



# UNIVERSIDAD NACIONAL DE CÓRDOBA

FACULTAD DE CIENCIAS EXACTAS, FÍSICAS Y NATURALES

CÁTEDRA DE SISTEMAS DE COMPUTADORAS - 2025

TRABAJO PRÁCTICO N°1

“**Rendimiento y Time Profiling”**

GRUPO

Fork Around & Checkout

PROFESORES

Jorge, Javier & Solinas, Miguel

ALUMNOS

| Nombre | DNI |
|---|---|
| Ávalos, Sofia | 44195495 |
| Ramirez, Valentin José | 43700362 |
| Stefanovic, Fernando | 44090692 |

# Introducción

El presente informe tiene como objetivo analizar el rendimiento de sistemas de cómputo desde una perspectiva práctica, mediante el uso de herramientas de benchmarking, análisis de tiempos de ejecución y pruebas sobre plataformas reales. El trabajo se divide en dos etapas principales:

**En la primera etapa**, se abordará el análisis del rendimiento a nivel de sistema operativo y programas de usuario. Para ello:

- Se explorarán distintos tipos de **benchmarks** (sintéticos, reducidos, de núcleo y de programas reales), seleccionando los más apropiados para las tareas que realiza cada integrante del grupo, y organizando esta relación en una tabla comparativa.

- Se investigará el **tiempo de compilación del kernel de Linux** en distintos procesadores modernos, utilizando los resultados disponibles en OpenBenchmarking, con foco en los modelos:

    - Intel Core i5-13600K

    - AMD Ryzen 9 5900X (12 núcleos)

    - AMD Ryzen 9 7950X (16 núcleos)
      Se calculará también la **aceleración** obtenida entre los distintos procesadores.

- Se realizará un tutorial de **time profiling con `gprof`**, incluyendo capturas de pantalla y conclusiones sobre el uso del tiempo por parte de las funciones dentro de un programa.

**En la segunda etapa**, el enfoque se trasladará al análisis en sistemas embebidos. Se desarrollará un código con una duración controlada (de 10 segundos) para ejecutarlo en una **placa ESP32**, con el fin de observar y comparar el comportamiento temporal del sistema en una plataforma de recursos limitados.

A lo largo del trabajo se buscará entender cómo distintas variables, tanto a nivel de hardware como de software, afectan el rendimiento general de un sistema, y cómo herramientas como los benchmarks y los perfiles temporales pueden ayudarnos a cuantificarlo y optimizarlo.

# Desarrollo

## 1. Análisis de Benchmarks

### 1.1 ¿Qué es un benchmark?

Un **benchmark** es una prueba diseñada para medir el rendimiento de un sistema o componente, bajo condiciones específicas y controladas. Permite obtener métricas comparables y reproducibles, útiles tanto para tomar decisiones de diseño o compra, como para evaluar el impacto de optimizaciones a nivel de hardware o software.

### 1.2 Clasificación de benchmarks según el tipo de prueba

Existen diferentes formas de clasificar los benchmarks. Una de las más utilizadas en el ámbito académico y técnico es según el **grado de representatividad del comportamiento real del sistema**. Bajo este criterio, se distinguen cuatro grandes grupos:

#### 1.2.1 Benchmarks sintéticos

Los **benchmarks sintéticos** son programas breves, específicamente diseñados para probar una característica puntual del sistema, como el acceso a memoria, operaciones aritméticas o lectura de disco. No se basan en cargas de trabajo reales, sino en escenarios artificiales pero ampliamente controlados.

- **Objetivo**: analizar cómo responde el sistema ante un patrón específico de uso.
- **Ventaja**: permiten aislar variables y realizar mediciones precisas.
- **Desventaja**: no siempre reflejan el rendimiento real en aplicaciones complejas.

Ejemplos:

- **CPU**: Dhrystone, Whetstone, Linpack
- **Memoria**: STREAM
- **Disco**: ATTO Disk Benchmark
- **Red**: iPerf

#### 1.2.2 Benchmarks reducidos

Los **benchmarks reducidos** son fragmentos de código real o versiones simplificadas de programas complejos, diseñados para representar patrones típicos de ejecución. No simulan un sistema completo, pero sí una parte significativa.

- **Objetivo**: modelar una carga realista pero contenida, con menos variabilidad.
- **Ventaja**: mejor aproximación a situaciones reales sin requerir grandes recursos.
- **Desventaja**: aún pueden simplificar demasiado el comportamiento de aplicaciones reales.

Ejemplos:

- **SPEC CPU2006 / CPU2017**: colecciones de programas pequeños que representan diferentes tipos de cargas (compilación, física, IA, etc.).
- **Livermore Loops**: prueba clásica que agrupa bucles representativos de aplicaciones científicas.
- **NAS Parallel Benchmarks**: representan patrones de computación paralela reducida.

### 1.2.3 Benchmarks kernel o de núcleo

Los **benchmarks de núcleo** (del inglés *kernel benchmarks*) ejecutan partes fundamentales del sistema, como el compilador, el sistema de archivos, el stack de red o el planificador. Suelen enfocarse en un aspecto central del funcionamiento del sistema operativo o del entorno de desarrollo.

