Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**

Grado en Ingeniería Informática

Trabajo Fin de Grado

Análisis y optimización de un motor de ray-tracing mediante técnicas de paralelización sobre CPU y GPU

Autor: David Morilla Sorlí
Tutor: Ángel Herranz Nieva

Madrid, Julio 2025

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática

Título: Análisis y optimización de un motor de ray-tracing mediante técnicas de paralelización sobre CPU y GPU

Julio 2025

Autor: David Morilla Sorlí
Tutor: Ángel Herranz Nieva
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

Resumen

El objetivo principal de este **Trabajo de Fin de Grado** ha sido mejorar el rendimiento de un motor de *ray-tracing*, una técnica utilizada para generar imágenes realistas simulando el comportamiento de la luz. Dado que el trazado de rayos es un proceso muy demandante en cómputo pero con un alto grado de paralelismo inherente, se eligió este algoritmo para explorar técnicas de paralelización que aprovechen el hardware moderno y así reducir significativamente los tiempos de cálculo intensivo.

Esta mejora en el rendimiento es especialmente relevante porque el *ray-tracing* tradicionalmente requiere tiempos largos de procesamiento, lo que limita su uso en aplicaciones en tiempo real o en proyectos con recursos computacionales restringidos. Por ello, se decidió aplicar técnicas de paralelización sobre CPU y GPU, buscando demostrar cómo el aprovechamiento eficiente del paralelismo puede hacer que este algoritmo sea más accesible y práctico en distintos entornos.

Para ello, se analizó una implementación concreta y más exigente del algoritmo de *ray-tracing* llamada *Stochastic Path Tracing with Russian Roulette Termination*, identificando las secciones del código más críticas en cuanto a tiempo de ejecución y paralelización. Se aplicaron dos enfoques principales: primero, la paralelización sobre CPU mediante OpenMP, para aprovechar el paralelismo a nivel de núcleos de procesador; y luego la aceleración mediante CUDA, que permite explotar el paralelismo masivo de las GPUs. En ambos casos, se adaptó y refactorizó el código para garantizar la correcta gestión de la memoria y evitar errores comunes en programación paralela, como condiciones de carrera. Finalmente, se compararon los resultados obtenidos con cada método para evaluar las mejoras en tiempo de ejecución y eficiencia, mostrando que la aceleración por GPU ofrece un rendimiento superior en cargas de trabajo elevadas aunque también requiere más adaptaciones del código original.

Este estudio evidencia cómo adaptar un algoritmo de *ray-tracing* tradicional a entornos paralelos puede generar beneficios significativos, y sienta las bases para futuros trabajos centrados en la optimización de la transferencia de datos y la escalabilidad en sistemas multicore y multiprocesador gráfico.

Abstract

The main objective of this **Bachelor's Thesis** has been to improve the performance of a ray-tracing engine, a technique used to generate realistic images by simulating the behavior of light. Since ray tracing is a highly computationally demanding process but inherently parallel, this algorithm was chosen to explore parallelization techniques that take advantage of modern hardware and thus significantly reduce intensive computation times.

This performance improvement is especially relevant because traditional ray-tracing requires long processing times, which limits its use in real-time applications or projects with constrained computational resources. Therefore, parallelization techniques were applied on both CPU and GPU, aiming to demonstrate how efficient exploitation of parallelism can make this algorithm more accessible and practical in different environments.

To this end, a specific and more demanding implementation of the ray-tracing algorithm called *Stochastic Path Tracing with Russian Roulette Termination* was analyzed, identifying the most critical code sections regarding execution time and parallelization. Two main approaches were applied: first, parallelization on the CPU using OpenMP to exploit core-level parallelism; and then acceleration using CUDA, which allows exploiting the massive parallelism of GPUs. In both cases, the code was adapted and refactored to ensure correct memory management and avoid common parallel programming errors such as race conditions. Finally, the results obtained with each method were compared to evaluate improvements in execution time and efficiency, showing that GPU acceleration offers superior performance on high workloads but also requires more code adaptation.

This study demonstrates how adapting a traditional ray-tracing algorithm to parallel environments can yield significant benefits and lays the groundwork for future work focused on optimizing data transfer and scalability in multicore and multiprocessor GPU systems.

Tabla de contenidos

1. Introducción **3**
- 1.1. Motivación del proyecto . . . 3
- 1.2. Justificación y contexto del proyecto . . . 3
- 1.3. Objetivos del proyecto . . . 4
  - 1.3.1. Objetivo general . . . 4
  - 1.3.2. Objetivos específicos . . . 4
- 1.4. Metodología de Trabajo . . . 5
- 1.5. Estructura del documento . . . 6

2. Preliminares **7**
- 2.1. Principios de paralelización y concurrencia . . . 7
  - 2.1.1. Condiciones de carrera . . . 8
  - 2.1.2. Principios y límites teóricos de la paralelización . . . 9
- 2.2. Tipos de paralelismo: datos, tareas, instrucciones . . . 10
- 2.3. Arquitecturas de procesamiento paralelo . . . 10
  - 2.3.1. CPUs multinúcleo . . . 10
  - 2.3.2. GPUs y arquitectura CUDA . . . 11
- 2.4. Herramientas de análisis y *profiling* . . . 13

3. Selección y estudio del sistema a paralelizar **15**
- 3.1. Búsqueda preliminar de sistemas candidatos . . . 15
- 3.2. Análisis del sistema FUME . . . 15
  - 3.2.1. Descripción funcional del sistema . . . 15
  - 3.2.2. Estudio del flujo de ejecución . . . 16
  - 3.2.3. Cuellos de botella identificados mediante *profiling* . . . 18
  - 3.2.4. Limitaciones en observar resultados tangibles tras la paralelización en FUME . . . 20
  - 3.2.5. Justificación del cambio de sistema . . . 20
- 3.3. Criterios de selección del nuevo sistema . . . 21

4. Arquitectura del Código Base **23**
- 4.1. Funcionamiento del algoritmo de Ray-Tracing . . . 24
  - 4.1.1. Inicialización de la cámara (`init_camera`) . . . 24
  - 4.1.2. Renderizado de la escena (`render`) . . . 24
  - 4.1.3. Trazado de rayos y cálculo de iluminación (`trace_path`) . . . 24
  - 4.1.4. Intersección de rayos (`intersect` y funciones auxiliares) . . . 26

4.2. Resumen simplificado del funcionamiento del algoritmo de Ray-Tracing . . 26
4.2.1. Resumen del flujo de ejecución 27
4.3. Métricas de rendimiento y profiling inicial 27
4.3.1. Resultados según tiempo de ejecución 28
4.3.2. Resultados según el consumo de memoria 32
4.3.3. Imágenes generadas 33
4.4. Identificación de partes paralelizables 34

5. Implementación de optimizaciones 35
5.1. Paralelización mediante OpenMP 35
5.1.1. Optimizaciones secuenciales previas 35
5.1.2. Estrategia general aplicada 37
5.1.3. Control de dependencias y sincronización 37
5.1.4. Paralelización del bucle principal de píxeles 38
5.1.5. Evaluación de rendimiento con OpenMP 41
5.2. Paralelización mediante CUDA 46
5.2.1. Fundamentos prácticos de CUDA en el proyecto 46
5.2.2. Generación de números aleatorios 47
5.2.3. Inicialización y transferencia de datos a la GPU 48
5.2.4. Diseño de kernels para trazado de rayos 49
5.2.5. Gestión de memoria 53
5.2.6. Evaluación de rendimiento con CUDA 55
5.3. Definición del entorno de pruebas 59
5.4. Comparativa: secuencial vs OpenMP vs CUDA 60
5.4.1. Tiempos de ejecución 60
5.4.2. Análisis de speedup 61
5.4.3. Análisis del uso de memoria 61

6. Conclusiones y trabajo futuro 63
6.1. Discusión sobre ventajas, limitaciones, aplicabilidad según contextos (CPU vs GPU) y coste computacional 63
6.2. Dificultades técnicas encontradas 63
6.3. Objetivos logrados 64
6.4. Líneas de trabajo futuro 65
6.5. Análisis de impacto 65

Bibliografía 67

Anexos 71

A. Informe de originalidad 71
A.1. Informe Turnitin 72
A.2. Porcentaje de similitud 73
A.3. Fuentes principales 74

Acrónimos

ALU (Arithmetic Logic Unit) Unidad Aritmético-Lógica, responsable de las operaciones matemáticas y lógicas en un procesador.

API (Application Programming Interface) Conjunto de funciones que permiten la interacción entre programas y servicios.

CPU (Central Processing Unit) Unidad Central de Procesamiento, encargada de ejecutar instrucciones secuenciales en un sistema.

CUDA (Compute Unified Device Architecture) Plataforma de programación de NVIDIA para ejecutar código en GPUs de forma paralela.

E/S (Entrada/Salida) Entrada y salida de datos en sistemas computacionales.

FUME (Flexible Universal Processor for Modeling Emissions) Procesador universal flexible para la modelización de emisiones.

GPGPU (General-Purpose computing on Graphics Processing Units) Uso de GPUs para tareas que tradicionalmente realiza la CPU, más allá del renderizado gráfico.

GPU (Graphics Processing Unit) Unidad de Procesamiento Gráfico, especializada en operaciones paralelas y cálculo intensivo.

MB (Megabyte) Unidad de medida de almacenamiento digital..

OpenMP (Open Multi-Processing) Extensión de C/C++ para paralelización en CPUs mediante directivas de compilador.

SIMD (Single Instruction, Multiple Data) Modelo de paralelismo en el que una instrucción se ejecuta simultáneamente sobre múltiples datos.

SM (Streaming Multiprocessor) Unidad de ejecución paralela dentro de una GPU, que contiene múltiples núcleos CUDA.

SQL (Structured Query Language) Lenguaje estándar para gestión y consulta de bases de datos relacionales.

Capítulo 1

Introducción

1.1. Motivación del proyecto

La elección de este **Trabajo de Fin de Grado** nace del interés personal por los desafíos que plantea la optimización del rendimiento en sistemas computacionales. Durante el transcurso del grado, asignaturas como *Sistemas Distribuidos*, *Técnicas de Computación Científica*, *Computación de Alto Rendimiento* y *Concurrencia* despertaron un fuerte interés en técnicas orientadas al aprovechamiento del paralelismo, consolidando una base técnica para abordar proyectos exigentes desde el punto de vista computacional.

En la actualidad, el crecimiento exponencial del volumen de datos y la complejidad de los modelos ha hecho que la eficiencia deje de ser un valor añadido para convertirse en un requisito esencial. Desde aplicaciones científicas hasta videojuegos y simulaciones físicas, el rendimiento se ha vuelto crítico [1]. A este fenómeno se suma la evolución del hardware moderno: el aumento del rendimiento ya no se logra únicamente mediante mayores frecuencias de reloj, sino a través del uso eficiente de arquitecturas paralelas [2].

La motivación principal del proyecto, por tanto, es explorar de forma práctica cómo las técnicas de paralelización pueden mejorar el rendimiento de sistemas intensivos en cálculo. Este enfoque permite desarrollar una mirada crítica y técnica sobre cómo escalar algoritmos de manera eficiente, trasladando los conocimientos teóricos adquiridos durante la formación académica a un escenario real.

1.2. Justificación y contexto del proyecto

El marco técnico de este proyecto se inscribe dentro de la computación de alto rendimiento y la programación paralela, dos disciplinas fundamentales para enfrentar problemas que requieren un uso intensivo de recursos computacionales. El avance del hardware —especialmente con la aparición de procesadores multinúcleo y GPUs— ha impulsado la necesidad de diseñar software capaz de aprovechar estas capacidades, haciendo de la paralelización una competencia esencial para el desarrollo eficiente de aplicaciones [3].

En un primer intento, el proyecto se orientó hacia el entorno FUME, un simulador ambiental enfocado en el preprocesamiento de datos para modelización de emisiones. No obstante, tras un análisis de su comportamiento, se detectó que la mayor parte del tiempo de ejecución

se dedicaba a operaciones de E/S sobre una base de datos, lo que no suponía un cuello de botella sobre el cómputo y por tanto, si bien no impedía la aplicación de técnicas de paralelización sobre las partes costosas del cómputo, sí que limitaba significativamente la capacidad de observar mejoras de rendimiento sustanciales únicamente aplicando estas técnicas. Este hallazgo motivó una reevaluación del enfoque inicial.

Con el objetivo de mantener el interés en la optimización del rendimiento, se decidió reformular el proyecto en torno a un nuevo código: un renderizador de ray-tracing. Esta técnica, ampliamente utilizada en gráficos por computadora, presenta una estructura altamente paralelizable, ya que cada rayo puede calcularse de forma independiente. En específico, se empleará el algoritmo conocido como *Stochastic Path Tracing with Russian Roulette Termination*, o simplemente Path Tracing con Russian Roulette. Este método es ampliamente utilizado en gráficos 3D de alta calidad, como en videojuegos modernos o películas animadas, ya que permite representar de forma realista cómo la luz interactúa con los objetos de una escena. A diferencia del *ray tracing* clásico, que suele simular solo los rayos de luz directa (como los reflejos o sombras directas), el *path tracing* lanza varios rayos de luz por cada píxel y simula cómo esa luz rebota muchas veces en diferentes superficies. Esto permite capturar efectos visuales más complejos como la iluminación indirecta, las sombras suaves o el cambio de color entre superficies cercanas. Para evitar que el número de rebotes crezca sin control y afecte el rendimiento, se usa una técnica llamada *Russian Roulette*, que decide de forma aleatoria cuándo terminar un rayo de luz que ya aporta poca iluminación. Gracias a esto, se logran imágenes más realistas sin necesidad de hacer todos los cálculos posibles, aunque con algo de ruido que se puede reducir con filtros o más muestras por píxel.

Este nuevo contexto permite un estudio más directo y controlado del impacto del paralelismo sobre métricas clave como el tiempo de ejecución (sin preocuparnos de altas latencias producidas por la E/S), la escalabilidad del sistema y el aprovechamiento del hardware. De este modo, se garantiza la validez técnica del proyecto, proporcionando resultados extrapolables a otros dominios donde sólo las operaciones de cómputo son un factor determinante.

1.3. Objetivos del proyecto

1.3.1. Objetivo general

Analizar y aplicar técnicas de paralelización en un sistema de alta carga de cómputo, con el fin de obtener mejoras visibles de rendimiento en términos de tiempo total de ejecución del sistema completo y explorar el impacto de diferentes estrategias de programación paralela en la eficiencia y escalabilidad del software.

1.3.2. Objetivos específicos

- Estudiar los fundamentos teóricos y prácticos de la programación paralela, con énfasis en el modelo de memoria compartida y el uso de herramientas como `OpenMP` y `CUDA`.
- Comprender la arquitectura y funcionamiento del código base del renderizador, identificando las secciones del mismo susceptibles de ser paralelizadas.

- Aplicar diferentes estrategias de paralelización sobre el sistema, evaluando su impacto en términos de rendimiento y eficiencia.
- Medir y analizar los resultados obtenidos mediante experimentos controlados, prestando especial atención a los tiempos de ejecución, escalabilidad, aprovechamiento de recursos y la ocupación real de los núcleos.
- Comparar el comportamiento del sistema antes y después de las optimizaciones, justificando las decisiones técnicas adoptadas.
- Documentar las técnicas aplicadas, resultados obtenidos, dificultades enfrentadas y conclusiones, mediante la elaboración de una memoria técnica detallada.

1.4. Metodología de Trabajo

El proceso seguido para optimizar el rendimiento de código se ha basado en una metodología iterativa. A continuación, se detallan los pasos seguidos:

1. **Selección del código base:** Se elige un programa base real, que funcione correctamente, pero cuyo rendimiento inicial pueda mejorarse y que actualmente no haga uso de técnicas específicas de optimización. Esta elección permite evaluar de forma clara y práctica los beneficios que aportan las técnicas de paralelización, evitando centrarse únicamente en mejoras algorítmicas básicas.
2. **Análisis de rendimiento (profiling):** Se aplican herramientas de análisis de rendimiento con el objetivo de localizar cuellos de botella y funciones críticas en términos de tiempo de cómputo.
3. **Evaluación de paralelización:** Si los cuellos de botella están relacionados con operaciones de cálculo intensivo, se procede a evaluar si el código es paralelizable y si no existen dependencias insalvables entre los datos.
4. **Reevaluación en caso de restricciones:** En caso de que el código analizado no sea apto para paralelización efectiva, se descarta y se vuelve al paso uno, seleccionando una nueva base de código.
5. **Pruebas de rendimiento:** Una vez identificada una base de código adecuada, se realizan pruebas con distintos niveles de carga (desde casos simples hasta escenarios más exigentes) para medir tiempos de ejecución y consumo de memoria.
6. **Aplicación de optimizaciones:** Con base en los resultados del profiling y de las pruebas, se aplican optimizaciones progresivas. Después de cada optimización, se repiten las mediciones para evaluar su impacto.
7. **Evaluación del rendimiento:** Finalmente, se comparan los resultados obtenidos antes y después de las optimizaciones, calculando el *speedup* alcanzado y evaluando la eficiencia general del proceso de paralelización.

Es importante señalar que los resultados de los tiempos de ejecución presentados se han obtenido mediante el siguiente procedimiento de estimación: cada prueba fue ejecutada cinco veces consecutivas, tras lo cual se descartaron el valor máximo y el mínimo para reducir el impacto de posibles valores atípicos. La media aritmética se calculó a partir de los

tres valores intermedios restantes, proporcionando así una estimación más representativa y estable del tiempo de ejecución real.

1.5. Estructura del documento

El presente Trabajo de Fin de Grado se organiza en seis capítulos principales, además de bibliografía y anexos. A continuación se describe brevemente el contenido de cada capítulo:

- **Capítulo 1 – Introducción:** se presenta la motivación y justificación del proyecto, el contexto en el que se enmarca, los objetivos generales y específicos, la metodología utilizada y la estructura general del documento.
- **Capítulo 2 – Preliminares:** se exponen los conceptos fundamentales de paralelización y concurrencia, los diferentes tipos de paralelismo, las arquitecturas de procesamiento paralelo (CPU multinúcleo y GPU), así como tecnologías relevantes como OpenMP y CUDA y las herramientas de análisis empleadas con el objetivo de presentar los conocimientos teóricos base para el correcto entendimiento del presente **Trabajo de Fin de Grado**.
- **Capítulo 3 – Selección y estudio del sistema a paralelizar:** se realiza un estudio detallado del sistema inicial FUME 2.0, analizando su funcionamiento, el flujo de ejecución, los cuellos de botella detectados y las limitaciones para paralelizarlo. Se justifica el cambio a un sistema basado en ray-tracing y se analiza el nuevo renderizador propuesto.
- **Capítulo 4 – Arquitectura del código base:** se detalla el funcionamiento del algoritmo de ray-tracing, incluyendo la inicialización, renderizado, cálculo de iluminación e intersección de rayos. Se presentan las métricas iniciales de rendimiento y profiling, y se identifican las partes susceptibles de paralelización.
- **Capítulo 5 – Implementación de optimizaciones:** se describen las técnicas aplicadas para paralelizar el renderizador usando OpenMP y CUDA, incluyendo estrategias de optimización, gestión de dependencias, sincronización y diseño de kernels. Finalmente, se presenta el entorno de desarrollo utilizado y se muestran los resultados comparativos entre las versiones secuencial, OpenMP y CUDA.
- **Capítulo 6 – Conclusiones y trabajo futuro:** ofrece una discusión sobre ventajas, limitaciones y costes computacionales, dificultades técnicas, resumen de los objetivos logrados, análisis de impacto y se proponen líneas de trabajo futuro.
- **Bibliografía:** incluye todas las referencias utilizadas durante el desarrollo del trabajo.
- **Anexos:** contiene materiales complementarios, como el informe de originalidad.

Capítulo 2

Preliminares

En este capítulo se exponen los conceptos teóricos y tecnológicos necesarios para comprender el desarrollo del presente proyecto. Se abordan los principios de paralelización y concurrencia, los distintos tipos de paralelismo, las arquitecturas de procesamiento paralelo, y las tecnologías empleadas para la implementación de mejoras y la evaluación del rendimiento: OpenMP, CUDA y herramientas de *profiling*.

2.1. Principios de paralelización y concurrencia

La **paralelización** [4] es una técnica fundamental en la computación de alto rendimiento que consiste en dividir una tarea en múltiples subtareas que pueden ejecutarse de forma simultánea en diferentes unidades de procesamiento. Esta estrategia permite acelerar significativamente el tiempo de ejecución de algoritmos complejos, especialmente en entornos donde la eficiencia computacional es crítica, como simulaciones científicas, procesamiento de imágenes, modelado 3D o aprendizaje automático.

Históricamente, la búsqueda de paralelismo ha sido una respuesta directa a las limitaciones del incremento de frecuencia en los procesadores. Durante décadas, el aumento del rendimiento se logró principalmente mediante el escalado de la frecuencia de reloj. Sin embargo, a partir de mediados de los años 2000, esta tendencia se encontró con barreras físicas (como el calentamiento excesivo y el consumo energético), lo que motivó la adopción masiva de arquitecturas *multicore* y, en consecuencia, la necesidad de explotar el paralelismo tanto a nivel de hardware como de software.

Por otro lado, la **concurrencia** [4] se refiere a la capacidad de un sistema para gestionar múltiples tareas al mismo tiempo. Aunque parecida en concepto, la concurrencia no implica necesariamente ejecución paralela. En un sistema con un único procesador, por ejemplo, las tareas pueden entrelazarse temporalmente mediante *context switching*, permitiendo que parezcan ejecutarse simultáneamente desde la perspectiva del usuario. Sin embargo, la esencia de la concurrencia va más allá de la mera ejecución simultánea, sea paralela o no. Lo que realmente distingue a la concurrencia es una forma de sincronización más flexible y menos estructurada entre las tareas que se ejecutan. En contraste, el **paralelismo** suele implicar una sincronización más rígida y organizada, donde múltiples tareas corren al mismo tiempo en diferentes procesadores o núcleos, coordinadas de manera estricta para

cumplir un objetivo común.

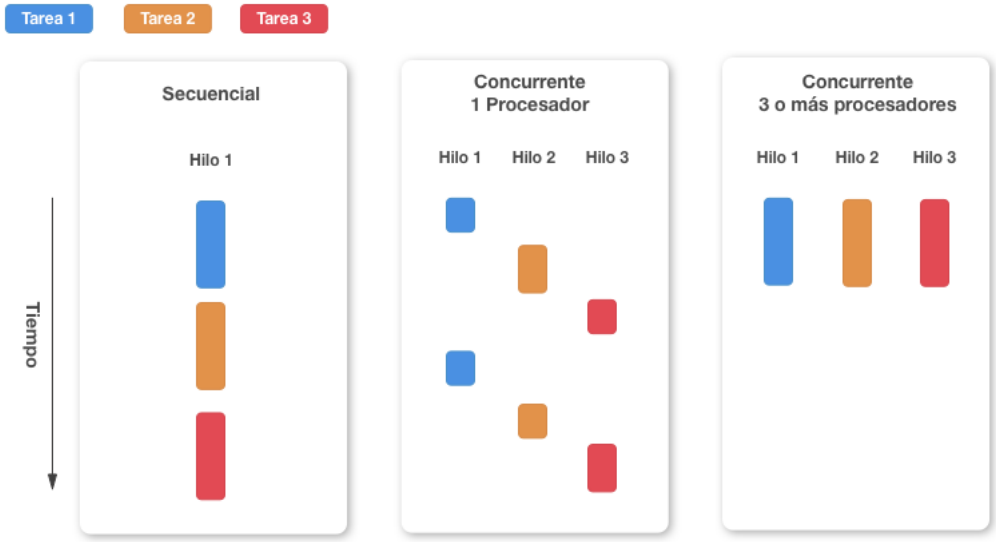


Figura 2.1: Comparación entre ejecución secuencial, concurrencia y paralelismo. Fuente: Make It Real

Ambos conceptos, paralelización y concurrencia, son pilares en el diseño de aplicaciones modernas. Mientras que la paralelización busca mejorar el rendimiento explotando el hardware subyacente, la concurrencia es esencial para gestionar múltiples procesos, eventos o usuarios en sistemas interactivos o distribuidos.

