigación pueden ser implementadas en contextos educativos, facilitando el acceso a tecnologías avanzadas y proporcionando recursos que pueden ayudar a la formación de profesionales en áreas relacionadas con la computación y el aprendizaje automático.

De esta manera, el proyecto además de aportar avances en el campo de la computación también contribuye en la consecución de un futuro más sostenible y equitativo, en consonancia con los Objetivos de desarrollo sostenible (ODS).

8 Bibliografía

[1] CPU Intel(R) Core(TM) i7-8550U. https://www.intel.la/content/www/xl/es/products/sku/122589/intel-core-i78550u-processor-8m-cache-up-to-4-00-ghz/specifications.html.

[2] Documentación de JAX. https://jax.readthedocs.io/en/latest/quickstart.html.

[3] Documentación de Numba. https://numba.readthedocs.io/en/stable/index.html.

[4] Documentación de NumPy. https://numpy.org/doc/stable/.

[5] Documentación del Compilador LLVM. https://llvm.org/.

[6] GPU NVIDIA Tesla T4. https://www.nvidia.com/es-es/data-center/tesla-t4/.

[7] Página Web Oficial de CuPy. https://cupy.dev/.

[8] Página Web Oficial para descargar la distribución de Anaconda. https://www.anaconda.com/download/success.

[9] TPU v3-8. https://www.kaggle.com/docs/tpu.

[10] Cláudio Lemos. JAX vs NumPy. https://www.tensorops.ai/post/should-i-switch-from-numpy-to-jax-1, 31, July 2023.

[11] Daniel San Martin y Claudio E. Torres. 2D Simplified Wildfire Spreading Model in Python: From Numpy to CuPy. https://clei.org/cleiej/index.php/cleiej/article/view/578/449, 5, May 2023.

[12] Gavin Chan. Python Compiler Comparison (1). https://www.linkedin.com/pulse/python-compiler-comparison-1-gavin-chan, 13, July 2022.

[13] Gavin Chan. Python Compiler Comparison (2). https://www.linkedin.com/pulse/python-compiler-comparison-2-gavin-chan, 26, August 2022.

[14] Peng Xu, Ming-Yan Sun, Yin-Jun Gao, Tai-Jiao Du, Jing-Ming Hu, and Jun-Jie Zhang. Influence of data amount, data type and implementation packages in GPU coding. https://www.sciencedirect.com/science/article/pii/S2590005622000947, 16, November 2022.