- **Objetivo**: medir el rendimiento de funciones críticas del sistema.
- **Ventaja**: permiten evaluar el rendimiento real en tareas complejas y relevantes.
- **Desventaja**: pueden estar condicionados por configuraciones específicas del sistema.

**Ejemplos:**

- **Phoronix Test Suite – Build Linux Kernel**: mide el tiempo que tarda en compilar el kernel de Linux.
- **I/O Zone / Bonnie++**: analizan el rendimiento del subsistema de archivos.
- **Netperf**: mide rendimiento en la capa de red.

### 1.2.4 Benchmarks basados en programas reales

Los **benchmarks reales** utilizan aplicaciones completas, tal como serían utilizadas por un usuario final. Evalúan el rendimiento en un entorno complejo, con múltiples interacciones entre CPU, memoria, disco, sistema operativo y otras aplicaciones.

- **Objetivo**: obtener una medida directa del rendimiento percibido por el usuario.
- **Ventaja**: máxima representatividad del mundo real.
- **Desventaja**: difícil de replicar con exactitud; alto consumo de recursos.

**Ejemplos:**

- **Blender Benchmark**: para renderizado 3D.
- **Compilar Firefox / LibreOffice**: usado para medir entornos de desarrollo reales.
- **GIMP / Photoshop Batch Tests**: para edición de imágenes.
- **Video encoding con Handbrake**: en tareas multimedia.

## 1.3 Aplicación de los benchmarks según el tipo de sistema evaluado

Los benchmarks, independientemente de su clasificación (sintético, reducido, de núcleo o real), pueden orientarse a evaluar distintos subsistemas o componentes dentro de una computadora. En esta sección se describen las principales áreas evaluadas, junto con ejemplos típicos de benchmarks utilizados para cada una.

### 1.3.1 Benchmarks de hardware

Evalúan el rendimiento de los componentes físicos del sistema. Son útiles para identificar cuellos de botella a nivel de arquitectura, capacidad de procesamiento, ancho de banda y latencia.

- **CPU Benchmark**: mide la capacidad de cómputo del procesador, incluyendo cálculos enteros, en punto flotante, operaciones vectoriales y multihilo.
  *Ejemplos*: Geekbench, Cinebench, SPEC CPU.
- **GPU Benchmark**: evalúa el rendimiento gráfico en tareas como renderizado 3D, físicas en juegos y cargas paralelas.
  *Ejemplos*: Unigine Heaven, 3DMark, Blender Benchmark.
- **RAM Benchmark**: analiza la velocidad de lectura/escritura, la latencia y el ancho de banda de acceso a la memoria principal.
  *Ejemplos*: AIDA64, STREAM.

- **Benchmark de almacenamiento**: mide la velocidad de acceso a disco, tanto en lectura como en escritura, en operaciones secuenciales y aleatorias.
  *Ejemplos*: CrystalDiskMark, Bonnie++, fio.

### 1.3.2 Benchmarks de software y sistemas operativos

Evalúan el comportamiento del sistema desde el punto de vista del software y la interacción con el sistema operativo, considerando su capacidad de respuesta, eficiencia en multitarea y manejo de recursos.

- **Benchmark de rendimiento general del sistema**: mide tiempos de arranque, respuesta ante eventos del usuario, y eficiencia del planificador.
  *Ejemplos*: PassMark, PCMark, Phoronix System Test Suite.

- **Benchmark de compilación y desarrollo**: analiza el tiempo requerido para compilar grandes proyectos, útil en entornos de programación.
  *Ejemplos*: Phoronix Build Linux Kernel, compilación de Firefox o LibreOffice.

### 1.3.3 Benchmarks de red

Se enfocan en medir el rendimiento de los sistemas de comunicación, ya sea a nivel de red local o conexión a Internet. Son esenciales para analizar latencia, ancho de banda y estabilidad.

- **Benchmark de velocidad de Internet**: evalúa el ancho de banda de bajada, subida y latencia de conexión.
  *Ejemplos*: Speedtest, fast.com.

- **Benchmark de estabilidad y latencia**: mide jitter, pérdida de paquetes, tiempo de ida y vuelta (RTT) y otras métricas relevantes en aplicaciones en tiempo real.
  *Ejemplos*: comando `ping`, iPerf, Netperf.

### 1.3.4 Benchmarks de rendimiento en juegos

Estos benchmarks evalúan la capacidad de la máquina para ejecutar videojuegos con fluidez y estabilidad, lo que implica una combinación de CPU, GPU, memoria y disco.

- **Benchmarks integrados en juegos**: algunas aplicaciones incluyen su propio sistema de benchmarking para medir FPS y estabilidad gráfica bajo distintas configuraciones.
  *Ejemplos*: Shadow of the Tomb Raider Benchmark, F1 Benchmark.

- **Simuladores de carga gráfica**: programas diseñados específicamente para poner a prueba el rendimiento gráfico con escenarios complejos.
  *Ejemplos*: Unigine Heaven, 3DMark Time Spy.