2.1.1. Condiciones de carrera

Un aspecto crítico en la programación concurrente y paralela es el manejo adecuado de los recursos compartidos. Cuando múltiples hilos acceden simultáneamente a una misma variable o estructura de datos sin la debida sincronización, pueden producirse condiciones de carrera (*race conditions*) [5], que resultan en comportamientos impredecibles y errores difíciles de reproducir.

Un ejemplo típico de **condición de carrera** ocurre cuando dos o más hilos intentan actualizar simultáneamente una misma variable compartida, como un contador. Supongamos que inicialmente el contador vale 0 y dos hilos intentan incrementar su valor en 1 al mismo tiempo.

Sin la sincronización adecuada, ambos hilos podrían leer el valor 0, luego ambos sumar 1 independientemente, y finalmente escribir el valor 1 de nuevo. El resultado esperado sería 2, pero debido a la interferencia, el contador queda en 1. Este error es difícil de detectar y reproducir porque depende del orden y el tiempo exacto en que los hilos acceden a la variable compartida.

Por ello, es fundamental usar mecanismos de sincronización, como *mutexes* o *semáforos*, regiones críticas o barreras (`barriers`), que aseguran un acceso controlado a los recursos compartidos [5].

2.1.2. Principios y límites teóricos de la paralelización

Un aspecto clave al analizar el impacto de la paralelización es entender cuáles son los **resultados ideales** frente a los **resultados reales o usuales**. En un escenario ideal, si una tarea se divide entre N núcleos de procesamiento, se esperaría que el tiempo de ejecución se redujera a $1/N$, es decir, un *aceleramiento lineal*. Sin embargo, en la práctica, diversos factores como la necesidad de sincronización, la sobrecarga de comunicación entre hilos, y la parte del código que no puede paralelizarse limitan el rendimiento alcanzable.

Este fenómeno está descrito formalmente por la **Ley de Amdahl** [6], que establece que la mejora máxima en el rendimiento de un sistema al aplicar paralelismo está acotada por la fracción secuencial del programa. La ley se expresa matemáticamente como:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

donde $S(N)$ es la aceleración teórica con N procesadores, y P representa la proporción del programa que puede ejecutarse en paralelo. Esto implica que incluso con infinitos procesadores, el máximo *speedup* está limitado por el código secuencial, lo que resalta la importancia de reducir esta fracción tanto como sea posible.

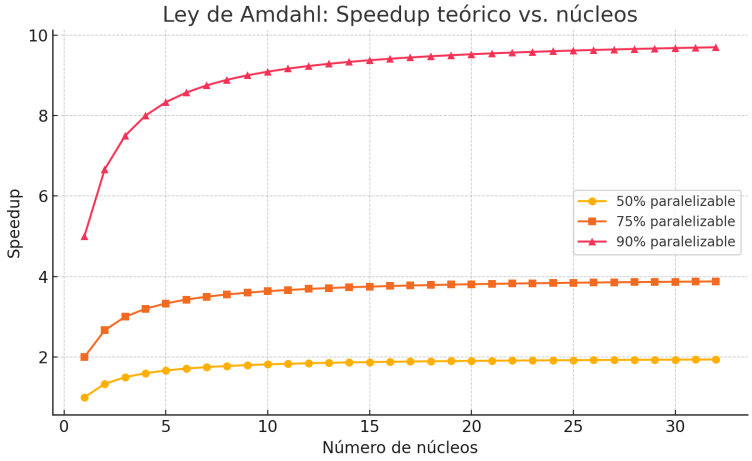


Figura 2.2: Ley de Amdahl, representación gráfica.

En el contexto de este proyecto, tendremos presente la **Ley de Amdahl** a la hora de seleccionar el sistema que posteriormente vamos a paralelizar. Si la parte paralelizable del programa es muy reducida, incluso cuando aplicásemos la paralelización, no observaríamos resultados notables, por lo que sería un motivo de descarte del sistema.

Adicionalmente, uno de los retos más relevantes en la paralelización es el equilibrio entre la granularidad del paralelismo y el coste de la sincronización. Una paralelización demasiado fina puede implicar una sobrecarga de coordinación que neutralice los beneficios del procesamiento paralelo. Por lo tanto, diseñar algoritmos paralelos eficientes requiere comprender a fondo las interdependencias entre tareas, los patrones de acceso a memoria y la arquitectura del sistema objetivo.

2.2. Tipos de paralelismo: datos, tareas, instrucciones

El paralelismo puede clasificarse según el nivel en el que se aplica:

- **Paralelismo de datos:** se aplica la misma operación sobre múltiples elementos de datos de forma simultánea. Es común en operaciones vectoriales o matrices, y se aprovecha en arquitecturas SIMD (Single Instruction, Multiple Data).
- **Paralelismo de tareas:** se divide un programa en tareas independientes que pueden ejecutarse en paralelo. Cada tarea puede realizar una función diferente, y no necesariamente comparten datos.
- **Paralelismo a nivel de instrucciones:** se refiere a la ejecución simultánea de múltiples instrucciones individuales dentro de una sola CPU mediante técnicas como la segmentación de instrucciones (pipelining), ejecución fuera de orden, o múltiples unidades funcionales.

2.3. Arquitecturas de procesamiento paralelo

Las arquitecturas modernas están diseñadas para facilitar la ejecución simultánea de múltiples hilos de ejecución. A continuación, se describen las dos arquitecturas más utilizadas en este proyecto: CPUs multinúcleo y GPUs.

2.3.1. CPUs multinúcleo

Una CPU multinúcleo integra múltiples núcleos de procesamiento dentro de un mismo chip. Cada núcleo puede ejecutar hilos independientes de forma concurrente. Además, las CPUs modernas incluyen técnicas como la ejecución especulativa, predicción de saltos y memoria caché compartida para mejorar el rendimiento.

El paralelismo en CPUs se aprovecha mediante hilos (threads) gestionados por el sistema operativo o bibliotecas como `pthread` y `OpenMP`. Aunque las CPUs ofrecen menor número de núcleos comparadas con las GPUs, su versatilidad, potentes mecanismos de control de flujo y jerarquías de memoria más sofisticadas las hacen ideales para tareas con dependencia de control o baja granularidad.

OpenMP: características, ventajas y casos de uso

OpenMP (Open Multi-Processing) [7] es una API para la programación paralela en arquitecturas de memoria compartida. Se basa en directivas de compilador, funciones de biblioteca y variables de entorno que permiten gestionar fácilmente la creación y sincronización de hilos.

Entre sus características principales se encuentran:

- Facilidad de uso mediante directivas como `#pragma omp parallel`, `for`, `sections`, `schedule`, etc.
- Control de concurrencia mediante constructos como `critical`, `barrier` y `reduction`.
- Gestión de afinidad de hilos, programación de tareas y soporte para jerarquías de paralelismo.

A modo de ejemplo, el siguiente código en C muestra cómo paralelizar un bucle que suma los elementos de un array usando OpenMP:

```
int sum = 0;
#pragma omp parallel for reduction(+:sum) schedule(static)
for (int i = 0; i < N; i++) {
    sum += array[i];
}
```

En este fragmento, la directiva `#pragma omp parallel for` indica que las iteraciones del bucle for a continuación se distribuirán entre múltiples hilos en paralelo. Además, como definimos un schedule estático, todos los hilos recibirán el mismo número de iteraciones desde el principio. La cláusula `reduction(+:sum)` asegura que cada hilo mantenga su propia copia local de `sum` para evitar condiciones de carrera, y al finalizar el bucle, se combinan (reducen con el operador +) los resultados parciales en la variable `sum` final.

Este ejemplo sencillo ilustra cómo OpenMP permite paralelizar código existente con mínimos cambios y sin necesidad de gestionar manualmente la creación y sincronización de hilos.

2.3.2. GPUs y arquitectura CUDA

Las GPUs (Unidades de Procesamiento Gráfico) están diseñadas para ejecutar miles de hilos en paralelo, lo que las hace especialmente eficaces para tareas que se pueden dividir en muchas operaciones similares, como cálculos matemáticos o procesamiento de imágenes. A diferencia de las CPUs, que están optimizadas para ejecutar pocas tareas complejas con alta velocidad, las GPUs priorizan la capacidad de procesar grandes cantidades de datos simultáneamente.

Ejemplo de suma de vectores: código secuencial vs CUDA

Para entender la diferencia entre programación secuencial (clásica) y paralela con CUDA (plataforma de programación paralela creada por NVIDIA. Permite a los desarrolladores utilizar la GPU como GPGPU), veamos cómo se implementa la suma de dos vectores en cada caso.

```
void sumaVectoresCPU(int *A, int *B, int *C, int N) {
    for (int i = 0; i < N; i++) {
        C[i] = A[i] + B[i];  // Sumar elemento a elemento
    }
}
```

En este enfoque, un solo hilo ejecuta la suma, procesando cada elemento uno tras otro. Esto puede ser lento cuando los vectores son muy grandes.

```
// Kernel CUDA: función que se ejecuta en la GPU
__global__ void sumaVectoresCUDA(int *A, int *B, int *C, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x; // Identificador
        único de hilo
```

```
4       if (i < N) { // A menudo se lanzan más hilos que elementos por
            lo que debemos finalizar aquellos fuera de rango
5           C[i] = A[i] + B[i];  // Cada hilo suma un elemento del
                vector
6       }
7   }
```

Aquí, en lugar de un único hilo, miles de hilos se ejecutan simultáneamente en la GPU. Cada hilo calcula la suma de un único elemento, por lo que la operación completa se realiza mucho más rápido para vectores grandes.

Para lanzar el kernel en CUDA, se definen los bloques y el número de hilos por bloque. Por ejemplo, si tenemos 1024 elementos y lanzamos 256 hilos por bloque:

```
1   int N = 1024;
2   int threadsPerBlock = 256;
3   int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
4
5   sumaVectoresCUDA<<<blocksPerGrid, threadsPerBlock>>>(A_dev, B_dev,
        C_dev, N);
```

Jerarquía de memoria en CUDA

CUDA dispone de diferentes tipos de memoria con distintas velocidades y alcances:

- **Memoria global:** accesible por todos los hilos, con alta latencia y gran capacidad.
- **Memoria compartida:** rápida y accesible solo para los hilos dentro de un mismo bloque; se encuentra en el chip y permite la cooperación entre hilos.
- **Memoria local:** privada para cada hilo, pero en realidad está alojada en memoria global, por lo que tiene alta latencia.
- **Memoria constante:** de solo lectura, accesible por todos los hilos, rápida si todos los hilos leen la misma dirección simultáneamente.
- **Registros:** privados y muy rápidos, almacenados dentro de cada núcleo de procesamiento; limitados en cantidad por hilo.

Optimizar el uso de estas memorias y organizar correctamente los hilos es clave para maximizar el rendimiento.

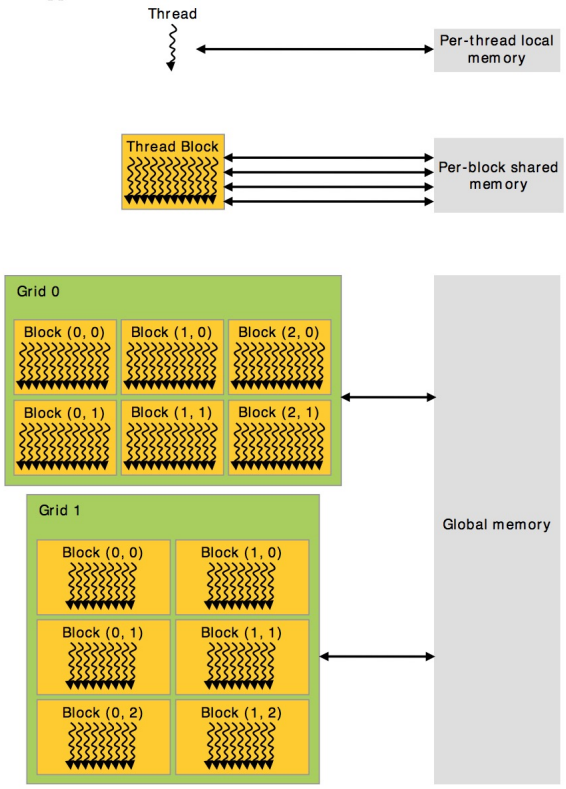


Figura 2.3: Jerarquía de memoria CUDA, representación gráfica. Fuente: ResearchGate

2.4. Herramientas de análisis y *profiling*

Para evaluar el rendimiento de las versiones paralelas del sistema, se utilizaron herramientas de análisis y *profiling* que permiten identificar cuellos de botella y oportunidades de optimización.

Algunas herramientas destacadas son:

- **cProfile:** permite el profiling de programas escritos en Python.
- **gprof:** generador de perfiles de ejecución para aplicaciones compiladas en C/C++.
- **Valgrind (Callgrind):** permite obtener estadísticas detalladas sobre llamadas a funciones y uso de recursos.
- **KCachegrind:** interfaz gráfica para visualizar los perfiles generados por Callgrind, facilitando el análisis de rendimiento.
- **nvprof / Nsight Compute:** herramientas de NVIDIA para analizar el rendimiento de kernels CUDA.
- **perf:** herramienta del kernel de Linux para monitorear eventos de hardware y software.

- **Intel VTune:** permite analizar la eficiencia de paralelización en CPUs.

Estas herramientas fueron fundamentales para guiar el proceso de optimización y validar las mejoras obtenidas.

Capítulo 3

Selección y estudio del sistema a paralelizar

3.1. Búsqueda preliminar de sistemas candidatos

Dado que el objetivo principal de este **Trabajo de Fin de Grado** es obtener mejoras visibles de rendimiento en programas computacionalmente intensivos mediante el uso de técnicas de paralelización, el primer paso consistió en identificar un sistema base que pudiera servir como campo de pruebas. Para ello, se realizó una búsqueda inicial de proyectos de código abierto que cumpliesen los siguientes criterios:

- Presentaran tareas computacionalmente costosas y que éstas representasen los principales cuellos de botella en cuanto a tiempo de ejecución y que fuesen susceptibles de ser aceleradas mediante paralelización.
- Contaran con documentación razonable y estructura modular que facilitase su comprensión y modificación.

Tras evaluar varias opciones, se seleccionó inicialmente el sistema FUME, una herramienta de procesamiento de emisiones atmosféricas utilizada en contextos científicos. Su elección se justificó por la modularidad de su arquitectura, la variedad de operaciones que realiza y su uso en entornos de producción reales.

Esta selección permitió iniciar un análisis detallado del sistema con el fin de identificar posibles puntos de paralelización. Los resultados de ese estudio se presentan en la siguiente sección.

3.2. Análisis del sistema FUME

3.2.1. Descripción funcional del sistema

FUME [8] (Flexible Universal Processor for Modeling Emissions) es un programa de software libre que facilita la preparación de datos sobre emisiones contaminantes para el estudio y la predicción de la calidad del aire. Los modelos matemáticos utilizados para simular cómo los contaminantes, como gases o partículas, se dispersan en la atmósfera,

conocidos como modelos de transporte químico, requieren datos específicos y organizados sobre las emisiones para funcionar correctamente.

FUME procesa datos de emisiones que pueden provenir de múltiples formatos y estar desorganizados, unificándolos y transformándolos en un formato claro y adecuado para que dichos modelos puedan interpretarlos y utilizarlos eficazmente.

Además, FUME permite ajustar estos datos en función de la ubicación, el tiempo y el tipo de contaminante, desglosándolos en detalles más específicos, como la distribución horaria, la segmentación por zonas urbanas, o la clasificación química necesaria para simular con precisión el comportamiento de los contaminantes en la atmósfera.

Finalmente, FUME es una herramienta flexible que puede aplicarse a distintas escalas, desde estudios urbanos hasta investigaciones continentales, y que es capaz de manejar datos muy variados, lo que la convierte en un recurso valioso para científicos y autoridades encargadas del monitoreo y la gestión de la calidad del aire.

El sistema es utilizado en instituciones como el Instituto Checo de Hidrometeorología y participa en proyectos europeos relevantes, lo cual demuestra su aplicabilidad y robustez. El repositorio de código de FUME puede encontrarse en FUME (https://github.com/FUME-dev/fume).

3.2.2. Estudio del flujo de ejecución

El flujo de ejecución de FUME se estructura como una serie de pasos secuenciales que procesan los datos de entrada hasta producir las salidas necesarias para los modelos químicos. Algunos de los pasos clave del workflow son:

1. **input.init_static**: Importa datos estáticos y de configuración.
2. **input.import_sources**: Importa las fuentes de emisión.
3. **case.prepare_conf**: Inicializa la proyección geométrica y la cuadrícula. Este paso es obligatorio si se va a ejecutar cualquier paso posterior.
4. **case.create_new_case**: Inicializa el caso de estudio.
5. **transformations.prepare**: Inicializa las colas de transformación.
6. **transformations.run**: Aplica las colas de transformación sobre los grupos de fuentes definidos.
7. **case.process_point_sources**: Realiza cálculos específicos para fuentes puntuales.
8. **case.collect_meteorology**: Importa los datos meteorológicos necesarios para los cálculos de modelos externos.
9. **case.process_case_spec_time**: Calcula y aplica factores de especiación química y de desagregación temporal.
10. **case.process_vertical_distributions**: Calcula los factores de distribución vertical según la estructura vertical de la cuadrícula de salida, las categorías de emisión y las especies químicas.
11. **case.preproc_external_models**: Preprocesamiento para modelos externos, es decir, cálculos necesarios solo una vez, usualmente relacionados con procesos específicos.
12. **case.run_external_models**: Ejecuta los modelos externos para los pasos temporales definidos.
13. **postproc.run**: Ejecuta los postprocesadores seleccionados.

La siguiente figura muestra como funciona el flujo de trabajo de FUME.

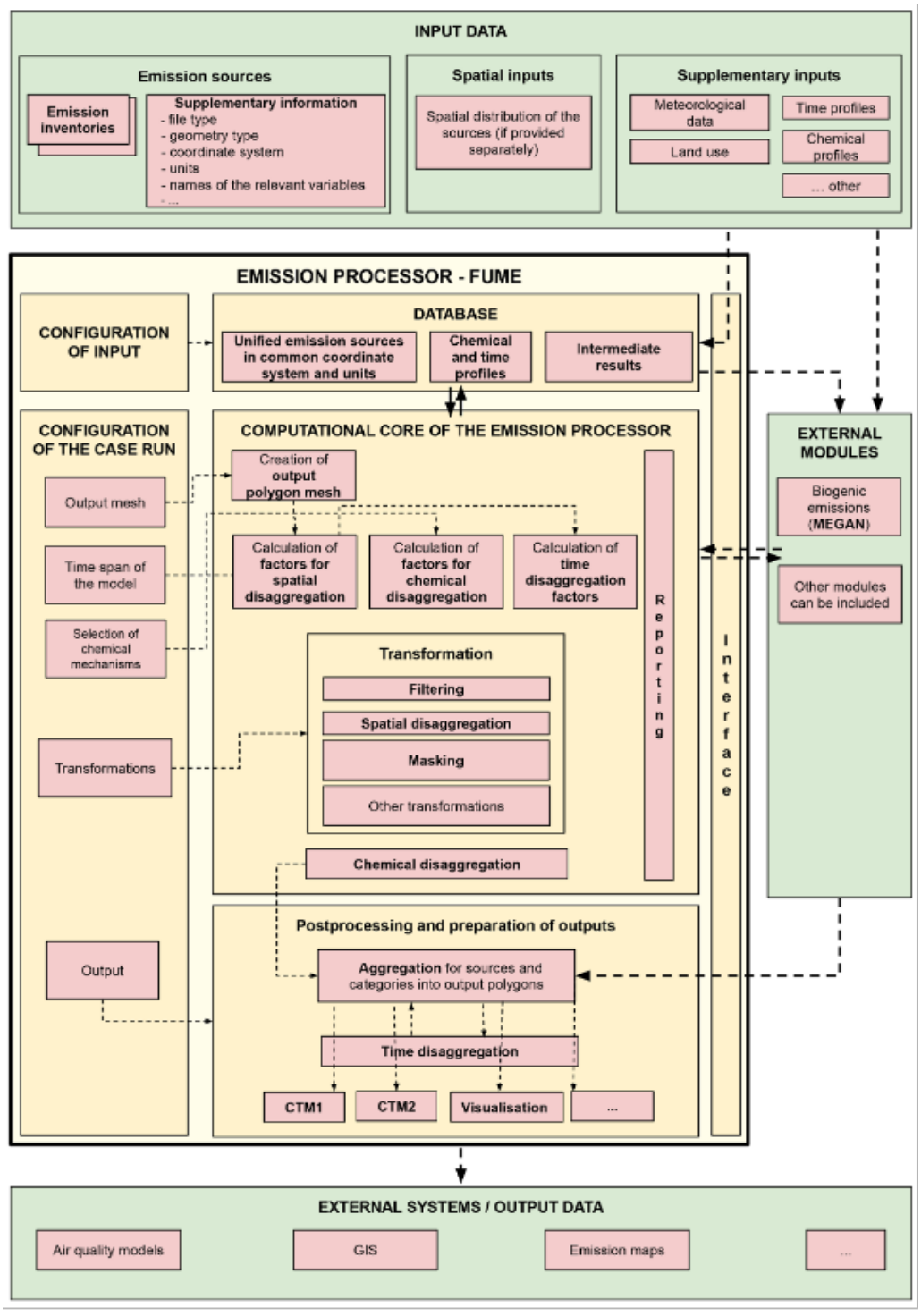


Figura 3.1: FUME workflow.

3.2.3. Cuellos de botella identificados mediante *profiling*

Durante las primeras semanas del proyecto se realizó un análisis exhaustivo del código con herramientas de *profiling*, como `cProfile`, `line_profiler` y trazas personalizadas sobre dos casos de prueba [9]. El objetivo era localizar las funciones responsables del mayor tiempo de ejecución y ver si estas funciones se demoraban tanto debido a cuellos de botella relacionados con el cómputo y uso de CPU o si por el contrario los altos tiempos de ejecución no estaban relacionados con el uso de CPU y el cálculo sino con factores como la E/S.

Los resultados fueron concluyentes: más del 75 % del tiempo total de ejecución se dedicaba a operaciones de acceso a la base de datos especialmente a través de la librería `psycopg2` sin suponer ningún estrés de cómputo en CPU. A continuación se muestra una tabla resumen extraída del análisis:

ncalls	**tottime**	**percall**	**cumtime**	**percall**	**filename:line(func)**
523672	489.4	0.0009346	489.9	0.0009355	<method execute of psycopg2.extensions.cursor objects>
16	208.2	13.01	208.2	13.01	<method callproc of psycopg2.extensions.cursor objects>
683	100.9	0.1478	209.8	0.3071	netcdf.py:266 (receive_area_emiss_by_species_and_category)
2729638	52.96	1.94e-05	87.8	3.216e-05	utils.py:81 (_StartCountStride)
10156315	14.01	1.379e-06	37.62	3.704e-06	ogr.py:4412 (GetField)
684	13.49	0.01973	13.5	0.01974	<method fetchmany of psycopg2.extensions.cursor objects>
51879307	13.14	2.532e-07	20.55	3.961e-07	function_base.py:348 (iterable)
518335	12.28	2.369e-05	12.28	2.369e-05	<built-in method osgeo._ogr.Geometry_ExportToWkt>
53047834	7.529	1.419e-07	7.529	1.419e-07	<built-in method builtins.iter>
2729638	7.007	2.567e-06	8.544	3.13e-06	utils.py:451 (_out_array_shape)
-	<0.5	<0.1	<0.1	<0.1	6668 funciones más

Aunque la tabla no proporciona información directa sobre el uso de CPU por parte de cada función, es posible deducir que las llamadas asociadas a `psycopg2` —como `execute`, `callproc` o `fetchmany`— no suponen una carga de cómputo significativa. Estas funciones están relacionadas con operaciones de E/S, en concreto con el envío de consultas SQL y la recepción de datos desde un servidor de base de datos externo. Este tipo de operaciones son

intrínsecamente bloqueantes y dependientes de latencias de red o de acceso a disco, pero no requieren apenas procesamiento intensivo por parte de la CPU. De hecho, en muchos sistemas, durante estas esperas la CPU permanece inactiva o cambia de contexto para ejecutar otras tareas. Por tanto, el elevado tiempo acumulado en estas funciones no se debe a la ejecución de cálculos complejos, sino al tiempo que el programa pasa esperando a que lleguen los datos desde la base de datos.

A raíz de este diagnóstico, se implementaron diversas estrategias de mejora enfocadas en reducir la latencia y el coste computacional asociado al acceso a la base de datos. Las principales fueron:

- **Simplificación de consultas SQL**: Se revisaron las consultas más costosas para reducir su complejidad. Esto incluyó la eliminación de subconsultas innecesarias, el uso explícito de índices y la reestructuración de sentencias `JOIN` para evitar operaciones costosas como `NESTED LOOP` sobre tablas grandes.
- **Uso de buffers y escritura en lotes**: Se implementó un sistema de almacenamiento temporal en memoria (buffer) para acumular los registros a insertar y procesarlos en bloques más grandes. Esto permitió utilizar funciones como `execute_batch()` de la librería `psycopg2.extras`, que permite enviar múltiples instrucciones `INSERT`, `UPDATE` o `DELETE` en un solo viaje a la base de datos. Esta técnica es significativamente más eficiente que usar múltiples llamadas a `cursor.execute()`, especialmente cuando el número de operaciones es elevado. Por ejemplo, en lugar de ejecutar:

```
for record in data:
cursor.execute("INSERT INTO tabla VALUES (%s, %s)", record)
```

  Se reemplazó por:

```
execute_batch(cursor,"INSERT INTO tabla VALUES (%s, %s)", data)
```

  Lo que permite reducir el número de viajes cliente-servidor y mejorar la velocidad global por reducción de latencias del proceso de inserción o actualización de la base de datos.

Estas estrategias tenían como objetivo principal minimizar al máximo los tiempos de E/S para poder comenzar a paralelizar funciones como `receive_area_emiss_by_species_and_category`, que realizan operaciones computacionales susceptibles de paralelización. De esta forma, se buscaba demostrar el potencial que estas técnicas de paralelización podían ofrecer.

Sin embargo, aunque estos métodos permitieron mejorar parcialmente el rendimiento (con ganancias de aproximadamente un 10 %), no lograron resolver el problema fundamental: el sistema seguía limitado por una arquitectura basada principalmente en operaciones de E/S. Esto reducía significativamente el beneficio de paralelizar las secciones con alta carga computacional, ya que, incluso en el mejor de los casos, la aceleración máxima teórica según la **Ley de Amdahl** no superaría un factor de 1.5. Como el objetivo principal del proyecto era lograr mejoras significativas en el rendimiento global del programa, no tenía sentido enfocarnos en reducir los tiempos de CPU cuando estos representaban solo una pequeña fracción del tiempo total de ejecución. Por esta razón, se decidió finalmente cambiar la base del proyecto.

3.2.4. Limitaciones en observar resultados tangibles tras la paralelización en FUME

Aunque FUME presenta una estructura modular, su diseño está profundamente ligado al acceso secuencial a bases de datos. Las principales barreras para obtener mejoras significativas en rendimiento fueron:

- **Bajo uso de CPU**: Las operaciones computacionales representan solo una pequeña parte del tiempo total de ejecución, por lo que su paralelización, aún siendo ideal, no aportaría mejoras relevantes.
- **Cuello de botella en la base de datos**: La mayoría del tiempo se consume en consultas SQL, donde el paralelismo tiene poco impacto debido a que muchas actualizaciones son bloqueantes y dependen unas de otras.
- **Granularidad de las tareas**: Las operaciones son demasiado pequeñas o son muy interdependientes, dificultando extraer paralelismo efectivo.
- **Altos accesos a memoria**: Para la paralelización en GPU, las latencias de acceso a grandes volúmenes de datos hubieran ralentizado aún más el programa, anulando las posibles mejoras.

Por estas razones, se concluyó que para lograr mejoras notables sería necesario rehacer funciones SQL o reorganizar la estructura de la base de datos, tareas que, aunque relevantes, se alejan del objetivo principal de este Trabajo de Fin de Grado: aplicar técnicas de paralelización para obtener mejoras notables de rendimiento y analizar su impacto en el rendimiento del sistema completo.

3.2.5. Justificación del cambio de sistema

Llegados a este punto del desarrollo, y ante las limitadas posibilidades de obtener unos buenos resultados tras la paralelización sobre un sistema ampliamente condicionado por operaciones de E/S, así como el creciente alejamiento del objetivo inicial, se optó por reorientar la base del proyecto hacia un entorno más adecuado para la implementación de técnicas de paralelización.

Concretamente, se seleccionó una implementación de un renderizador de imágenes 3D mediante *ray-tracing* destacable por su claridad estructural y su elevado potencial para ser optimizada mediante paralelización.

Esta modificación no representa un abandono del trabajo realizado previamente sobre FUME, sino una decisión estratégica. El análisis efectuado —especialmente mediante herramientas de profiling y el estudio de cuellos de botella— proporciona una base sólida para justificar este cambio. El objetivo original del **Trabajo de Fin de Grado**, centrado en la exploración de técnicas de paralelización y la evaluación de su impacto en el rendimiento, se mantiene sin alteraciones, aunque ahora aplicado sobre una base de código que nos permita obtener unos resultados mucho más notables, puesto que como se verá a continuación, en este sistema el cómputo sí es el principal cuello de botella.

3.3. Criterios de selección del nuevo sistema

Tras constatar las limitaciones del sistema FUME para aplicar paralelización efectiva —fundamentalmente debido al peso de operaciones de entrada/salida (E/S) y acceso a base de datos—, se optó por cambiar la base del proyecto hacia un entorno más propicio para poder observar las mejoras que introducen las diferentes técnicas de computación paralela.

Los criterios que guiaron la elección del nuevo sistema fueron:

- **Alta proporción de operaciones computacionales puras**: sistemas donde la CPU sea el principal cuello de botella, no el disco ni la red.
- **Bajo acoplamiento entre tareas**: ideal para aplicar paralelismo a nivel de hilos o procesos sin necesidad de excesiva sincronización.
- **Código base reducido y comprensible**: necesario para implementar y controlar cambios de manera eficiente durante el desarrollo del TFG.
- **Implementación en C**: A diferencia del código de FUME, escrito en Python, y dónde habríamos tenido que reescribir las funciones a paralelizar en C para poder aplicar las técnicas mencionadas, la implementación del ray-tracer estaba escrita en C, un lenguaje de bajo nivel, cercano al hardware, con control explícito sobre memoria y concurrencia.
- **Aplicación visualizable**: permite evaluar rápidamente el impacto de las optimizaciones mediante el contraste de las imágenes generadas y los tiempos de ejecución empleados.

Con estos criterios, se seleccionó un renderizador por *ray-tracing* escrito en C [10], conocido por su estructura simple, claridad y potencial para paralelización intensiva.

Capítulo 4

Arquitectura del Código Base

El motor de renderizado [11] utilizado como base secuencial para este proyecto fue implementado por Jakob Maier [10] y sigue una arquitectura monolítica con una estructura modular por funciones. Se basa en las implementaciones en C++ como las de *scratchapixel.com* [12] y *smallpt* [13]. En específico, utiliza el algoritmo conocido como *Stochastic Path Tracing with Russian Roulette Termination*. Permite al usuario elegir los parámetros de entrada de la imagen a renderizar; alto (px), ancho (px) y número de muestras/pixel. Está dividido en varios componentes clave, que interactúan entre sí a través de estructuras compartidas:

- **Escena**: Representada por un array de objetos (‘Object[]‘), que incluye esferas, luces y elementos de entorno. Estos objetos contienen propiedades como posición, radio, color, emisión y tipo de material.
- **Generación de rayos**: Se realiza mediante funciones que construyen rayos desde la cámara hacia la escena. El archivo `raytracer.c` implementa funciones como `get_camera_ray` para este propósito.
- **Intersección**: El sistema determina si un rayo colisiona con algún objeto utilizando la función `intersect`, que devuelve información detallada del punto de impacto a través de estructuras de tipo `Hit`.
- **Sombreado**: Una vez detectada una intersección, se calcula el color del píxel mediante modelos de iluminación y técnicas de *path-tracing* (`trace_path`). También se consideran reflexiones y refracciones según el tipo de material.
- **Renderizado**: El núcleo del renderizado está en la función `render`, que recorre cada píxel de la imagen, lanza múltiples rayos por píxel y promedia los resultados.
- **Salida**: Los datos del buffer de la imagen (en adelante, framebuffer) se almacenan como una imagen .PNG utilizando la librería `stb_image_write.h`.

Cada función está diseñada para facilitar la modificación y la extensión, aunque comparten memoria (como la escena y el framebuffer), lo que permite una fácil paralelización, como se hace mediante OpenMP para aprovechar múltiples núcleos de CPU.

4.1. Funcionamiento del algoritmo de Ray-Tracing

El algoritmo de *Ray-Tracing* implementado consta principalmente de dos fases iniciales llamadas desde `main`: la inicialización de la cámara con `init_camera` y la generación de la imagen con `render`.

4.1.1. Inicialización de la cámara (`init_camera`)

Esta función configura los parámetros necesarios para definir la cámara virtual desde la cual se emitirá cada rayo. Se calculan:

- El campo de visión vertical (aproximadamente 60^o).
- La proporción de aspecto, basada en las dimensiones de la imagen de salida.
- Los vectores `forward`, `right` y `up` para orientar la cámara en el espacio 3D.
- Los vectores `horizontal` y `vertical` que definen el tamaño del plano de imagen (viewport).
- El punto `lower_left_corner` que representa la esquina inferior izquierda del plano de imagen desde donde se dispararán los rayos.

Esta configuración es fundamental para generar los rayos correctos que atraviesan cada píxel de la imagen.

4.1.2. Renderizado de la escena (`render`)

Esta función recorre cada píxel del framebuffer, y para cada uno genera múltiples muestras para realizar un muestreo anti-aliasing (técnica que suaviza los bordes y reduce artefactos visuales promediando múltiples muestras por píxel).

- Por cada muestra se calcula un par de coordenadas `(u, v)` normalizadas en el rango [0,1] que representan la posición relativa en el viewport.
- Se obtiene un rayo desde la cámara con `get_camera_ray`, utilizando las coordenadas `(u, v)` para determinar la dirección correcta.
- Para cada rayo, se llama a `trace_path` para calcular la iluminación que recibe ese rayo, considerando interacciones con los objetos de la escena.
- Los colores devueltos por cada muestra se acumulan y promedian para obtener el color final del píxel.
- Finalmente, el color se aplica una corrección gamma para mejorar la percepción visual y se almacena en el framebuffer.

4.1.3. Trazado de rayos y cálculo de iluminación (`trace_path`)

Esta función es el núcleo recursivo del algoritmo, que realiza las siguientes tareas:

- Incrementa el contador global de rayos disparados.
- Verifica si el rayo actual intersecta con algún objeto de la escena.

- Si no hay intersección o se alcanza la profundidad máxima de recursión, devuelve el color de fondo.
- En caso de intersección, obtiene la información del punto de impacto, el material y sus propiedades (color, emisión, tipo de reflexión).
- Se aplica un esquema de *russian roulette*, cuyo funcionamiento se explicará más adelante, para decidir probabilísticamente si se sigue calculando la iluminación indirecta.
- Según el tipo de material, se calcula:
  - Reflexión especular: se genera un rayo reflejado.
  - Refracción: se calcula un rayo refractado considerando el índice de refracción.
  - Difusa: se genera un rayo aleatorio en el hemisferio alrededor de la normal para simular difusión.
- Se llama recursivamente a `trace_path` para obtener la contribución de iluminación indirecta.
- Se combinan las contribuciones directas y las recursivas para obtener el color final en el punto.

Aplicación del método *Russian Roulette*

En el trazado de rayos, cuando se calcula la iluminación mediante la función `path_trace`, las llamadas recursivas para simular la dispersión de la luz pueden crecer exponencialmente, lo que genera un coste computacional muy elevado. Para controlar esto, se usa el método conocido como *Russian Roulette* [14], que consiste en probabilísticamente decidir si se continúa o se detiene la trayectoria del rayo.

Sea $\mathbf{L}$ la radiancia a calcular para un rayo, y sea $\mathbf{a} = (a_x, a_y, a_z)$ el albedo (color) del objeto interceptado. Definimos la probabilidad de continuar la trayectoria como:

$$p = \text{máx}\{a_x, a_y, a_z\}$$

Entonces, se genera un número aleatorio $r \in [0, 1)$. Si $r < p$, la trayectoria continúa y se escala el albedo para mantener la energía promedio:

$$\mathbf{a}' = \frac{\mathbf{a}}{p}$$

Si $r \geq p$, la trayectoria termina y se devuelve la emisión directa $\mathbf{e}$.

Esto se traduce en el código de la función `trace_path`:

```
double prob = MAX(albedo.x, MAX(albedo.y, albedo.z));
if (random_double(seed) < prob)
  albedo = vec3_scalar_mult(albedo, 1 / prob);
else
  return emission;
```

De esta forma, la técnica de *Russian Roulette* permite:

- Reducir el número esperado de llamadas recursivas.
- Mantener el valor esperado de la radiancia invariante (al escalar por $1/p$).
- Controlar el balance entre precisión y rendimiento.

Además, en el código, si la trayectoria continúa, se calcula la siguiente dirección del rayo según las propiedades del material (reflexión, refracción o difusión), y se sigue trazando recursivamente la radiancia.

4.1.4. Intersección de rayos (`intersect` y funciones auxiliares)

Se comprueba la intersección del rayo con cada objeto usando algoritmos geométricos específicos; para esferas se usa la solución algebraica del problema de intersección rayo-esfera.

Se determina el punto más cercano de intersección válido, su normal, coordenadas de textura y demás parámetros relevantes. Se evalúa la sombra mediante un rayo de luz hacia la fuente para determinar si el punto está iluminado o en sombra, y se modula la iluminación en consecuencia.

4.2. Resumen simplificado del funcionamiento del algoritmo de Ray-Tracing

El algoritmo de *Ray-Tracing* es una técnica para generar imágenes realistas simulando cómo la luz viaja y rebota en una escena 3D. Los siguientes puntos resumen de una manera simplificada los pasos del ray-tracer que han sido explicados en profundidad en las secciones que hemos visto anteriormente y finaliza con un pequeño ejemplo ilustrativo de este funcionamiento.

1. **Preparar la cámara:** Primero, se configura una cámara virtual que decide desde dónde 'vemos' la escena y hacia dónde se envían los rayos de luz.
2. **Generar la imagen:** Para cada punto o píxel de la imagen, se envían varios rayos desde la cámara hacia la escena, buscando calcular la luz que llega a ese punto.
3. **Seguir los rayos:** Cada rayo puede chocar con objetos (como una pelota o una pared). Al chocar, el algoritmo calcula cómo la luz se refleja, refracta (como el vidrio), o se difunde, y puede disparar más rayos para simular estas interacciones.
4. **Controlar el cálculo:** Para evitar que los rayos se multipliquen infinitamente (y el cálculo se haga interminable), se usa un método llamado *Russian Roulette* que decide probabilísticamente cuándo dejar de seguir un rayo.
5. **Resultado final:** Al combinar la información de todos estos rayos para cada píxel, se obtiene la imagen final que simula efectos como sombras, reflejos y transparencias con mucho realismo.

Ejemplo simple:

Imagina que estás en una habitación con una linterna (la cámara) y una pelota de cristal (un objeto). Apuntas la linterna y ves cómo la luz golpea la pelota, parte de esa luz atraviesa la pelota y otra parte se refleja. El algoritmo de *Ray-Tracing* simula exactamente

ese comportamiento, enviando rayos desde tus ojos (la cámara) y calculando dónde y cómo se reflejan o pasan a través de los objetos para crear una imagen muy realista.

4.2.1. Resumen del flujo de ejecución

A continuación puede verse el gráfico de las llamadas más importantes en el flujo de ejecución del renderizador. Este gráfico ha sido generado con la herramienta *valgrind* con la opción de `-tool=callgrind` y la herramienta de visualización *kcachegrind*.

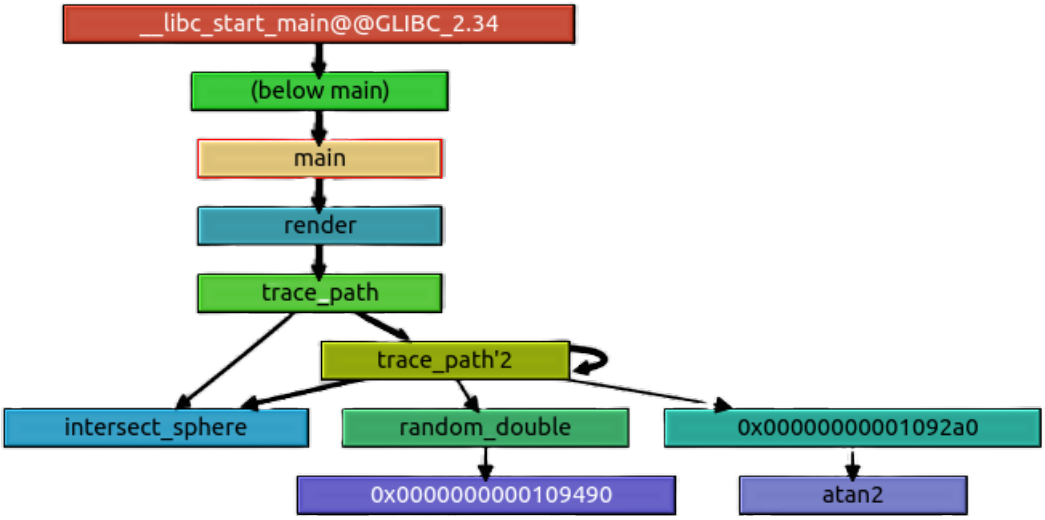


Figura 4.1: Flujo de ejecución de las llamadas más importantes del renderizador.

El código completo de la versión secuencial puede encontrarse en Ray-Tracer Secuencial (https://github.com/davidmorilla/raytracer/tree/main/secuencial), además de en el repositorio original [11].

4.3. Métricas de rendimiento y profiling inicial

Para establecer una línea base que permita comparar mejoras posteriores, se evaluó el rendimiento del renderizador original en ejecución secuencial utilizando herramientas de profiling como *Intel VTune*, *perf* y *valgrind/kcachegrind*. Se seleccionaron dos métricas clave para el análisis:

- **Tiempo total de ejecución**: mide el tiempo transcurrido desde la carga de la escena hasta la generación del archivo final. Esta métrica es fundamental para evaluar la eficiencia temporal del programa y su capacidad para procesar la tarea en un tiempo razonable.
- **Consumo de memoria**: cuantifica las asignaciones y liberaciones de memoria en megabytes durante la ejecución. Controlar esta métrica es esencial para identificar posibles fugas de memoria, optimizar el uso de recursos y garantizar que el programa pueda ejecutarse en sistemas con limitaciones de memoria.

Se realizaron 3 pruebas con diferentes parámetros de entrada: una prueba sencilla, poco exigente, de 500×500 píxeles, con 10 esferas y 200 muestras por píxel; una intermedia, de 1024×720 píxeles, con 10 esferas y 1000 muestras por píxel; y una exigente, de 1920×1080 píxeles, con 10 esferas y 8192 muestras por píxel.

4.3.1. Resultados según tiempo de ejecución

A continuación se muestran los resultados de cada prueba según la métrica de tiempo de ejecución:

Cuadro 4.1: Tiempo total de ejecución y principales hotspots de la prueba sencilla

Función	Módulo	CPU Time	% del Tiempo CPU
Tiempo total de ejecución	—	**97.814 s**	—
`intersect_sphere`	raytracer	26.600 s	27.2 %
`vec3_dot`	raytracer	14.440 s	14.8 %
`intersect`	raytracer	11.950 s	12.2 %
`__atan2`	libm.so.6	8.760 s	9.0 %
`__ieee754_sqrt`	libm.so.6	4.410 s	4.5 %
`[Others]`	N/A*	31.650 s	32.4 %

Cuadro 4.2: Tiempo total de ejecución y principales hotspots de la prueba intermedia

Función	Módulo	CPU Time	% del Tiempo CPU
Tiempo total de ejecución	—	**1426.128 s**	—
`intersect_sphere`	raytracer	388.824 s	27.3 %
`vec3_dot`	raytracer	209.807 s	14.7 %
`intersect`	raytracer	178.096 s	12.5 %
`__atan2`	libm.so.6	128.870 s	9.0 %
`__ieee754_sqrt`	libm.so.6	64.589 s	4.5 %
`[Others]`	N/A*	455.873 s	32.0 %

Debido al largo tiempo de ejecución que tarda la prueba exigente con la versión secuencial, nos vemos obligados a interpolar los resultados del tiempo estimado de ejecución en base a los dos casos anteriores. Los datos de progreso y tiempo los sacamos de la traza de ejecución del programa; el siguiente fragmento de código ubicado en la función `render` imprime la barra de progreso junto con el tiempo que lleva ejecutando.

```
1  int tic = clock();
2  for (uint y = 0; y < options->height; y++)
3  {
4      if (y % 10 == 0)
5      {
6        double percentage = ((double)y / (double)options->height) *
             100.0;
7        int p = str_len - (percentage / 100.0) * str_len;
```

```
        printf("[%s%s] %0.02f %%\t[%d seconds]\n", done + (p), todo
            + (str_len - p), percentage, (int)((clock() - tic)/(
            CLOCKS_PER_SEC)));
    }
    [...]
}
```

El resultado de graficar esta traza se puede ver en la siguiente figura:

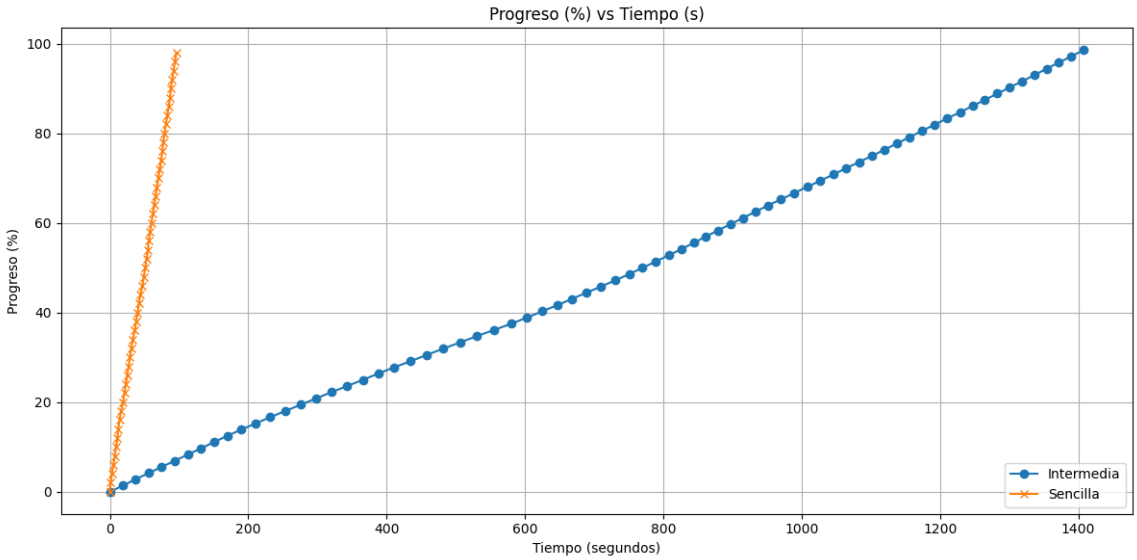


Figura 4.2: Correlación entre progreso vs tiempo de ejecución.

Tras analizar las curvas de progreso vs tiempo de cada caso, podemos observar como existe una posible **correlación lineal** entre ambas variables. Lo comprobamos realizando un estudio estadistico de *Pearson* sobre las variables progreso-tiempo de ejecución.

```
x = np.array(progress_percent)
y = np.array(time_seconds)
x2 = np.array(progress_2)
y2 = np.array(time_2)

# Correlaciones de Pearson
corr, p_value = pearsonr(x, y)
corr2, p_value2 = pearsonr(x2, y2)

# Crear gráfico
plt.figure(figsize=(8, 5))
plt.scatter(x, y, label="Sencilla", color="steelblue")
plt.scatter(x2, y2, label="Intermedia", color="orange")

# Ajuste lineal para ambos
m, b = np.polyfit(x, y, 1)
plt.plot(x, m*x + b, color="crimson", linestyle="--", label="Ajuste
    lineal Sencilla")

```

```
m2, b2 = np.polyfit(x2, y2, 1)
plt.plot(x2, m2*x2 + b2, color="green", linestyle="--", label="
    Ajuste lineal Intermedia")

# Anotar correlaciones y p-values
textstr = f"Sencilla:\nr = {corr:.4f}\np = {p_value:.4e}\n\
    nIntermedia:\nr = {corr2:.4f}\np = {p_value2:.4e}"
plt.text(0.02, 0.75, textstr, transform=plt.gca().transAxes,
    fontsize=10, verticalalignment='top', bbox=dict(boxstyle="round"
    , facecolor='white', alpha=0.8))

# Etiquetas y formato
plt.title("Correlación entre progreso (%) y tiempo (s)")
plt.xlabel("Progreso (%)")
plt.ylabel("Tiempo de ejecución (s)")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Para las pruebas sencilla e intermedia obtenemos el siguiente output:

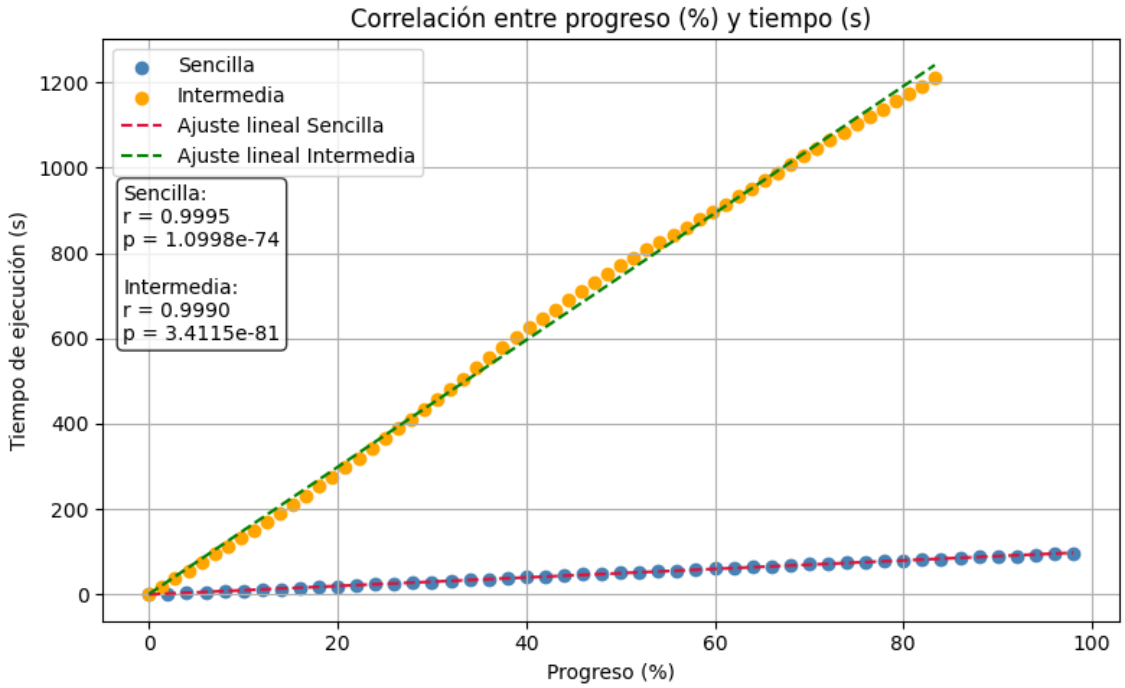


Figura 4.3: *p-values* y correlación entre progreso vs tiempo de ejecución.

A partir de la Figura 4.3, se observa que en ambos casos (sencillo e intermedio) los datos muestran un comportamiento fuertemente alineado con una tendencia lineal. Esto se confirma numéricamente a través del coeficiente de correlación de *Pearson* r, que en ambos casos es cercano a 1, indicando una correlación positiva muy fuerte entre el porcentaje de progreso y el tiempo de ejecución.

Además, los valores de *p-value* obtenidos son extremadamente bajos (muy por debajo del umbral común de significancia estadística de 0.05), lo cual permite rechazar la hipótesis nula que asumiría que no existe relación lineal entre ambas variables. Por tanto, la relación observada no es producto del azar y es estadísticamente significativa.

La presencia de esta correlación fuerte y significativa implica que el progreso del algoritmo crece de forma aproximadamente lineal con respecto al tiempo, tanto en la versión sencilla como en la intermedia. Esto es especialmente útil para interpolar el comportamiento del algoritmo en la prueba exigente, dado que nos permite estimar su progreso y tiempo de ejecución en base a los patrones observados en los casos más simples.

En resumen, los resultados respaldan la hipótesis inicial de que existe una correlación lineal entre el progreso del proceso y el tiempo de ejecución, y lo hacen de forma cuantitativa y visualmente consistente.

De esta manera, tras obtener la traza:

```
[----------------------------------------] 0.00 % [0 seconds]
[=---------------------------------------] 0.19 % [74 seconds]
[=---------------------------------------] 0.37 % [148 seconds]
[=---------------------------------------] 0.56 % [221 seconds]
[=---------------------------------------] 0.74 % [295 seconds]
[=---------------------------------------] 0.93 % [369 seconds]
[=---------------------------------------] 1.11 % [443 seconds]
[=---------------------------------------] 1.30 % [517 seconds]
[=---------------------------------------] 1.48 % [591 seconds]
[=---------------------------------------] 1.67 % [665 seconds]
[=---------------------------------------] 1.85 % [739 seconds]
[=---------------------------------------] 2.04 % [813 seconds]
[=---------------------------------------] 2.22 % [887 seconds]
[=---------------------------------------] 2.41 % [961 seconds]
[==--------------------------------------] 2.59 % [1034 seconds]
```

tenemos que al interpolar cual sería el tiempo de ejecución real obtenemos unos 39921 segundos:

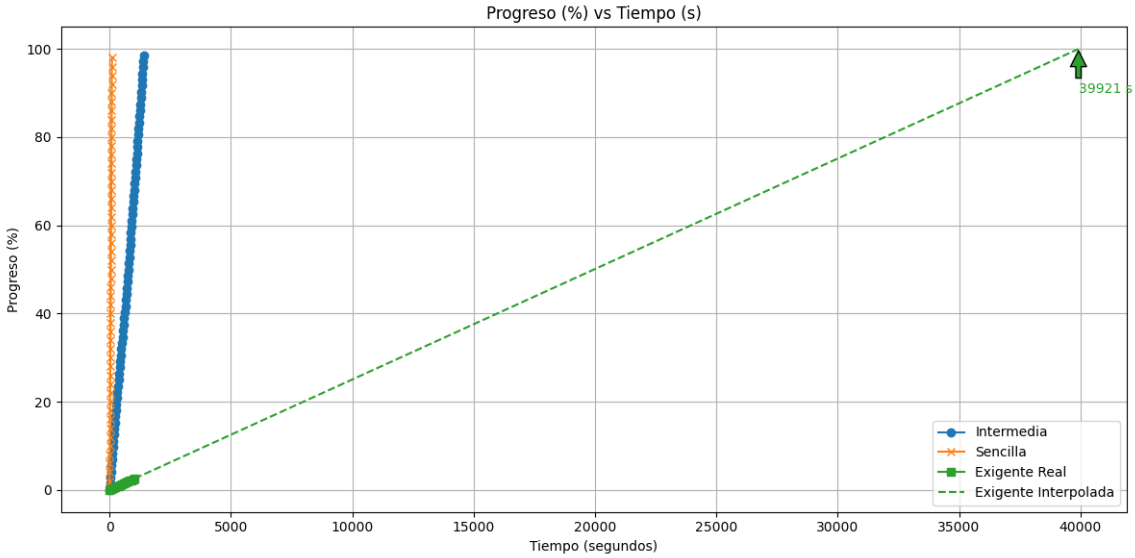


Figura 4.4: Interpolacion del tiempo de ejecución de la prueba exigente.

4.3.2. Resultados según el consumo de memoria

A continuación se muestran los resultados de cada prueba según la métrica de consumo de memoria:

Cuadro 4.3: Uso de memoria de la prueba sencilla

Métrica	Valor
Tamaño total de memoria asignada	9.9 MB
Tamaño total de memoria liberada	9.9 MB
Número total de asignaciones	65,526
Número de hilos	1
Tiempo en pausa	0 s

Cuadro 4.4: Uso de memoria de la prueba intermedia

Métrica	Valor
Tamaño total de memoria asignada	18.1 MB
Tamaño total de memoria liberada	18.1 MB
Número total de asignaciones	61,615
Número de hilos	1
Tiempo en pausa	0 s

4.3.3. Imágenes generadas

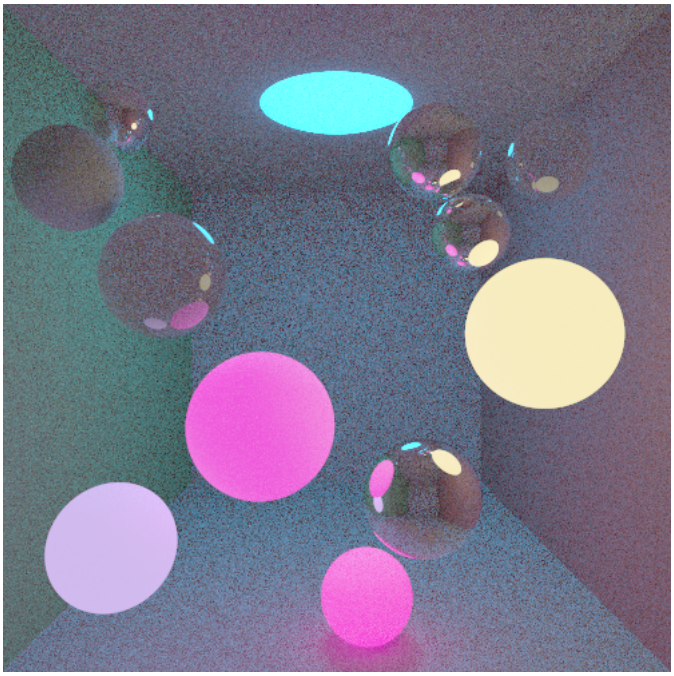


Figura 4.5: Imagen resultado de la prueba sencilla.

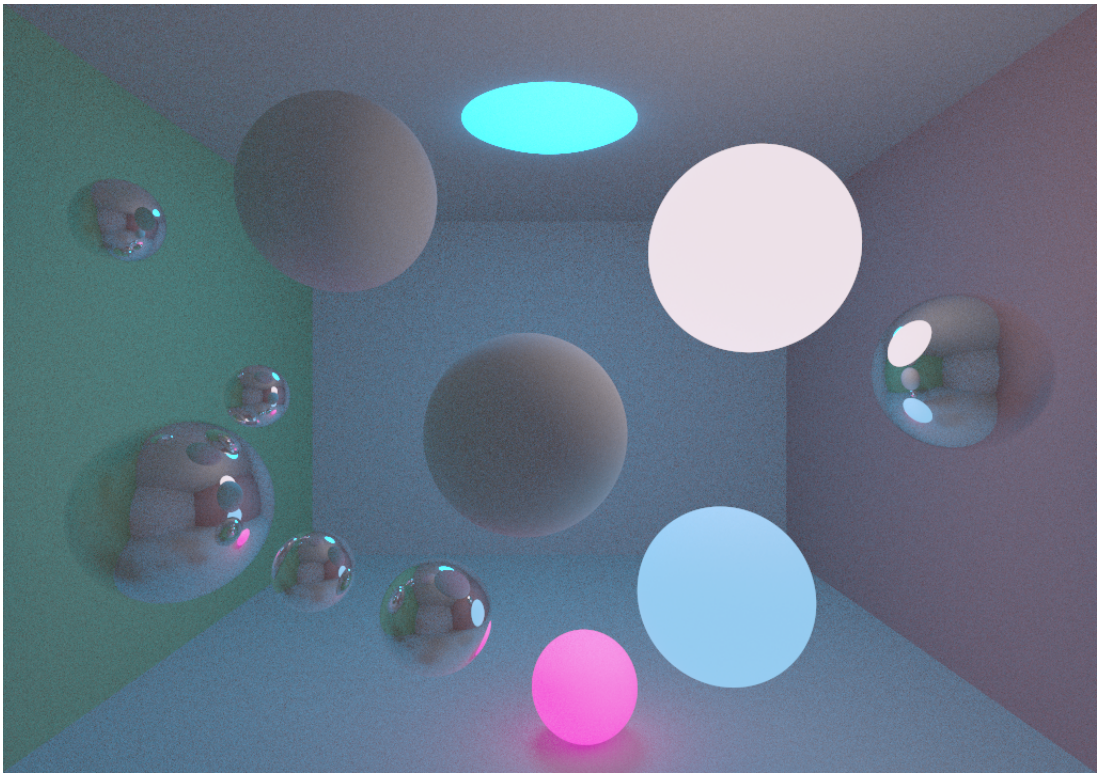


Figura 4.6: Imagen resultado de la prueba intermedia.

4.4. Identificación de partes paralelizables

El análisis del perfil de ejecución reveló los principales *hotspots* donde se concentra el tiempo de CPU, lo que guía la identificación de las partes paralelizables del código:

- **Función `intersect_sphere`**: Es la función con mayor consumo de CPU (27.3 %). Este cálculo de intersección entre rayos y esferas es computacionalmente intensivo pero se puede paralelizar fácilmente porque cada evaluación es independiente.
- **Función `vec3_dot`**: Operación matemática de producto punto entre vectores (14.7 %), muy frecuente y paralelizable sin dependencia.
- **Función `intersect`**: Otro *hotspot* importante (12.5 %) que realiza pruebas de intersección, también sin dependencia entre píxeles.
- **Funciones de librería matemática `__atan2` y `__ieee754_sqrt`**: Estas funciones suman un 13.5 % del tiempo y son llamadas dentro del proceso de renderizado, por lo que su paralelización es implícita al paralelizar las llamadas a funciones superiores.

Se identifican además áreas sensibles o no paralelizables:

- **Acceso concurrente al búfer de imagen**: La escritura en el búfer debe coordinarse para evitar condiciones de carrera, aunque es posible paralelizar garantizando que cada hilo escriba en regiones exclusivas, por lo que lo catalogamos como **área sensible**.
- **Inicialización y carga de datos de la escena**: Operaciones secuenciales no paralelizables que deben realizarse antes del renderizado pero que no representan cuellos de botella en el rendimiento.

Con base en este análisis, la paralelización se enfoca principalmente en el bucle de renderizado por píxeles, donde las funciones de intersección y cálculos vectoriales constituyen el núcleo del procesamiento intensivo y pueden distribuirse eficientemente entre múltiples hilos.

Capítulo 5

Implementación de optimizaciones

5.1. Paralelización mediante OpenMP

5.1.1. Optimizaciones secuenciales previas

Función `intersect_sphere`

A continuación se detallan las mejoras hechas a la función `intersect_sphere`. Ambas versiones implementan el mismo algoritmo para calcular la intersección de un rayo con una esfera. Sin embargo, la versión optimizada es significativamente más rápida debido a varias mejoras en la implementación. A continuación, se describen las diferencias clave:

- **Eliminación de funciones auxiliares:**
  La versión original utiliza funciones como `vec3_sub` y `vec3_dot`, que implican llamadas adicionales con cierto overhead. En la versión optimizada, estas operaciones se realizan directamente en línea, lo que reduce la latencia y permite una mejor optimización por parte del compilador. Es decir, pasamos de tener esto:

```
vec3 L = vec3_sub(center, ray->origin);
double tca = vec3_dot(L, ray->direction);
```

  a tener lo siguiente:

```
double Lx = center.x - ray->origin.x;
double Ly = center.y - ray->origin.y;
double Lz = center.z - ray->origin.z;
double tca = Lx * ray->direction.x + Ly *
        ray->direction.y + Lz * ray->direction.z;
```

- **Reducción de condiciones y retornos tempranos:**
  La versión original emplea condicionales explícitos (`if`). Se eliminan, en la medida de lo posible, los saltos condicionales lo que mejora la predictibilidad del flujo de ejecución. Además, se hacen retornos tempranos una vez las condiciones son falsas lo que ahorra tiempo de ejecución en comparaciones lógicas posteriores.

Función `intersect`

La función `intersect` determina si un rayo intersecta alguno de los objetos (esferas) en la escena. A continuación se explican las diferencias clave entre la versión original y la versión optimizada, junto con las razones de su mayor eficiencia:

- **Eliminación de funciones intermedias:** La versión original utilizaba llamadas explícitas a funciones auxiliares como `point_at()` y `vec3_normalize()`, lo que añade sobrecarga por llamada a función. En la versión optimizada, estas funciones se han insertado directamente(*inlined*) en el código principal, lo que elimina la sobrecarga por llamadas y permite al compilador realizar más optimizaciones (por ejemplo, reordenar instrucciones o eliminar cálculos redundantes). El código pasa de:

```
if (intersect_sphere(ray, objects[i].center, objects[i].radius
    , &local) && local.t < min_t){
      min_t = local.t;
      local.object_id = i;
      local.point = point_at(ray, local.t);
      local.normal= vec3_normalize(vec3_sub(local.point,
         objects[i].center));
      local.u = atan2(local.normal.x, local.normal.z) / (2 * PI
         ) + 0.5;
      local.v = local.normal.y * 0.5 + 0.5;
    }
```

  a lo siguiente:

```
    if (intersect_sphere(ray, objects[i].center, objects[i].
       radius, &local) && local.t < min_t){
    min_t = local.t;
    local.object_id = i;
    vec3 temp = vec3_add(ray->origin,
    vec3_scalar_mult(ray->direction, local.t));
    vec3 n = vec3_sub(temp, objects[i].center);
    double inv_len = 1.0 / sqrt(n.x*n.x + n.y*n.y + n.z*n.z);
       // norm. vector length
    n.x *= inv_len;
    n.y *= inv_len;
    n.z *= inv_len;
    local.point = temp;
    local.normal = n;
    local.u = atan2(n.x, n.z)*(1.0 / (2.0 * PI)) + 0.5;
    local.v = local.normal.y * 0.5 + 0.5;
```

- **Cálculo manual del vector normal:** En lugar de normalizar el vector con una función genérica, se realiza la normalización de forma explícita usando una inversa de raíz cuadrada. Esto permite evitar la creación de vectores intermedios y posibles accesos a memoria innecesarios. Además, se evitan variables temporales adicionales en memoria (por ejemplo, el vector normal intermedio).

- **Cálculo de coordenadas de textura optimizado:** El valor de la coordenada `u` se sigue calculando con `atan2`, pero con constantes precalculadas en lugar de realizar la división al final.

Al eliminar funciones intermedias, reducir llamadas a funciones externas y escribir directamente las operaciones matemáticas en línea, la versión optimizada minimiza la sobrecarga computacional y mejora la *localidad de datos*, lo que conlleva una mejora sustancial del rendimiento en tiempo de ejecución, especialmente en escenas con muchos objetos.

5.1.2. Estrategia general aplicada

La paralelización del ray-tracer se basa en dividir la carga de trabajo del renderizado de píxeles entre varios hilos utilizando la librería OpenMP. Cada hilo procesa un subconjunto de píxeles, ejecutando de manera concurrente y paralela el trazado de rayos para cada uno. El objetivo principal es aprovechar el paralelismo a nivel de píxel, ya que cada cálculo de color de píxel es independiente de los demás, lo que minimiza las dependencias y facilita la escalabilidad.

Se utiliza las directivas `#pragma omp parallel` y `#pragma omp for` para distribuir el bucle principal que recorre todos los píxeles. Además, se emplea un esquema de programación dinámico para balancear la carga, dada la variabilidad del coste de renderizado por píxel.

5.1.3. Control de dependencias y sincronización

Las dependencias principales del código son las variables `ray_count` e `intersection_test_count`. Estas variables son incrementadas en uno cada vez que se llama a `trace_path` en el caso de `ray_count` y cada vez que se llama a `intersect_sphere` en el caso de `intersection_test_count`. Son las variables encargadas de mostrar al usuario el número de intersecciones y de rayos analizados durante el renderizado. Como al paralelizar tenemos varios hilos modificando el valor de estas variables podríamos tener multitud de condiciones de carrera si no las manejamos correctamente. Para ello vamos a convertir estas variables en arrays con un tamaño definido correspondiente al número de threads que se utilicen. De esta manera, cada hilo puede ir aumentando el valor de su índice del array sin sobreescribir el de otros hilos. Al final del todo, podemos manejar una sección crítica que sume todos los resultados en el índice cero de cada variable. Las variables son inicializadas a cero por el hilo con identificador 0 mediante el uso de `calloc`. Debemos añadir un `#pragma omp barrier` tras la inicialización puesto que todos los hilos deben esperar a que haya memoria reservada para estas variables antes de modificarlas. Si no sincronizásemos tras la inicialización, podríamos caer en errores de tipo *segmentation fault* que suponen accesos ilegales a direcciones de memoria que el sistema operativo no ha asignado a nuestro proceso.

La inicialización quedaría de la siguente manera:

```
#pragma omp parallel
{
  if (omp_get_thread_num() == 0) {
    // Solo el hilo 0 ejecuta esto
    ray_count = calloc(omp_get_num_threads(), sizeof(long long));
    intersection_test_count = calloc(omp_get_num_threads(), sizeof(
       long long));
}
// Todos los hilos esperan aquí hasta que hilo 0 termine
#pragma omp barrier
[...]
```

```
}
```

A su vez, la sección crítica iría tras el cierre del bucle que estamos paralelizando:

```
#pragma omp critical
      {
        if (thread_id!=0){
          ray_count[0]+=ray_count[thread_id];
          intersection_test_count[0]+=
          intersection_test_count[thread_id];
        }
      }
```

Dado que cada píxel se calcula independientemente, no existen dependencias de datos entre iteraciones del bucle principal, lo que simplifica el paralelismo.

Sin embargo, se debe asegurar que la semilla del generador de números aleatorios sea distinta por hilo para evitar correlaciones en las muestras. Para ello, se puede derivar una semilla específica por hilo a partir de una semilla inicial común, por ejemplo, combinándola con el `thread_id`. Esto es importante porque utilizar la misma semilla en todos los hilos produciría secuencias idénticas, mientras que emplear semillas completamente independientes podría introducir problemas adicionales, como una distribución no controlada o falta de reproducibilidad entre ejecuciones. Esto lo conseguimos sumando a la semilla inicial un múltiplo grande del identificador de cada hilo:

```
int thread_id = omp_get_thread_num();
int seed = master_seed + thread_id * 9973;
```

5.1.4. Paralelización del bucle principal de píxeles

El código paralelo se encuentra en la función `render()`, donde el renderizado se realiza sobre un array unidimensional (resultado de fusionar los bucles de x e y de la versión secuencial) que recorre todos los píxeles:

```
#pragma omp parallel
{
  if (omp_get_thread_num() == 0) {
    ray_count = calloc(omp_get_num_threads(), sizeof(long long));
    intersection_test_count = calloc(omp_get_num_threads(), sizeof(
       long long));
  }
  #pragma omp barrier
  int thread_id = omp_get_thread_num();
  int seed = master_seed + thread_id * 9973;
  #pragma omp for schedule(dynamic)
  for (uint i = 0; i < options->height*options->width; i++){
    uint x= i%options->width, y= i/options->width;
    if (thread_id==0 && x == 0){
       double percentage = 100.0 * y / (double)(options->height);
       int p = (percentage / 100.0) * str_len;
       printf("[%.*s%.*s] %0.02f %%\t[%d seconds]\n", p, done,
          str_len - p, todo, percentage, (int)((clock() - tic)/ (
          omp_get_num_threads()*CLOCKS_PER_SEC)));
```

```
17      }
18      REAL X = 0.0, Y = 0.0, Z = 0.0;
19      for (uint s = 0; s < options->samples; s++){
20          double u = (double)(x + random_double(&seed)) / ((double)
                options->width - 1.0);
21          double v = (double)(y + random_double(&seed)) / ((double)
                options->height - 1.0);
22          Ray ray = get_camera_ray(camera, u, v);
23          vec3 sample_pixel=trace_path(&ray, objects, n_objects, 0, &
                seed);
24          X += sample_pixel.x;
25          Y += sample_pixel.y;
26          Z += sample_pixel.z;
27      }
28          vec3 pixel = vec3_scalar_mult(VECTOR(X, Y, Z), 1.0 / (double
                )options->samples);
29          float inv_gamma = 1.0f / gamma;
30          uint a = (y * options->width + x) * 3;
31          framebuffer[a + 0] = (uint8_t)(255.0 * CLAMP(pow(pixel.x,
                inv_gamma)));
32          framebuffer[a + 1] = (uint8_t)(255.0 * CLAMP(pow(pixel.y,
                inv_gamma)));
33          framebuffer[a + 2] = (uint8_t)(255.0 * CLAMP(pow(pixel.z,
                inv_gamma)));
34  }
35  #pragma omp critical
36  {
37     if (thread_id!=0){
38       ray_count[0]+=ray_count[thread_id];
39       intersection_test_count[0]+=
40       intersection_test_count[thread_id]; }
41     }
42  }
```

El paralelismo con OpenMP en este código se estructura de la siguiente forma:

- `#pragma omp parallel`: Se crea un equipo de hilos que ejecutan el bloque de código en paralelo. Cada hilo obtiene su propio `thread_id` mediante `omp_get_thread_num`.
- Inicialización condicional: Solo el hilo 0 (maestro) realiza la asignación de memoria para los contadores `ray_count` e `intersection_test_count`, reservando espacio para cada hilo. Los demás hilos esperan en la barrera `#pragma omp barrier` hasta que esta inicialización termina.
- Variables privadas: Cada hilo genera su semilla pseudoaleatoria (`seed`) basada en el `master_seed` y su `thread_id`, garantizando independencia en la generación de números aleatorios.
- `#pragma omp for schedule(dynamic)`: Divide el bucle `for` entre los hilos, asignando iteraciones de forma dinámica para equilibrar la carga, especialmente cuando el tiempo por iteración puede variar.

  Se eligió la directiva `#pragma omp for schedule(dynamic)` porque permite una mejor distribución de la carga de trabajo entre los hilos en un contexto donde el

coste de cada iteración del bucle es impredecible. Cada iteración procesa un píxel, pero el tiempo de cómputo asociado puede variar considerablemente debido al uso de muestreo estocástico (Monte Carlo) y trazado de caminos recursivo mediante la función `trace_path`. Esta función genera rayos secundarios en función de las interacciones con la escena, lo cual introduce un alto grado de variabilidad entre píxeles.

Con `schedule(static)`, las iteraciones se reparten equitativamente entre los hilos al inicio de la ejecución del bucle, pero como algunas tardan mucho más que otras, se produce un desbalance: algunos hilos terminan temprano y quedan inactivos mientras otros continúan trabajando. La estrategia `schedule(guided)` intenta mejorar este comportamiento disminuyendo progresivamente el tamaño de los bloques asignados a medida que avanza la ejecución, pero puede seguir siendo ineficiente si las iteraciones costosas están dispersas o no se concentran al principio del dominio.

En cambio, `schedule(dynamic)` permite que los hilos soliciten nuevas tareas conforme terminan las anteriores, adaptándose dinámicamente a la carga real. Este enfoque logra un mejor balance y mantiene todos los hilos ocupados durante la mayor parte del tiempo de ejecución. Aunque introduce una ligera sobrecarga administrativa por la asignación dinámica de tareas, esta es despreciable frente al tiempo computacional elevado de cada iteración, que involucra varias muestras por píxel y cálculos matemáticos complejos.

Por lo tanto, en este escenario específico, la programación dinámica del bucle permite una ejecución paralela más eficiente y reduce significativamente el tiempo total de renderizado.

- Dentro del bucle, cada hilo procesa un subconjunto distinto de píxeles (índice `i`), calculando la posición `x` e `y` del píxel en la imagen.
- El hilo 0 imprime periódicamente el progreso basado en la posición actual en la imagen, para informar sobre el avance del renderizado.
- Para cada píxel, se realiza un muestreo múltiple (`options->samples`) con valores aleatorios, generando rayos y acumulando los colores resultantes para obtener un suavizado.
- Finalmente, el color calculado se aplica en el framebuffer correspondiente al píxel.
- `#pragma omp critical`: Al finalizar el procesamiento, los hilos actualizan de forma segura los contadores globales `ray_count` e `intersection_test_count` mediante una sección crítica para evitar condiciones de carrera.

Este esquema permite que el trabajo pesado del renderizado se distribuya eficientemente entre los núcleos disponibles, aprovechando la paralelización para acelerar el cálculo sin interferencias entre hilos.

Aunque habría sido posible obtener un mayor rendimiento modificando en mayor profundidad la lógica de los cálculos, una de las ventajas principales de OpenMP es que permite equilibrar la mejora en el *speedup* con una modificación muy ligera del código original. Esta sección se ha centrado precisamente en mostrar ese equilibrio y la eficacia de dicha aproximación.

El código que incluye mis modificaciones para hacer uso de OpenMP puede encontrarse en Ray-Tracer OpenMP (https://github.com/davidmorilla/raytracer/tree/main/openMP).

5.1.5. Evaluación de rendimiento con OpenMP

A continuación se muestran los resultados de las ejecuciones del código optimizado con OpenMP según las métricas seleccionadas.

Resultados según tiempo de ejecución

Cuadro 5.1: Resumen de tiempos para la prueba sencilla con OpenMP

Tiempo y configuración			
Elapsed Time:	5.667 s	Total Thread Count:	16
CPU Time:	87.570 s	Paused Time:	0 s
Top Hotspots			
Function	**Module**	**CPU Time**	**% of CPU Time**
intersect	raytracer_opt	39.670	45.3 %
intersect_sphere	raytracer_opt	11.081	12.7 %
trace_path	raytracer_opt	10.221	11.7 %
memcpy	raytracer_opt	6.799	7.8 %
render._omp_fn.0	raytracer_opt	5.040	5.8 %
[Others]	N/A*	14.759	16.9 %

Cuadro 5.2: Resumen de tiempos para la prueba intermedia con OpenMP

Tiempo y configuración			
Elapsed Time:	76.782 s	Total Thread Count:	16
CPU Time:	1193.427 s	Paused Time:	0 s
Top Hotspots			
Function	**Module**	**CPU Time**	**% of CPU Time**
intersect	raytracer_opt	529.938 s	44.4 %
intersect_sphere	raytracer_opt	151.472 s	12.7 %
trace_path	raytracer_opt	140.851 s	11.8 %
memcpy	raytracer_opt	93.582 s	7.8 %
render._omp_fn.0	raytracer_opt	73.029 s	6.1 %
[Others]	N/A*	204.554 s	17.1 %

Cuadro 5.3: Resumen de tiempos para la prueba exigente con OpenMP

Tiempo y configuración			
Elapsed Time:	2002.517 s	Total Thread Count:	16
CPU Time:	31358.687 s	Paused Time:	0 s
Top Hotspots			
Function	**Module**	**CPU Time**	**% of CPU Time**
intersect	raytracer_opt	14330.912 s	45.7 %
intersect_sphere	raytracer_opt	5550.485 s	17.7 %
trace_path	raytracer_opt	3073.150 s	9.8 %
memcpy	raytracer_opt	1787.444 s	5.7 %
___atan2	libm.so.6	1693.368 s	5.4 %
[Others]	N/A*	4954.670 s	15.8 %

Resultados según consumo de memoria

Cuadro 5.4: Consumo de memoria en la prueba sencilla

Allocation Size:	9.8 MB
Deallocation Size:	9.8 MB
Allocations:	65,357
Total Thread Count:	16
Paused Time:	0 s

Cuadro 5.5: Consumo de memoria en la prueba intermedia

Allocation Size:	18.0 MB
Deallocation Size:	18.0 MB
Allocations:	58,838
Total Thread Count:	16
Paused Time:	0 s

Cuadro 5.6: Consumo de memoria en la prueba exigente

Allocation Size:	33.2 MB
Deallocation Size:	33.2 MB
Allocations:	44,854
Total Thread Count:	16
Paused Time:	0 s

Imágenes generadas

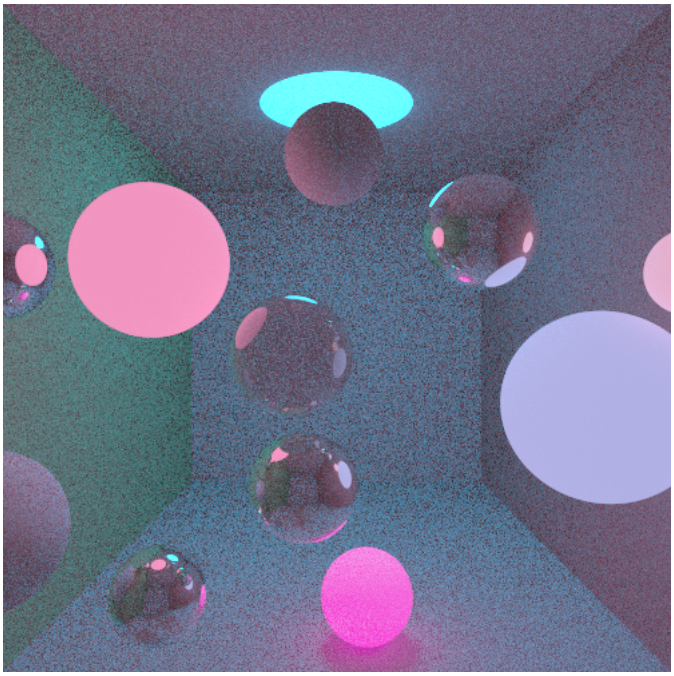


Figura 5.1: Imagen resultado de la prueba sencilla usando OpenMP.

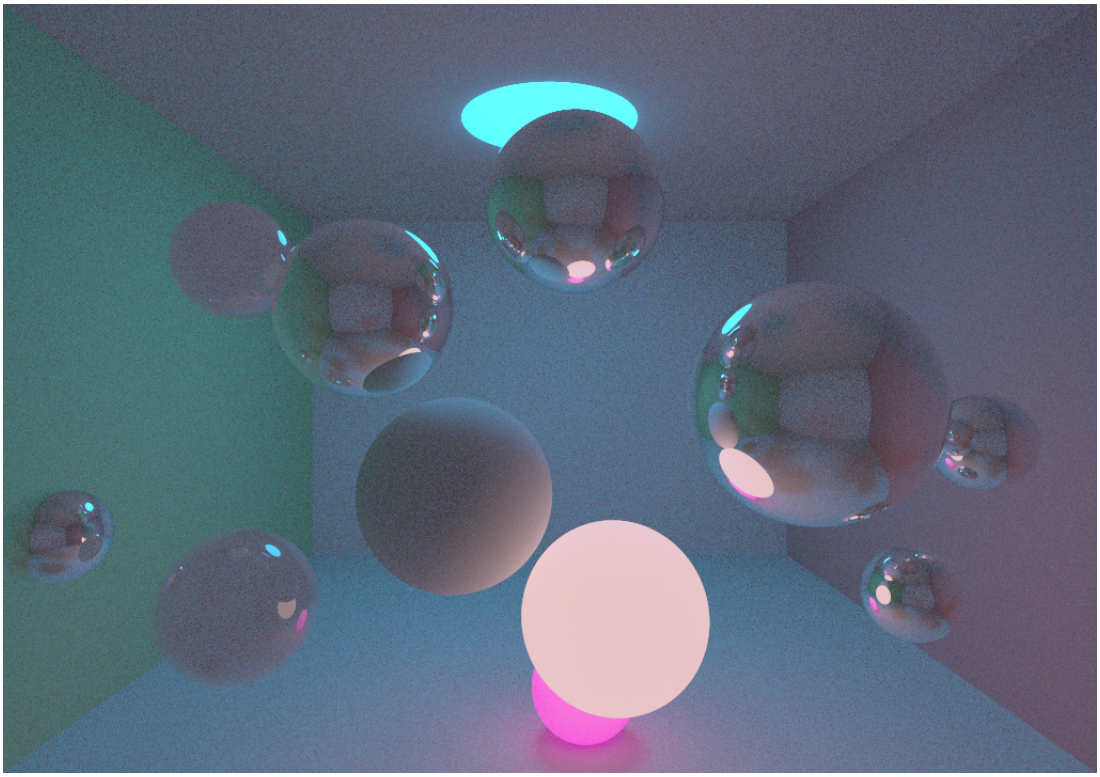


Figura 5.2: Imagen resultado de la prueba intermedia usando OpenMP.

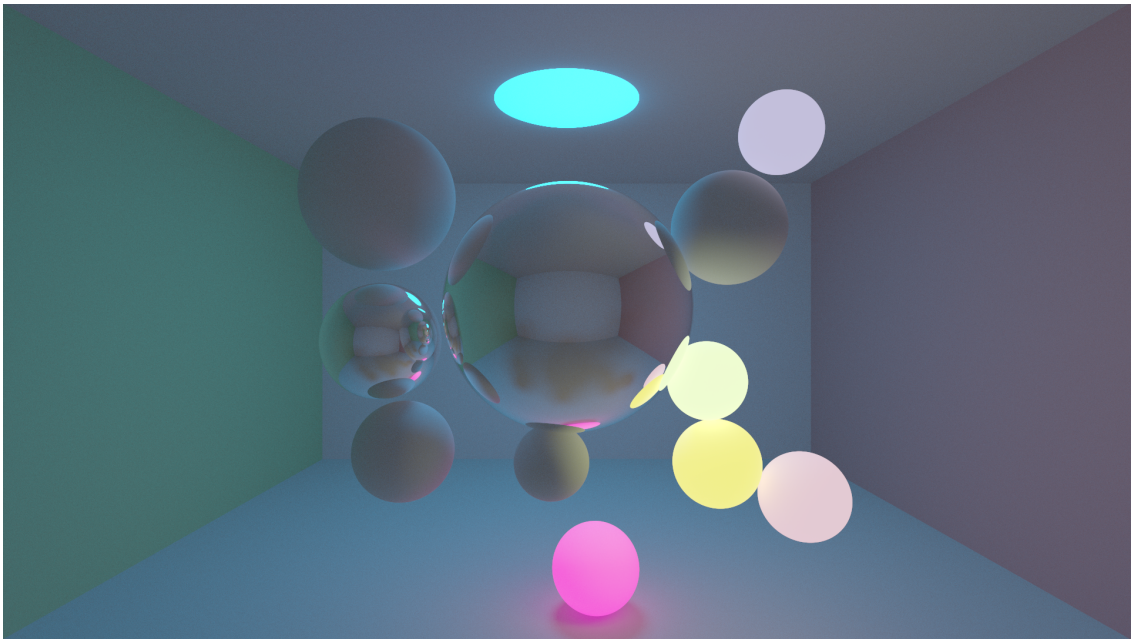


Figura 5.3: Imagen resultado de la prueba exigente usando OpenMP.

Comparación gráfica de tiempos de ejecución y speedup

En la Figura 5.4, se presenta una comparación visual entre los tiempos de ejecución obtenidos en tres pruebas (*Sencilla*, *Intermedia* y *Exigente*) tanto en su versión secuencial como en su versión optimizada con OpenMP. Además, se muestra el *speedup* logrado en cada prueba mediante una línea roja sobrepuesta, referenciada al eje vertical derecho.

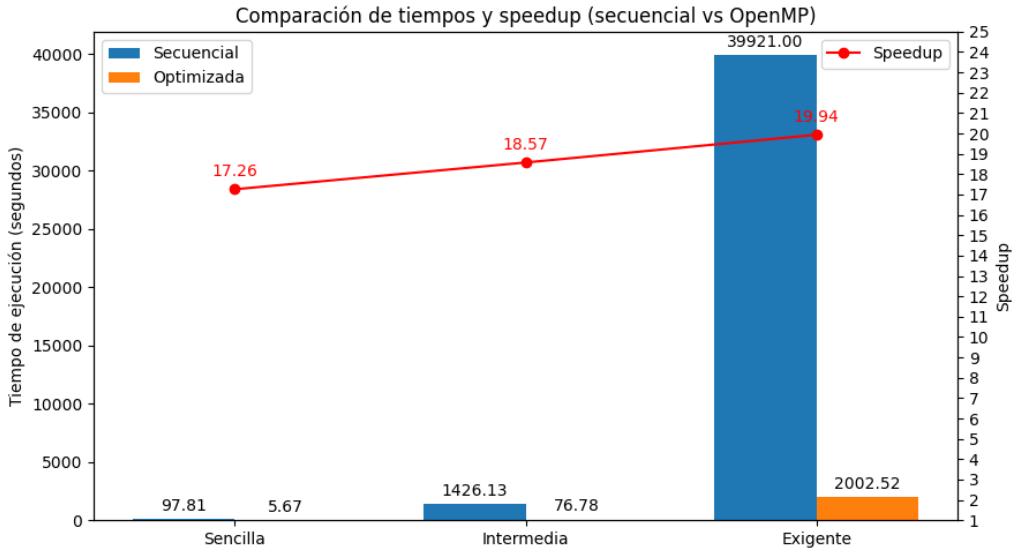


Figura 5.4: Comparación de tiempos y speedup entre versiones secuencial y paralela (OpenMP).

Se observa que el tiempo de ejecución se reduce drásticamente en la versión optimizada en todas las pruebas, especialmente en la prueba *Exigente*, donde el tiempo pasa de más de 39 000 segundos a solo unos 2 000 segundos. Esto refleja un *speedup* de aproximadamente $\times 19{,}94$, lo cual representa una ganancia significativa de rendimiento. Cuando hablamos de acelerar un programa usando OpenMP, la idea principal es dividir el trabajo entre los núcleos del procesador para hacerlo más rápido. En este caso, contamos con 16 núcleos y hemos logrado acelerar el programa entre 15 y 20 veces en comparación con la versión que solo usaba un núcleo.

Técnicamente, si todo el programa pudiera ejecutarse en paralelo sin ningún problema, el máximo aceleramiento teórico según la **Ley de Amdahl** que podríamos esperar con 16 núcleos es 16 veces, porque cada núcleo hace una parte del trabajo al mismo tiempo. Sin embargo, hemos visto que en algunos casos superamos ese límite de 16, alcanzando hasta 20 veces más rápido.

Esto puede parecer extraño, pero tiene explicación: además de repartir el trabajo, también hemos mejorado el código para que sea más eficiente. Por ejemplo, reducimos el tiempo perdido en llamadas a funciones y en decisiones condicionales del programa, y aprovechamos mejor las memorias rápidas internas de cada núcleo (llamadas a cachés). Al tener menos datos por núcleo y trabajar más localmente, los accesos a la memoria son más rápidos y hay menos esperas.

En la Figura 5.5, se presenta la misma información, pero utilizando una escala logarítmica en el eje y izquierdo. Esta visualización permite apreciar con mayor claridad las diferencias relativas entre las pruebas menos costosas (*Sencilla* e *Intermedia*), que en una escala lineal quedan visualmente minimizadas frente al caso *Exigente*.

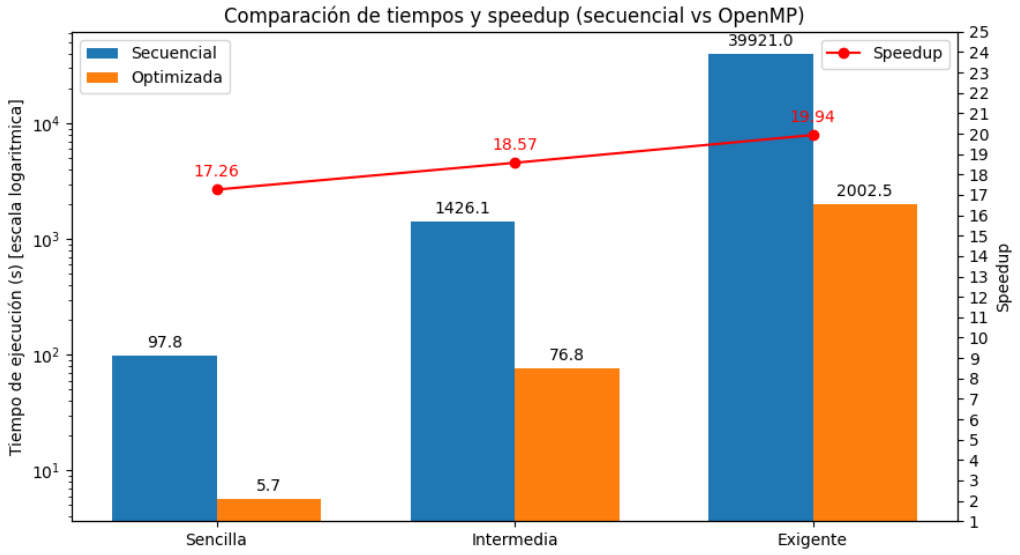


Figura 5.5: Comparación de tiempos y speedup en escala logarítmica.

La representación logarítmica resalta cómo incluso en pruebas con menor carga computacional, el uso de paralelismo sigue produciendo mejoras notables en el tiempo de

ejecución, logrando *speedups* de ×17,26 y ×18,57 para las pruebas *Sencilla* e *Intermedia*, respectivamente.

El incremento progresivo del *speedup* conforme se incrementa la resolución y calidad de la imagen se debe principalmente a dos factores clave:

1. **Mayor carga de trabajo por hilo:** A medida que aumenta la cantidad de píxeles a renderizar y el número de muestras por píxel (por ejemplo, en la prueba *Exigente*), cada hilo tiene más trabajo útil que realizar. Esto permite aprovechar mejor los recursos del sistema y reduce la sobrecarga relativa de gestionar múltiples hilos (sincronización, planificación, etc.).
2. **Mejor amortización de la sobrecarga paralela:** En tareas pequeñas (como la prueba *Sencilla*), la sobrecarga asociada al paralelismo (como el coste de crear/gestionar hilos o balancear carga) representa un porcentaje más alto del tiempo total. En cambio, en tareas intensivas, esta sobrecarga se diluye frente al tiempo de cómputo, permitiendo alcanzar un mayor rendimiento paralelo efectivo.

En resumen, cuanto más compleja y costosa es la imagen a renderizar, más provechoso resulta el uso de paralelismo con OpenMP, debido a que se incrementa el paralelismo útil y se minimiza el impacto relativo de las limitaciones inherentes a la paralelización.

5.2. Paralelización mediante CUDA

5.2.1. Fundamentos prácticos de CUDA en el proyecto

El algoritmo de ray-tracing es inherentemente paralelo, ya que cada rayo puede calcularse de forma independiente del resto. Esto hace que su implementación se adapte naturalmente al modelo de programación SIMD (Single Instruction, Multiple Data) que ofrece CUDA. En esta arquitectura, se pueden lanzar miles de hilos en paralelo, cada uno encargado de calcular una muestra de un píxel. Esta estrategia permite que los tiempos de renderizado, que en CPU pueden tardar horas, se reduzcan a minutos o incluso segundos dependiendo del hardware y la configuración. Para que se entienda mejor; imagina que queremos pintar un mural gigante. Si solo una persona pinta cada punto, tardaría mucho tiempo. Pero si muchas personas, miles, o cientos de miles de ellas, pintan diferentes puntos al mismo tiempo, el mural se termina mucho más rápido. Así funciona CUDA: miles de "personas"(hilos) pintan la imagen simultáneamente.Y no solo la pintan, sino que cada "persona"(hilo) también cálcula cual debe ser el color de su punto.

Por lo tanto, el uso de CUDA en este proyecto plantea un escenario de estudio interesante desde el punto de vista de la computación paralela:

- **Aprovechamiento del paralelismo masivo:** Un solo frame puede requerir el lanzamiento de millones de rayos. CUDA permite explotar el paralelismo a nivel de hilo para computar todos estos caminos de luz simultáneamente.
- **Reducción de cuellos de botella:** Las operaciones pesadas como intersecciones geométricas, reflexiones y cálculos de iluminación se realizan en la GPU, reduciendo la transferencia de datos entre CPU y GPU.
- **Flexibilidad y control:** CUDA proporciona acceso de bajo nivel al hardware,

permitiendo optimizaciones precisas como el uso de memoria compartida, constante, y control de sincronización entre hilos.

5.2.2. Generación de números aleatorios

El *path-tracing* requiere un uso intensivo de números aleatorios para muestrear direcciones de rayos, puntos de intersección, rebotes, y otras operaciones estocásticas esenciales para simular iluminación global. Cada hilo en GPU debe generar secuencias pseudoaleatorias de forma independiente, lo que plantea desafíos tanto de rendimiento como de calidad estadística.

La primera solución adoptada fue utilizar la biblioteca oficial de CUDA, `cuRAND`, [15] específicamente las funciones `curand_init()` y `curand_uniform()`. Esta aproximación garantiza un generador robusto y estadísticamente sólido para cada hilo. Sin embargo, al aplicarla en un contexto donde se lanzan millones de hilos (un hilo por cada muestra de cada píxel), surgieron problemas considerables de rendimiento.

Inicializar un estado `curandState` por hilo implicaba una fase previa costosa que añadía overhead innecesario. Además, cada una de estas estructuras ocupaba una cantidad significativa de memoria global, lo que reducía el espacio disponible para otros datos críticos como la escena o los acumuladores, lo que hacía que se disparasen los accesos a memoria global. Durante la ejecución, cada hilo debía acceder a su estado en esta memoria, lo cual añadía dos penalizaciones extra; mayor latencia de acceso y mayor número de fallos de caché debido a que cada hilo usaba una estructura de estado diferente.

Para mitigar estos problemas, se reemplazó el uso de `cuRAND` por un generador de números pseudoaleatorios ligero implementado directamente dentro del kernel. Esta solución utiliza variaciones simples del algoritmo *Xorshift*, lo que permite obtener valores aleatorios directamente desde registros, sin necesidad de inicialización previa ni almacenamiento en memoria global.

A modo resumen, usar `cuRAND` es como pedirle a miles de personas que usen un complicado programa para sacar números al azar, pero antes tienen que configurarlo bien, lo que consume tiempo. El método *Xorshift* es como darles una regla muy simple para generar números al azar rápidamente, aunque con menos precisión.

¿Qué hace este código? Básicamente, toma un número y lo transforma varias veces para generar otro número que parece aleatorio. Esto se hace dentro de cada hilo sin esperar a otros.

```
__device__ unsigned int xor_shift(unsigned int* state) {
    unsigned int x = *state;
    x ^= x << 13;
    x ^= x >> 17;
    x ^= x << 5;
    *state = x;
    return x;
}

__device__ float random_float(unsigned int* state) {
    return (float)xor_shift(state) / UINT_MAX;
}
```

```
13
14 __device__ float random_float_range(float min, float max, unsigned
       int* state) {
15     return min + (max - min) * random_float(state);
```

Cada hilo genera su propia semilla a partir de un identificador global que combina el índice del píxel con el número de muestra, sumado a una constante global (`master_seed`) que asegura variabilidad entre ejecuciones. Este esquema permite que cada hilo produzca secuencias independientes sin necesidad de sincronización o acceso compartido.

La calidad estadística de este método es suficiente para aplicaciones visuales como el trazado de rayos, donde los defectos menores en la distribución pueden enmascararse gracias al efecto de promediar múltiples muestras por píxel. En la práctica, esta solución permitió reducir significativamente el tiempo de ejecución sin introducir artefactos visibles en la imagen renderizada.

Aunque este generador manual no alcanza la robustez de alternativas como `Philox` o `cuRAND`, su rendimiento lo hace especialmente atractivo en contextos donde la calidad visual es más importante que la precisión estadística absoluta.

5.2.3. Inicialización y transferencia de datos a la GPU

Antes de ejecutar ningún kernel, es esencial transferir a la memoria de la GPU toda la información necesaria para el renderizado. Esto es como preparar todos los materiales en una mesa antes de comenzar a construir algo. Aquí, reservamos espacio en la GPU y copiamos la información desde la CPU para que la GPU la tenga a mano. El siguiente fragmento de código realiza la asignación de memoria en la GPU y la copia de datos desde la CPU:

```
1  uint8_t* device_framebuffer;
2  float* device_accum_buffer;
3  Object* device_objects;
4  Camera* device_camera;
5
6  RenderOptions render_options = {inv_gamma, options->width, options
       ->height, options->samples, master_seed, total_n_pixels,
       n_objects, 0};
7
8  CUDA_CHECK(cudaMalloc((void**)&device_framebuffer, fbuffer_len *
       sizeof(uint8_t)));
9  CUDA_CHECK(cudaMalloc((void**)&device_accum_buffer, fbuffer_len *
       sizeof(float)));
10 CUDA_CHECK(cudaMalloc((void**)&device_objects, n_objects * sizeof(
       Object)));
11 CUDA_CHECK(cudaMalloc((void**)&device_camera, sizeof(Camera)));
12
13 CUDA_CHECK(cudaMemset(device_framebuffer, 0, fbuffer_len * sizeof(
       uint8_t)));
14 CUDA_CHECK(cudaMemset(device_accum_buffer, 0, fbuffer_len * sizeof(
       float)));
15 CUDA_CHECK(cudaMemcpy(device_objects, objects, n_objects * sizeof(
       Object), cudaMemcpyHostToDevice));
```

```
16 CUDA_CHECK(cudaMemcpy(device_camera, camera, sizeof(Camera),
       cudaMemcpyHostToDevice));
17 CUDA_CHECK(cudaMemcpyToSymbol(d_opts, &render_options, sizeof(
       RenderOptions)));
```

Primero se declara memoria en la GPU para los distintos elementos utilizados durante el renderizado. Esto incluye el framebuffer final (`device_framebuffer`), el búfer intermedio de acumulación (`device_accum_buffer`), los objetos de la escena (`device_objects`) y la cámara virtual desde la que se emiten los rayos (`device_camera`). Cada uno se reserva mediante `cudaMalloc`, lo que garantiza su ubicación en el espacio de direcciones de la GPU.

Una vez asignada la memoria, se inicializa a cero tanto el framebuffer como el búfer de acumulación utilizando `cudaMemset`. Esto previene que queden valores residuales que podrían afectar el resultado del renderizado.

A continuación, se copian los datos de la escena desde el host hacia el dispositivo. En concreto, los objetos de la escena se transfieren con `cudaMemcpy`, al igual que la cámara. Estas estructuras se utilizan dentro del kernel para calcular intersecciones y generar rayos correctamente orientados.

Finalmente, se define un objeto de tipo `RenderOptions`, que agrupa todos los parámetros globales del renderizado: corrección gamma, resolución, número de muestras, semilla de aleatoriedad, número total de píxeles y número de objetos. Esta estructura se transfiere al dispositivo mediante `cudaMemcpyToSymbol`, que la copia a una variable global alojada en *constant memory* o memoria constante (`d_opts`) accesible desde cualquier kernel sin necesidad de ser pasada como argumento. Esto reduce el uso de memoria global o registros.

Esta fase de preparación es crítica para garantizar que la GPU disponga de todos los recursos necesarios antes de iniciar la ejecución paralela del renderizado.

5.2.4. Diseño de kernels para trazado de rayos

Kernel `render_sample`

El núcleo computacional del proyecto es el kernel `render_sample`, encargado de calcular las contribuciones de luz de cada muestra en cada píxel de forma masivamente paralela. Este kernel es lanzado con una grid bidimensional de hilos donde la dimensión `x` se asocia a los píxeles de la imagen, y la dimensión `y` a las muestras por píxel. De este modo, cada hilo computa una muestra de un píxel específico, permitiendo una paralelización altamente granular.

```
1 __global__ void render_sample(Camera* __restrict__ camera, Object*
      __restrict__ objects, float* __restrict__ accum_buffer) {
2     extern __shared__ char shared_mem[];
3
4     // Layout shared memory: first objects, then accumulation
          buffer
5     Object* shared_scene = (Object*)shared_mem;
6     float* shared_accum = (float*)(shared_mem + d_opts.n_objects *
          sizeof(Object));
7
```

```
8       int pixel_id = blockIdx.x * blockDim.x + threadIdx.x;
9       int sample_id = blockIdx.y * blockDim.y + threadIdx.y;
10
11      if (pixel_id >= d_opts.width * d_opts.height || sample_id >=
            d_opts.samples) return;
12
13      int tid = threadIdx.x + threadIdx.y * blockDim.x;
14      int nthreads = blockDim.x * blockDim.y;
15
16      // Load scene into shared memory
17      for (int i = tid; i < d_opts.n_objects; i += nthreads) {
18          shared_scene[i] = objects[i];
19      }
20      __syncthreads();
21
22      // Initialize shared accumulation buffer
23      if (threadIdx.y == 0) {
24          shared_accum[threadIdx.x * 3 + 0] = 0.0f;
25          shared_accum[threadIdx.x * 3 + 1] = 0.0f;
26          shared_accum[threadIdx.x * 3 + 2] = 0.0f;
27      }
28      __syncthreads();
29
30      // Ray generation
31      int x = pixel_id % d_opts.width;
32      int y = pixel_id / d_opts.width;
33      int global_id = sample_id * d_opts.width * d_opts.height +
            pixel_id;
34      uint state = d_opts.master_seed + global_id;
35
36      float u = (x + random_float(&state)) / ((float)d_opts.width -
            1.0f);
37      float v = (y + random_float(&state)) / ((float)d_opts.height -
            1.0f);
38
39      Ray ray = get_camera_ray(camera, u, v);
40      vec3 color = trace_path(&ray, shared_scene, d_opts.n_objects,
            d_opts.depth, &state);
41
42      // Accumulate using shared memory
43      atomicAdd(&shared_accum[threadIdx.x * 3 + 0], color.x);
44      atomicAdd(&shared_accum[threadIdx.x * 3 + 1], color.y);
45      atomicAdd(&shared_accum[threadIdx.x * 3 + 2], color.z);
46      __syncthreads();
47
48      // Write result back to global accumulation buffer
49      if (threadIdx.y == 0) {
50          int index = pixel_id * 3;
51          atomicAdd(&accum_buffer[index + 0], shared_accum[threadIdx.
                x * 3 + 0]);
52          atomicAdd(&accum_buffer[index + 1], shared_accum[threadIdx.
                x * 3 + 1]);
```

```
53          atomicAdd(&accum_buffer[index + 2], shared_accum[threadIdx.
                x * 3 + 2]);
54      }
55  }
```

Este kernel ha sido cuidadosamente diseñado para aprovechar el paralelismo masivo y la jerarquía de memoria de CUDA. Se destacan las siguientes decisiones de diseño:

Cada hilo en este kernel se encarga de calcular la contribución de una única muestra de un píxel específico a la imagen final. Para ello, se utilizan varias optimizaciones y estructuras clave:

- **Asignación de trabajo a hilos:**
  - La coordenada `blockIdx.x` y `threadIdx.x` determinan el índice del píxel (`pixel_id`).
  - La coordenada `blockIdx.y` y `threadIdx.y` determinan el índice de la muestra (`sample_id`) para ese píxel.
  - Esto permite que múltiples muestras por píxel se computen en paralelo, mejorando la eficiencia del renderizado.
- **Uso de memoria compartida:**
  - Se reserva memoria compartida (`__shared__`) para almacenar temporalmente la escena (`shared_scene`) y un pequeño buffer de acumulación (`shared_accum`) por bloque.
  - La escena completa se copia desde memoria global a memoria compartida por todos los hilos del bloque en paralelo, lo que mejora el rendimiento al reducir accesos a memoria lenta.
  - El buffer de acumulación también reside en memoria compartida para permitir operaciones rápidas de suma parcial antes de escribir al buffer global.
- **Generación del rayo:**
  - A partir de la posición del píxel y de la muestra, se genera un rayo utilizando coordenadas normalizadas (`u`, `v`) y ruido aleatorio controlado por una semilla única por hilo.
  - Este rayo se lanza a través de la función `trace_path`, que calcula el color resultante tras simular el camino de la luz en la escena.
- **Acumulación parcial y sincronización:**
  - Cada hilo acumula su resultado parcial (color calculado) usando `atomicAdd` sobre el buffer compartido del bloque (`shared_accum`), para evitar condiciones de carrera entre hilos que escriben a la misma posición.
  - Después de sincronizar, los hilos con `threadIdx.y == 0` (uno por píxel en el bloque) suman los valores del buffer compartido al `accum_buffer` global, también usando operaciones atómicas para evitar conflictos de escritura.

Es importante mencionar que la naturaleza recursiva de la función `trace_path` se elimina en la implementación con CUDA. En su lugar, se emplea una versión iterativa mediante un bucle *while*, que mantiene la lógica de profundidad del algoritmo original, pero evita el sobrecoste asociado a las llamadas recursivas. Aunque CUDA admite recursión desde versiones recientes, su uso no es eficiente debido a las limitaciones de memoria de pila por hilo y la dificultad de mantener un alto nivel de ocupación. Además, si la profundidad de la recursión es elevada o la función hace uso intensivo de registros, puede producirse *spilling*, es decir, un desbordamiento de registros o pila hacia memoria local. Dado que esta memoria local reside en memoria global, su latencia es significativamente mayor, lo que puede perjudicar seriamente el rendimiento del programa. Reescribir la función de forma iterativa mejora el control sobre la memoria utilizada y permite una ejecución más eficiente en paralelo, reduciendo así la latencia global del algoritmo.

Kernel `finalize_image`

Una vez acumuladas todas las muestras, el kernel `finalize_image` se encarga de calcular el color promedio de cada píxel y aplicar la corrección gamma, transformando los valores para formar la imagen final.

```
__global__ void finalize_image(float* __restrict__ accum_buffer,
    uint8_t* __restrict__ framebuffer) {
    int pixel_id = blockIdx.x * blockDim.x + threadIdx.x;
    if (pixel_id >= d_opts.width * d_opts.height) return;

    int index = pixel_id * 3;
    float inv_samples = 1.0f / d_opts.samples;

    float r = powf(accum_buffer[index + 0] * inv_samples, d_opts.
        inv_gamma);
    float g = powf(accum_buffer[index + 1] * inv_samples, d_opts.
        inv_gamma);
    float b = powf(accum_buffer[index + 2] * inv_samples, d_opts.
        inv_gamma);

    framebuffer[index + 0] = (uint8_t)(255.0f * CLAMP_FLOAT(r));
    framebuffer[index + 1] = (uint8_t)(255.0f * CLAMP_FLOAT(g));
    framebuffer[index + 2] = (uint8_t)(255.0f * CLAMP_FLOAT(b));
}
```

Este kernel es ligero y está altamente paralelizado: cada hilo procesa un píxel completo. El uso de funciones como `powf()` para corrección gamma y el escalado al rango [0,255] permiten producir imágenes visualmente correctas. Se destaca que no es necesaria sincronización entre hilos debido a que cada píxel es procesado de manera independiente.

Consideraciones clave en su diseño:

- **Alta ocupación de recursos:** Su simplicidad lo hace ideal para alcanzar una ocupación del 100 % en la mayoría de GPUs.
- **Evita atomics:** Como no hay escritura concurrente en el framebuffer, se eliminan operaciones atómicas, reduciendo el overhead.

- **Compatibilidad con diferentes resoluciones:** El kernel es escalable a cualquier tamaño de imagen gracias a la verificación de límites.

Configuración de grids y blocks

Una parte fundamental para lograr un rendimiento óptimo en CUDA es la correcta elección de la configuración de grids y bloques. En este proyecto, se adoptó una organización bidimensional de hilos mediante bloques de tamaño fijo `dim3(16, 16)`. Esta elección permite un equilibrio razonable entre ocupación, uso de registros y aprovechamiento del multiprocesador.

La grid se configuró de manera que cubra todas las combinaciones de píxeles y muestras por píxel. Considerando que cada hilo es responsable de generar una muestra para un píxel dado, el tamaño de la grid se calcula de la siguiente manera:

```
dim3 blockDim(16, 16);
dim3 gridDim(
  (total_n_pixels + blockDim.x - 1) / blockDim.x,
  (options->samples + blockDim.y - 1) / blockDim.y
);
int shared_mem_size = n_objects * sizeof(Object) + blockDim.x * 3 *
    sizeof(float);
render_sample<<<gridDim, blockDim, shared_mem_size>>>(...);
```

Con esta estructura, la dimensión X de la grid se encarga de distribuir los hilos entre píxeles, mientras que la dimensión Y reparte las muestras por cada uno. Cada hilo tiene un identificador único compuesto por su posición dentro del bloque y la grid, a partir del cual se calcula el píxel y la muestra que le corresponde. Esta organización facilita la generación de rayos, ya que cada muestra es computada de forma completamente paralela y con su propia semilla aleatoria.

Por otro lado, el kernel `finalize_image`, responsable de convertir los acumuladores en valores RGB de 8 bits, utiliza una configuración unidimensional de bloques y grid:

```
dim3 finalBlock(256);
dim3 finalGrid((total_n_pixels + finalBlock.x - 1) / finalBlock.x);
finalize_image<<<finalGrid, finalBlock>>>(...);
```

Dado que esta etapa consiste en una operación sencilla por píxel (normalización y corrección gamma), no requiere una configuración bidimensional ni uso de memoria compartida, y puede ejecutarse con alta eficiencia en paralelo.

Esta configuración ha sido afinada para balancear la ocupación de los SMs, la cantidad de registros por hilo y la utilización de memoria compartida, logrando un rendimiento sostenido y una utilización efectiva del hardware disponible.

5.2.5. Gestión de memoria

Memoria compartida y memoria constante

Para reducir accesos costosos a memoria global, se copiaron los objetos de la escena a **memoria compartida** [16] al inicio del kernel. Dado que estos datos son leídos múltiples veces durante el cálculo de intersecciones, almacenarlos en `shared memory` mejoró

notablemente la eficiencia. En el lanzamiento del kernel principal `render_sample`, se debe asignar memoria compartida dinámica:

```
int shared_mem_size = n_objects * sizeof(Object) + blockDim.x * 3 *
    sizeof(float);
render_sample<<<gridDim, blockDim, shared_mem_size>>>(...);
```

Esta memoria se emplea para cargar la escena una sola vez por bloque, y también para acumular de manera cooperativa los colores generados por cada hilo.

Además, se utilizó `__restrict__` en los punteros de entrada para permitir al compilador hacer optimizaciones de aliasing. También se aprovecharon variables `__constant__` para opciones de renderizado que no cambian a lo largo de la ejecución.

A continuación se puede observar las ventajas del uso de memoria compartida y constante en vez de global [17]:

Cuadro 5.7: Tipos de memoria en CUDA y penalización de rendimiento

Variable	**Penalización de rendimiento**
`int localVar;` (registro)	1x
`int localArray[10];` (local)	100x
`__shared__ int sharedVar;` (compartida)	1x
`__device__ int globalVar;` (global)	100x
`__constant int constantVar;` (constante)	1x

Como se puede ver en la tabla, la memoria global es como si los cocineros de un restaurante tuvieran que ir a una bodega lejana cada vez que necesitan un ingrediente, lo que los hace tardar mucho. La memoria compartida sería como tener una mesa cercana con los ingredientes listos para cada grupo de cocineros que están preparando el mismo plato, lo que agiliza mucho el proceso. La memoria constante es comparable a un cartel en la pared con las reglas del restaurante, que todos los cocineros pueden leer rápidamente. Finalmente, los registros serían como los utensilios que cada chef tiene siempre a mano sobre su estación de trabajo.

Operaciones de coma flotante: `double` versus `float`

Una de las decisiones técnicas más determinantes para el rendimiento del ray-tracer en GPU fue el cambio de precisión numérica de `double` (coma flotante en doble precisión, 64 bits) a `float` (precisión simple, 32 bits) [18].

Inicialmente, todo el cálculo de trayectorias, intersecciones y acumulación de color se realizaba utilizando `double`, motivado por la estabilidad numérica que ofrece esta representación. Sin embargo, este enfoque mostró limitaciones claras al ejecutarse en la GPU: los tiempos de ejecución eran significativamente más altos que los esperados, similares a versiones CPU paralelizadas.

Este problema se debe a la arquitectura de las GPU modernas. Estas tarjetas están optimizadas masivamente para cálculos en `float`, que son ejecutados con un rendimiento muy superior al de `double`. En GPUs de gama media, la cantidad de ALUs dedicadas a

doble precisión es muy reducida, ya que la mayoría del hardware está orientado a operaciones de precisión simple, como las utilizadas en videojuegos, gráficos y deep learning.

Además del soporte hardware, la precisión simple aporta las siguientes ventajas prácticas para CUDA:

- **Menor uso de registros por hilo**: las variables de tipo `float` ocupan la mitad del espacio de las `double`, lo que reduce la presión sobre los registros disponibles por SM. Esto permite lanzar más hilos concurrentes y mejorar la *ocupación*.
- **Menor uso de ancho de banda de memoria**: al mover datos más pequeños, se reduce el tráfico global de memoria y se mejora la eficiencia de cachés.
- **Mayor cantidad de hilos activos por SM**: dado que los recursos por hilo disminuyen, es posible mantener más hilos activos simultáneamente en cada multiprocesador, lo que mejora el aprovechamiento del hardware.

Tras migrar todos los cálculos a precisión simple, el rendimiento del kernel principal experimentó una aceleración notable: se consiguió una aceleración muy superior utilizando `float` en lugar de `double`. Esta mejora no solo es atribuible al cambio de precisión, sino también a una mayor ocupación efectiva de los SM de la GPU.

Antes de confirmar esta optimización, se validó cuidadosamente que la conversión no introdujera errores numéricos visibles. Se evaluaron casos con múltiples rebotes de rayos, reflexiones, refracciones y trayectorias largas para detectar posibles artefactos derivados de la pérdida de precisión. Visualmente, no se identificaron errores perceptibles, y se mantuvo una calidad de imagen satisfactoria como podrá verse en la sección de resultados.

El uso de `float` es, por tanto, plenamente adecuado en contextos de renderizado gráfico como el de este proyecto. Solo en casos de simulaciones físicas de alta fidelidad —como mecánica de fluidos o cálculos científicos— sería necesario recurrir a la doble precisión.

El código completo de la versión modificada para hacer uso de CUDA puede encontrarse en Ray-Tracer CUDA (https://github.com/davidmorilla/raytracer/tree/main/CUDA).

5.2.6. Evaluación de rendimiento con CUDA

Resultados según tiempo de ejecución

Cuadro 5.8: Resumen de tiempos de ejecución de la prueba sencilla.

Tiempo y configuración			
Elapsed Time:	0.297 s	Total Thread Count:	6
CPU Time:	0.290 s	Paused Time:	0 s

Cuadro 5.9: Resumen de tiempos de ejecución de la prueba intermedia.

Tiempo y configuración			
Elapsed Time:	2.908 s	Total Thread Count:	5
CPU Time:	1.024 s	Paused Time:	0 s

Cuadro 5.10: Resumen de tiempos de ejecución de la prueba exigente.

Tiempo y configuración			
Elapsed Time:	49.431 s	Total Thread Count:	5
CPU Time:	3.911 s	Paused Time:	0 s

Resultados según consumo de memoria

Cuadro 5.11: Consumo de memoria en la prueba simple

Memoria total utilizada:	4.30 MiB

Cuadro 5.12: Consumo de memoria total en la prueba intermedia

Memoria total utilizada:	13.86 MiB

Cuadro 5.13: Consumo de memoria total en la prueba exigente

Memoria total utilizada:	35.59 MiB

Imágenes generadas

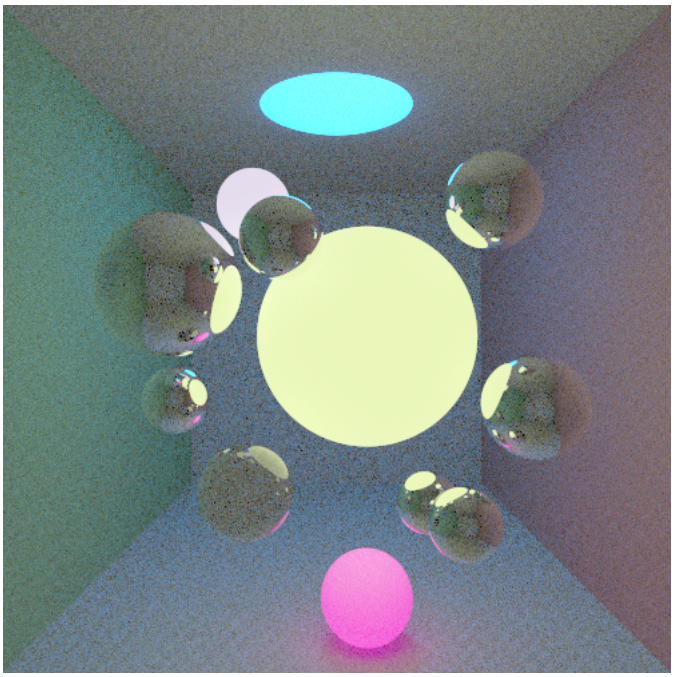


Figura 5.6: Imagen resultado de la prueba sencilla usando CUDA.

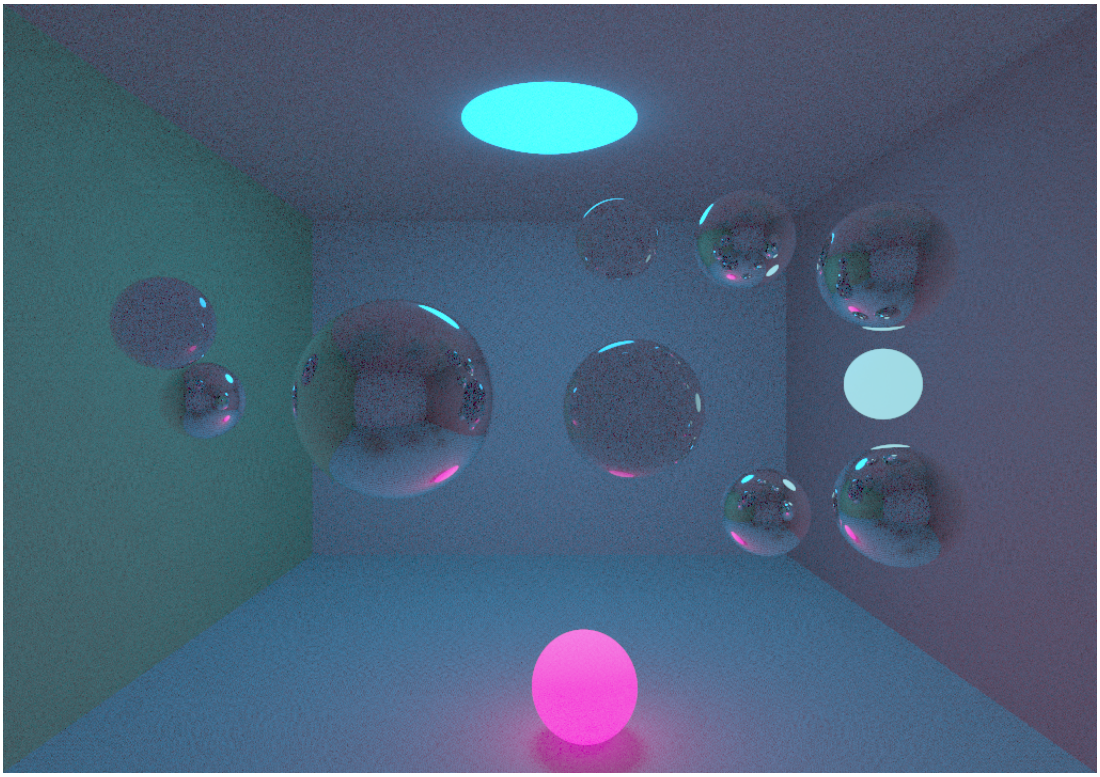


Figura 5.7: Imagen resultado de la prueba intermedia usando CUDA.

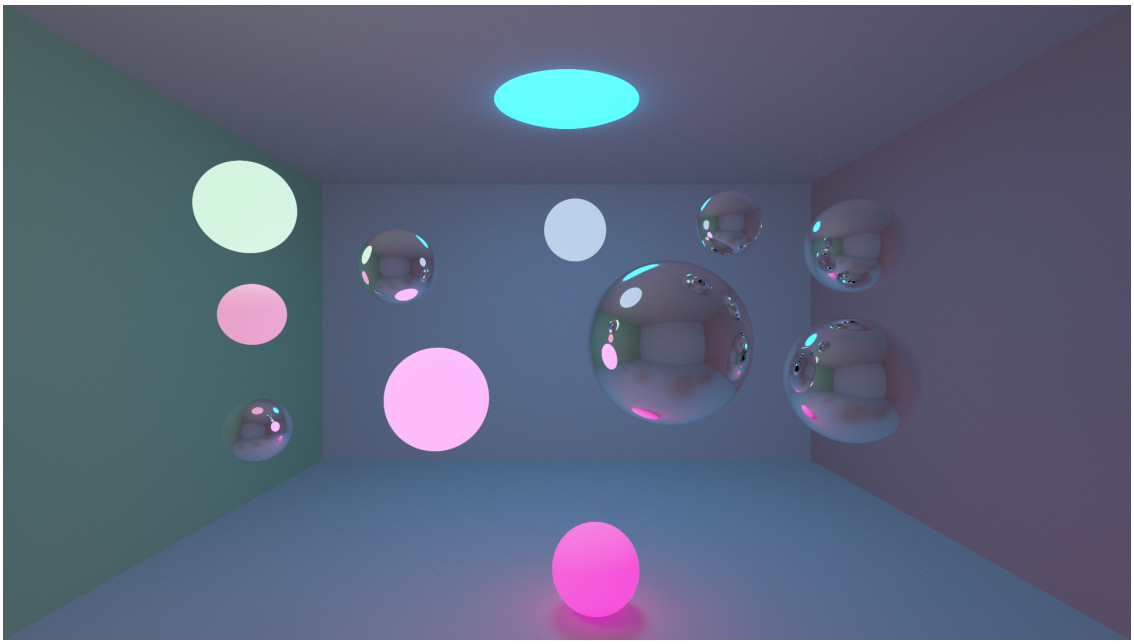


Figura 5.8: Imagen resultado de la prueba exigente usando CUDA.

Comparación gráfica de tiempos de ejecución y speedup

En la Figura 5.9, se presenta una comparación visual entre los tiempos de ejecución obtenidos en tres pruebas (*Sencilla*, *Intermedia* y *Exigente*) tanto en su versión secuencial como en su versión optimizada con CUDA. Además, se muestra el *speedup* logrado en cada prueba mediante una línea roja sobrepuesta, referenciada al eje vertical derecho.

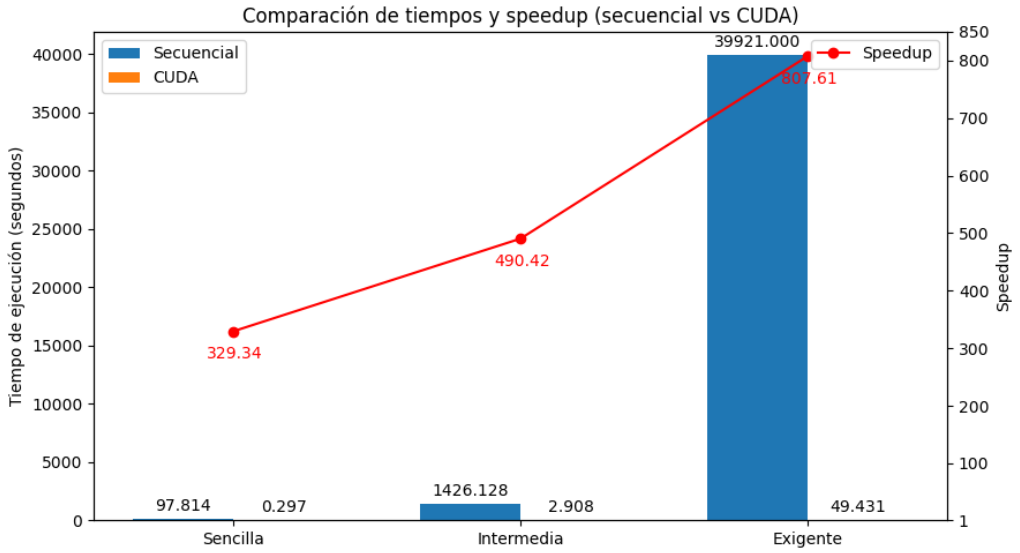


Figura 5.9: Comparación de tiempos y speedup entre versiones secuencial y paralela (CUDA).

Se observa que el tiempo de ejecución se reduce drásticamente en la versión optimizada con CUDA en todas las pruebas, alcanzando *speedups* extremadamente altos respecto a la versión secuencial: aproximadamente $\times 329{,}34$, $\times 490{,}42$ y $\times 807{,}61$ para las pruebas *Sencilla*, *Intermedia* y *Exigente*, respectivamente. Estos resultados reflejan ganancias de rendimiento muy significativas.

En la Figura 5.10, se presenta la misma información, pero utilizando una escala logarítmica en el eje y izquierdo. Esta visualización permite apreciar con mayor claridad las diferencias relativas entre las pruebas menos costosas (*Sencilla* e *Intermedia*), que en una escala lineal quedarían visualmente minimizadas frente al caso *Exigente*.

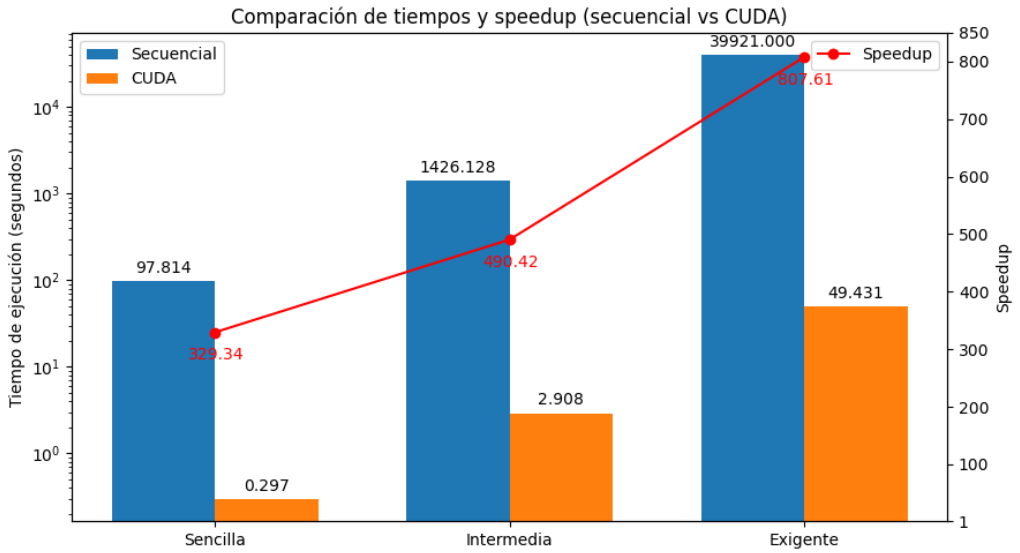


Figura 5.10: Comparación de tiempos y speedup en escala logarítmica.

El incremento progresivo del *speedup* conforme se incrementa la resolución y calidad de la imagen se debe a la gran capacidad de paralelismo de la arquitectura GPU, que permite manejar miles de hilos simultáneamente y aprovechar mejor la carga de trabajo a medida que esta crece. En tareas intensivas, la sobrecarga del paralelismo se amortiza eficazmente, maximizando el rendimiento.

Además, CUDA optimiza el acceso a la memoria y oculta latencias mediante la ejecución concurrente de múltiples hilos, lo que contribuye a alcanzar *speedups* muy elevados. La combinación de estos factores hace que el uso de CUDA sea especialmente ventajoso en problemas con alta carga computacional, como el renderizado en las pruebas presentadas.

En resumen, cuanto más compleja y costosa es la imagen a renderizar, mayor es el beneficio de utilizar CUDA, logrando mejoras sustanciales en tiempo de ejecución respecto a la versión secuencial.

5.3. Definición del entorno de pruebas

El entorno de desarrollo configurado para el proyecto incluyó los siguientes elementos:

- **Sistema operativo:** Ubuntu Linux 22.04.5 LTS 64 bits, por su compatibilidad con herramientas de desarrollo científico y soporte nativo para CUDA.
- **CPU:** 13th Gen Intel® Core™ i7-13620H × 16
- **GPU:** NVIDIA GeForce RTX™ 3050
- **Compiladores/Intérpretes:** `python3` para Python, `gcc` para C, `nvcc` para CUDA.
- **Editor de código:** VS Code con extensiones para C/C++, CUDA, Python y depuración.

- **Herramientas de compilación:** `Make`, `CMake` y scripts personalizados para automatizar el proceso de construcción.

- **Gestión de versiones:** Git, para control de versiones.

- **Otros:** Librería OpenMP, CUDA Toolkit.

Este entorno proporcionó la infraestructura necesaria para desarrollar, probar, analizar y comparar las diferentes versiones del sistema.

Los flags de compilación utilizados para cada versión del código fueron:

- **Versión secuencial:** gcc –std=c99 -Wall -g -fno-omit-frame-pointer -Wno-strict-aliasing -Wno-unused-variable -Wno-unused-function -O0

- **Versión OpenMP:** gcc –std=c99 -Wall -O3 -Wno-strict-aliasing -Wno-unused-variable -Wno-unused-function -fopenmp -g -fno-omit-frame-pointer

- **Versión CUDA:** nvcc -std=c++14 -rdc=true -g -O3 -arch=sm_86 -Xcompiler -Wall,-Wno-unused-variable,-Wno-unused-function

5.4. Comparativa: secuencial vs OpenMP vs CUDA

En esta sección se presentan los resultados obtenidos al ejecutar el mismo conjunto de pruebas bajo tres enfoques diferentes de paralelización: ejecución secuencial, paralelización mediante **OpenMP** (multithread sobre CPU) y paralelización mediante **CUDA** (GPU). Las pruebas realizadas fueron de tres niveles de exigencia: *Sencilla*, *Intermedia* y *Exigente*.

5.4.1. Tiempos de ejecución

La Figura 5.11 muestra los tiempos de ejecución medidos para cada una de las pruebas bajo los distintos enfoques en escala logarítmica. Como se puede observar, los códigos paralelizados (OpenMP y CUDA) ofrecen mejoras significativas con respecto a la versión secuencial, especialmente CUDA en los casos más exigentes.

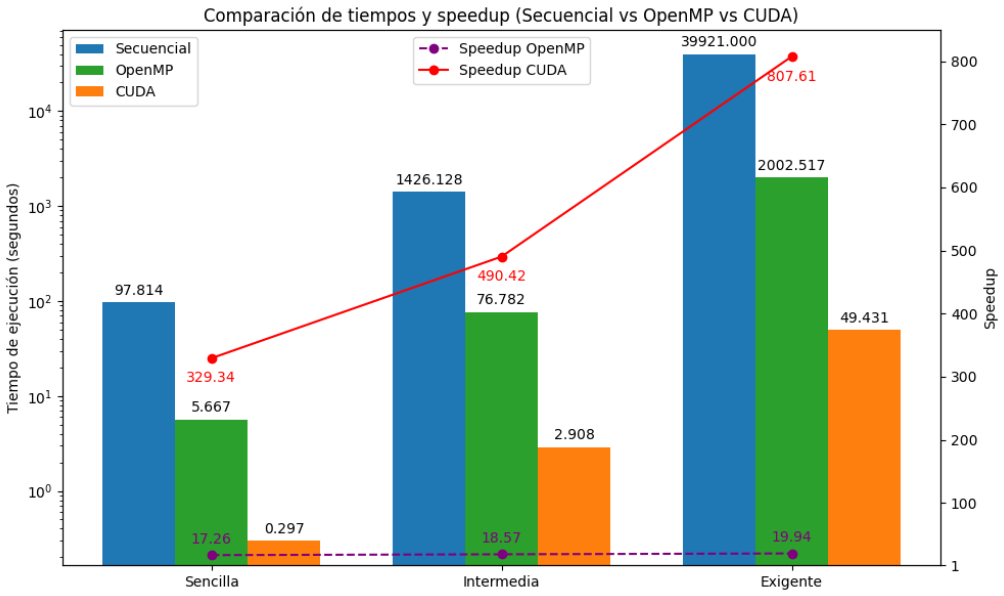


Figura 5.11: Comparación de tiempos de ejecución y speedup: Secuencial vs OpenMP vs CUDA.

5.4.2. Análisis de speedup

Se calcula el **speedup** como la relación entre el tiempo de ejecución secuencial y el tiempo de ejecución paralelizado:

$$\text{Speedup} = \frac{T_{\text{secuencial}}}{T_{\text{paralelo}}}$$

La línea punteada morada en la figura representa el speedup de OpenMP, mientras que la línea roja representa el speedup de CUDA. Los resultados muestran que:

- OpenMP ofrece un speedup moderado en las pruebas sencilla e intermedia, pero su escalabilidad es limitada frente a CUDA.
- CUDA logra un speedup considerablemente mayor, especialmente en el caso exigente, donde se alcanzan mejoras de hasta casi tres órdenes de magnitud.
- El uso de GPU (CUDA) resulta especialmente ventajoso para cargas de trabajo pesadas y altamente paralelizables.

5.4.3. Análisis del uso de memoria

A pesar de los significativos incrementos en rendimiento obtenidos mediante la paralelización, el uso de memoria del sistema se mantiene prácticamente inalterado. La figura 5.12 presenta una comparativa entre las distintas versiones del motor, donde se observa que no existen diferencias significativas en cuanto al consumo de memoria. Esto indica que las optimizaciones implementadas no han generado un mayor coste en este recurso, lo cual refuerza la eficiencia general de las soluciones propuestas.

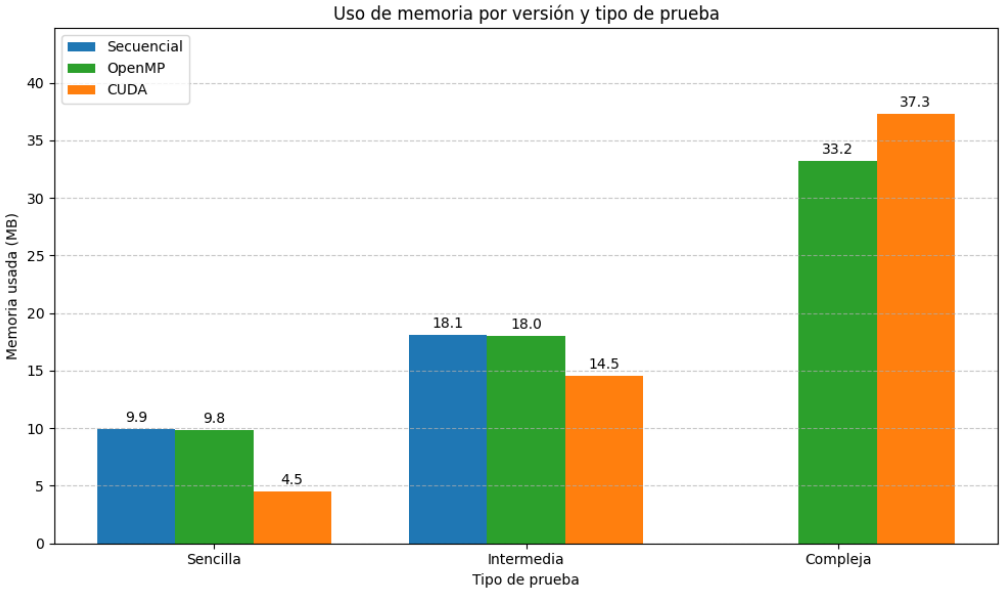


Figura 5.12: Comparativa del uso de memoria entre las diferentes versiones del renderizador.

Capítulo 6

Conclusiones y trabajo futuro

Conclusiones

La comparativa demuestra que, aunque OpenMP representa una mejora fácil de implementar sobre la ejecución secuencial, es la aceleración por GPU mediante CUDA la que proporciona los mejores resultados en términos de rendimiento. No obstante, el beneficio real depende del tipo de tarea, el tamaño del problema y los recursos de hardware disponibles.

6.1. Discusión sobre ventajas, limitaciones, aplicabilidad según contextos (CPU vs GPU) y coste computacional

En este trabajo se ha demostrado que, aunque OpenMP ofrece una forma sencilla y rápida de paralelizar código secuencial para mejorar el rendimiento, la aceleración mediante GPU con CUDA proporciona mejores resultados en términos de velocidad, especialmente en problemas grandes y muy paralelizables. Sin embargo, la mejora real depende del tipo de tarea, el tamaño del problema y el hardware disponible.

OpenMP resulta adecuado para situaciones donde solo se cuenta con CPU o se busca una mejora sin hacer grandes cambios en el código. Por otro lado, CUDA exige un mayor esfuerzo en programación y un conocimiento más profundo de la arquitectura de la GPU, incluyendo la gestión explícita de la memoria y la sincronización entre hilos, pero puede reducir significativamente los tiempos de ejecución cuando la aplicación es adecuada.

En cuanto al uso de recursos, CUDA suele requerir hardware especializado y puede consumir más energía, mientras que OpenMP aprovecha el CPU sin necesidad de invertir en componentes adicionales, siendo una opción práctica en entornos con recursos limitados.

En resumen, la elección entre OpenMP y CUDA depende del tipo de aplicación, las condiciones del entorno y los objetivos específicos del proyecto.

6.2. Dificultades técnicas encontradas

Durante el desarrollo de este trabajo surgieron varias dificultades técnicas que influyeron en los resultados y en el tiempo necesario para completar el proyecto. Una de las principales complicaciones estuvo en la gestión de la memoria en CUDA, especialmente al intentar

que la transferencia de datos entre la CPU y la GPU fuera lo más rápida posible. Esta comunicación suele ser lenta y, en algunos casos, se convirtió en un cuello de botella que limitaba el rendimiento general. Buscar la mejor manera de optimizar estas transferencias implicó probar distintas técnicas y ajustar el código, lo que consumió más tiempo del esperado.

Además, la programación paralela en CUDA trajo sus propios retos. Fue necesario manejar de forma cuidadosa la gestión de los hilos y las sincronizaciones para evitar errores, y esto añadió mucha complejidad al código. Por ejemplo, hubo que implementar operaciones atómicas para prevenir condiciones de carrera que podían causar resultados incorrectos. También se trabajó con buffers intermedios para organizar mejor los datos y evitar conflictos entre los hilos. Todo esto hizo que las refactorizaciones debieran hacerse con mucho cuidado para no introducir nuevos fallos difíciles de detectar.

Otro rompecabezas inicial importante fue la generación de números aleatorios en CUDA. Adaptar este proceso para que funcionara bien en un entorno paralelo no fue sencillo y requirió investigación y pruebas hasta encontrar una solución estable y eficiente que no introdujese artefactos visuales en las imágenes generadas.

Otra dificultad técnica relevante fue refactorizar la función `trace_path`, que originalmente estaba implementada de forma recursiva, para convertirla en una versión iterativa adecuada para CUDA. Esta transformación no solo implicó reescribir la lógica, sino también asegurarse de que fuera eficiente y compatible con la ejecución en paralelo, lo que demandó bastante esfuerzo.

Por último, diseñar kernels que sacaran el máximo provecho a la arquitectura de la GPU requirió varios intentos y ajustes finos, y la depuración de errores relacionados con accesos a memoria fue una tarea laboriosa.

6.3. Objetivos logrados

Tras la finalización del proyecto, se puede concluir que se han alcanzado todos los objetivos iniciales propuestos:

- Se ha realizado un estudio exhaustivo sobre los fundamentos teóricos y prácticos de la programación paralela, con énfasis en los modelos de memoria compartida y en el uso de herramientas como `OpenMP` y `CUDA`.
- Se ha analizado en profundidad la arquitectura y funcionamiento del código base del renderizador, identificando las secciones críticas susceptibles de paralelización.
- Se han aplicado diversas estrategias de paralelización sobre el sistema, tanto a nivel de CPU como de GPU, evaluando su impacto en el rendimiento y la eficiencia computacional.
- Se han medido y analizado los resultados obtenidos mediante experimentos controlados, prestando especial atención a los tiempos de ejecución, escalabilidad y aprovechamiento de recursos.
- Se ha realizado una comparación detallada entre las versiones secuencial, paralelizada con OpenMP y acelerada con CUDA, justificando las decisiones técnicas adoptadas

en cada caso.

- Finalmente, se ha elaborado una memoria técnica completa que documenta las técnicas utilizadas, los resultados obtenidos, las dificultades encontradas durante el desarrollo y las conclusiones alcanzadas.

6.4. Líneas de trabajo futuro

Aunque el proyecto cumplió con la mayoría de sus objetivos iniciales, algunos aspectos quedaron inicialmente fuera del alcance debido a limitaciones de tiempo, complejidad técnica o recursos disponibles.

Uno de los objetivos que no se pudo abordar fue la optimización avanzada de la transferencia de datos entre la CPU y la GPU. Aunque se realizaron algunas mejoras, la exploración de técnicas como la compresión de datos, el uso extensivo de memoria unificada o estrategias más sofisticadas para minimizar este coste quedó pendiente. Estas optimizaciones son complejas y requieren un análisis detallado que excedió el marco temporal del trabajo.

Asimismo, no se implementó ni evaluó la ejecución en entornos con múltiples GPUs, lo que hubiera permitido estudiar la escalabilidad en sistemas más grandes y complejos. La limitación en el acceso a hardware restringió las pruebas a una sola unidad de procesamiento gráfico, dejando la puerta abierta a futuras investigaciones sobre distribuciones de carga y sincronización entre varias GPUs.

Otro objetivo interesante que podría haberse buscado en este proyecto es el del estudio y aplicación de otros frameworks de paralelización, como OpenCL, que podrían haber ampliado la compatibilidad con diferentes marcas de hardware, así como la evaluación en tarjetas gráficas de fabricantes alternativos a NVIDIA. Esto habría aportado una visión más amplia y generalizable de las ventajas y desventajas de cada enfoque.

Finalmente, no se profundizó en el análisis del impacto energético y económico del uso de CPU frente a GPU. Dado que la eficiencia energética es un factor cada vez más relevante en el desarrollo de software de alto rendimiento, este aspecto habría enriquecido en gran medida la evaluación de las estrategias de paralelización.

En resumen, estos objetivos quedaron fuera principalmente por restricciones de tiempo, recursos y la complejidad técnica que implicaban, pero constituyen líneas claras de trabajo para el futuro, que podrían complementar y ampliar los resultados obtenidos en este **Trabajo de Fin de Grado**.

6.5. Análisis de impacto

Este Trabajo de Fin de Grado ha sido una experiencia muy enriquecedora a nivel personal y técnico. Me ha permitido profundizar en temas como la programación paralela, la optimización de algoritmos y el uso de tecnologías como CUDA, que hasta ahora solo conocía de forma teórica y ligeramente práctica gracias a asignaturas del grado como *Computación de Alto Rendimiento* y *Técnicas de Computación Científica*. También he mejorado en aspectos prácticos como el análisis de rendimiento y la detección de cuellos de botella, habilidades clave en el desarrollo de software eficiente.

Además de lo técnico, he aprendido a organizarme mejor, a ser constante y a tomar decisiones por mi cuenta. Afrontar un proyecto de esta magnitud me ha ayudado a ganar confianza y a desarrollar una forma de pensar más crítica y estructurada ante los problemas complejos.

Desde un punto de vista más general, el trabajo demuestra que la mejora del rendimiento computacional puede tener un impacto real en términos de ahorro de tiempos y costes, algo muy relevante en sectores que dependen de cálculos intensivos, como la simulación científica o el renderizado. Optimizar este tipo de procesos no solo mejora lo que ya existe, sino que también permite explorar nuevas aplicaciones y productos, lo que puede traducirse en oportunidades de innovación en el ámbito tecnológico.

En este sentido, el presente trabajo se alinea con diversos Objetivos de Desarrollo Sostenible [19]:

- **ODS 9: Industria, innovación e infraestructura.** La implementación y optimización de tecnologías computacionales avanzadas, como la aceleración mediante CUDA en ray-tracing, promueve la innovación tecnológica y el desarrollo de infraestructuras digitales eficientes, principalmente en la industria de los videojuegos, del cine, y por supuesto de la ciencia. Esto facilita el crecimiento económico sostenible y la competitividad.

- **ODS 12: Producción y consumo responsables.** La mejora en eficiencia computacional reduce el consumo energético asociado a procesos intensivos de cálculo, lo que puede traducirse en un menor impacto ambiental y una gestión más responsable de los recursos tecnológicos.

De este modo, el presente trabajo no solo aporta valor técnico y económico, sino que también se alinea con metas globales para un desarrollo sostenible, demostrando cómo avances en innovación tecnológica pueden tener un impacto positivo en retos ambientales y sociales contemporáneos.

Bibliografía

[1] B. Barney. «Introduction to Parallel Computing». Lawrence Livermore National Laboratory. dirección: https://computing.llnl.gov/tutorials/parallel_comp/.

[2] H. Sutter, «The free lunch is over: A fundamental turn toward concurrency in software», *Dr. Dobb's Journal*, 2005. dirección: http://www.gotw.ca/publications/concurrency-ddj.htm.

[3] T. G. Mattson, B. A. Sanders y B. L. Massingill, *Patterns for Parallel Programming*. Addison-Wesley, 2004.

[4] M. Gil, *Concurrencia*, Accessed: 2025-06-28, Universitat Politècnica de Catalunya, 2025. dirección: https://people.ac.upc.edu/marisa/miso/concurrencia.pdf.

[5] G. R. Díez, *Concurrencia: Condiciones de Carrera*, Accessed: 2025-06-28, Universidad Politécnica de Madrid, 2016. dirección: http://babel.ls.fi.upm.es/~xmc/courses/concurrencia-xmc/material/slides/groman/CC_CondCarrera.pdf.

[6] Wikipedia contributors. «Ley de Amdahl». Accessed: 2025-06-28. dirección: https://es.wikipedia.org/wiki/Ley_de_Amdahl.

[7] OpenMP Architecture Review Board, *OpenMP Application Programming Interface Reference Guide, Version 5.2*, Accessed: 2025-06-28, 2024. dirección: https://www.openmp.org/wp-content/uploads/OpenMPRefGuide-5.2-Web-2024.pdf.

[8] M. Belda et al., «FUME 2.0 – Flexible Universal processor for Modeling Emissions», *Geoscientific Model Development*, vol. 17, págs. 3867-3878, 2024. DOI: 10.5194/gmd-17-3867-2024. dirección: https://gmd.copernicus.org/articles/17/3867/2024/.

[9] N. Benešová, M. Belda, J. Resler, P. Huszár y O. Vlček, *Example Dataset from FUME Model Simulations*, Dataset available since 2023-10-02. Created between 2023-09-12 and 2023-09-12. Collected between 2023-09-01 and 2023-09-10. Language: English, 2023. dirección: https://data.narodni-repozitar.cz/general/datasets/bf6s2-5tq48.

[10] J. Maier. «An Introduction to Ray Tracing In C». Accedido: 28 de junio de 2025, Jakob Maier's Technical Blog. dirección: https://www.jakobmaier.at/posts/raytracing/.

[11] J. Maier. «gue-ni/raytracer.c». Repositorio de código fuente con implementación de ray tracing en C. dirección: https://github.com/gue-ni/raytracer.c.

[12] ScratchPixel, *ScratchPixel*, https://scratchapixel.com/, Accesed: 2025-07-02.

[13] K. Beason, *smallpt*, https://www.kevinbeason.com/smallpt/, Accesed: 2025-07-02.

[14] M. Pharr, W. Jakob y G. Humphreys. «13.7 Russian Roulette and Splitting». Accessed: 2025-06-28. dirección: https://pbr-book.org/3ed-2018/Monte_Carlo_Integration/Russian_Roulette_and_Splitting.

[15] NVIDIA Corporation, *cuRAND Library*, Accessed: 2025-06-28, NVIDIA, 2024. dirección: https://docs.nvidia.com/cuda/curand/index.html.

[16] NVIDIA Technical Blog. «Using Shared Memory in CUDA C/C++». Accessed: 2025-06-28. dirección: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/.

[17] S. S. Cho, *CUDA Memory Model – Lecture 9*, 2011. dirección: https://users.wfu.edu/choss/CUDA/docs/Lecture%209.pdf.

[18] ScienceDirect Topics. «Double Precision in Computer Science». Accessed: 2025-06-28, Elsevier. dirección: https://www.sciencedirect.com/topics/computer-science/double-precision.

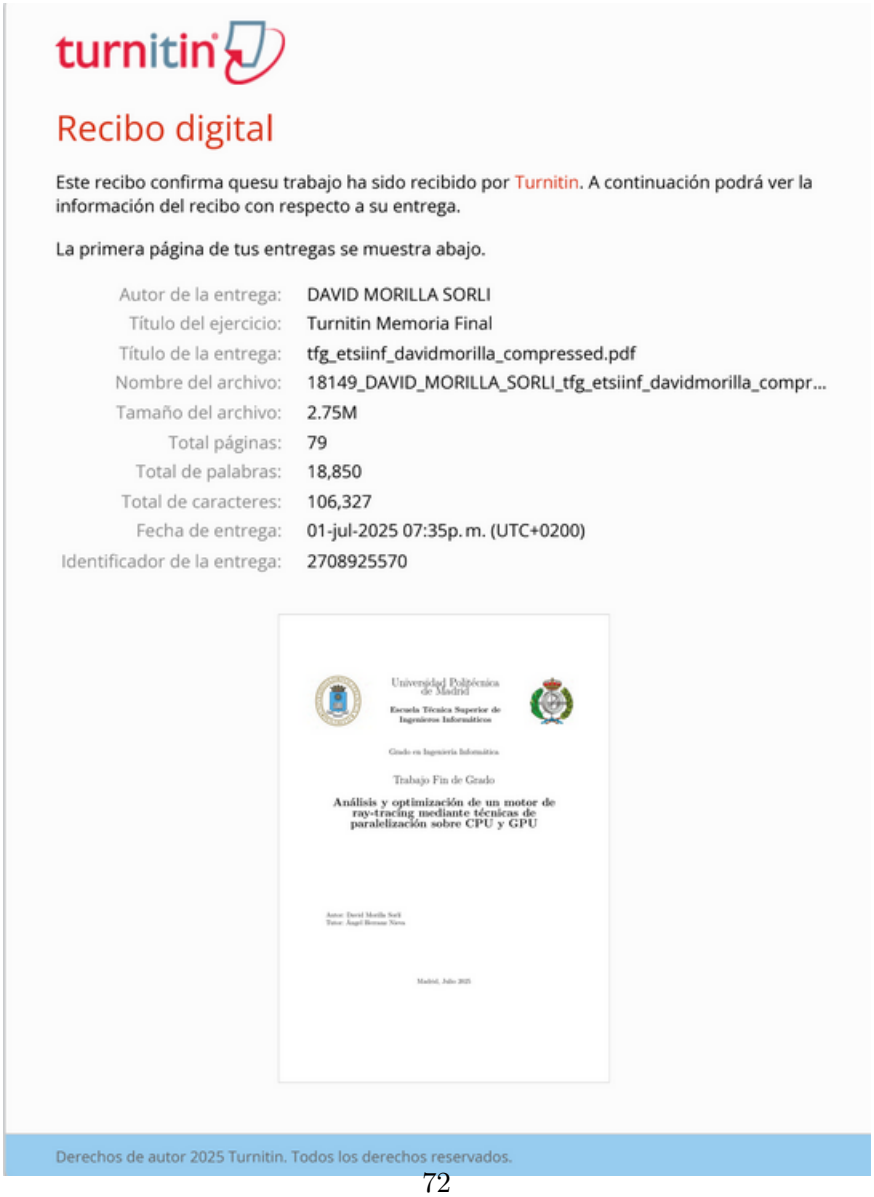
[19] Organización de las Naciones Unidas. «Objetivos de Desarrollo Sostenible». Accessed: 2025-06-28. dirección: https://www.un.org/sustainabledevelopment/es/objetivos-de-desarrollo-sostenible/.

Anexos

Apéndice A

Informe de originalidad

A.1. Informe Turnitin



turnitin

Recibo digital

Este recibo confirma quesu trabajo ha sido recibido por Turnitin. A continuación podrá ver la información del recibo con respecto a su entrega.

La primera página de tus entregas se muestra abajo.

Autor de la entrega:	DAVID MORILLA SORLI
Título del ejercicio:	Turnitin Memoria Final
Título de la entrega:	tfg_etsiinf_davidmorilla_compressed.pdf
Nombre del archivo:	18149_DAVID_MORILLA_SORLI_tfg_etsiinf_davidmorilla_compr...
Tamaño del archivo:	2.75M
Total páginas:	79
Total de palabras:	18,850
Total de caracteres:	106,327
Fecha de entrega:	01-jul-2025 07:35p. m. (UTC+0200)
Identificador de la entrega:	2708925570

A.2. Porcentaje de similitud


turnitin Página 2 of 83 - Descripción general de integridad Identificador de la entrega trn:oid:::1:3288600158

4% Similitud general

El total combinado de todas las coincidencias, incluidas las fuentes superpuestas, para ca...

Filtrado desde el informe

- Bibliografía
- Texto citado

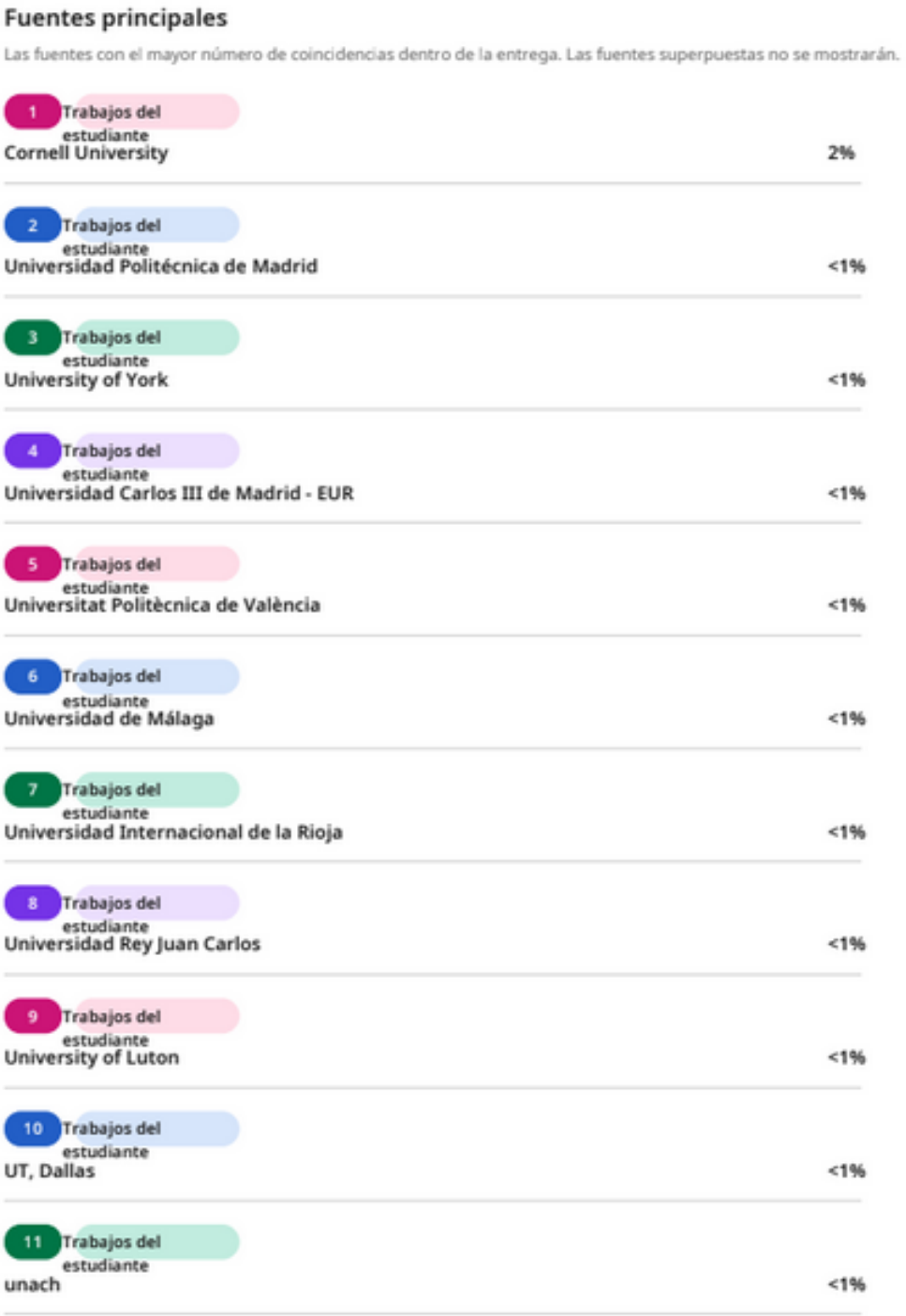
Exclusiones

- N.º de fuente excluida

Fuentes principales

0% Fuentes de Internet
0% Publicaciones
4% Trabajos entregados (trabajos del estudiante)

A.3. Fuentes principales



Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Wed Jul 09 14:11:48 CEST 2025
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)