## 1.4 Reflexión sobre Benchmarks representativos de Rutinas de Usuario

Aunque en el ámbito académico y profesional los benchmarks suelen ser herramientas formales y controladas, es importante reconocer que en la vida cotidiana realizamos evaluaciones de rendimiento de manera intuitiva y constante. Estas "mediciones informales" también constituyen formas de benchmarking, incluso si no se identifican como tales.

Por ejemplo, al considerar la compra de un videojuego, es habitual buscar videos en plataformas como YouTube con títulos del estilo “¿Corre Minecraft con Shaders gráficos en una GTX 1650?” o “Rendimiento de Elden Ring en i5-10400F”. Esta comparación directa de la experiencia de otros usuarios con hardware similar constituye un benchmarking orientado al consumidor.

Del mismo modo, cuando experimentamos problemas de conectividad, muchas veces accedemos a sitios como speedtest.net para medir la velocidad de descarga, subida y latencia. Este comportamiento evidencia cómo los usuarios utilizan herramientas de benchmarking para diagnosticar su entorno digital.

Incluso en nuestros dispositivos móviles, los sistemas operativos ejecutan internamente pruebas periódicas de rendimiento para tomar decisiones: por ejemplo, evaluaciones de calidad de señal Wi-Fi, consumo energético de aplicaciones en segundo plano, o priorización de procesos en función del uso reciente. Estas acciones automatizadas también forman parte de un ecosistema de benchmarking permanente, aunque invisible al usuario.

Más allá del entorno doméstico o de oficina, en el campo de la ingeniería y particularmente en sistemas embebidos y desarrollo de software, el benchmarking puede adquirir un rol activo y deliberado. Por ejemplo:

- Medir el uso de CPU y memoria de un firmware en ejecución.
- Evaluar el tiempo de respuesta de una rutina crítica en un microcontrolador (mediante timers o pines GPIO).
- Calcular el rendimiento energético de distintos modos de bajo consumo.
- Comparar versiones de un algoritmo en función de su latencia o throughput.

En estos casos, el benchmarking puede incluso integrarse como una etapa formal dentro del ciclo de desarrollo. Tal como en software se incorpora testing unitario o validación de regresión, podría considerarse el benchmarking como una pieza clave del pipeline de desarrollo, especialmente cuando se apunta a optimización, escalabilidad o eficiencia energética.

Implementar benchmarks personalizados para rutinas críticas no solo permite detectar cuellos de botella, sino también justificar decisiones de diseño o selección de hardware. Por ejemplo, comparar el tiempo de ejecución de una rutina de procesamiento de señales entre dos microcontroladores puede ser determinante al elegir el dispositivo para un producto final.

## 1.5 Relación entre tareas cotidianas y benchmarks representativos

A continuación, en la Tabla 1.1, se presenta el vínculo de las tareas comunes realizadas por los usuarios con los benchmarks que mejor representan su carga de trabajo. Esto permite identificar qué herramientas serían útiles en caso de querer evaluar objetivamente el rendimiento del sistema para cada situación.

| Tarea habitual del usuario | Benchmark representativo | Tipo de benchmark |
|---|---|---|
| Compilar proyectos en C/C++ | Phoronix Build Linux Kernel, SPEC CPU | De núcleo / Reducido |
| Ver videos en streaming | Speedtest, BufferBench | Red / Real |
| Jugar videojuegos en 3D | Unigine Heaven, 3DMark, benchmark interno del juego | Real / Sintético |
| Usar múltiples programas a la vez | PCMark, PassMark | Real |
| Medir latencia en conexión para gaming | iPerf, Netperf, comando `ping` | Sintético / Reducido |
| Editar imágenes o diseño gráfico | GIMP Benchmark Suite, Photoshop Bench | Real |
| Programar en microcontroladores (ESP32) | gprof, medición por timers o GPIO | Sintético / Propio / Reducido |
| Procesamiento de señales en firmware | Benchmark manual por medición de ejecución | Propio / Sintético |
| Evaluar eficiencia energética de un firmware | Análisis de consumo con osciloscopio / debug power tools | Propio / Real |
| Codificar video (por ejemplo, para redes sociales) | Handbrake Benchmark | Real |

*Tabla 1.1. Benchmarks Asociados a Trabajos de un Usuario de un Sistema de Computación*

## 2. Comparación de Métricas de Benchmarks Núcleo

Para esta experiencia, se analizará el tiempo de compilación del kernel de Linux como benchmark de núcleo, utilizando datos obtenidos del test `pts/build-linux-kernel-1.15.0` disponible en OpenBenchmarking.

Los procesadores seleccionados para la comparación son:

- Intel Core i5-13600K
- AMD Ryzen 9 5900X (12 núcleos / 24 hilos)
- AMD Ryzen 9 7950X (16 núcleos / 32 hilos)

A partir de los datos recolectados y la fórmula de rendimiento en **(2.1)**, se establecen los tiempos promedio de compilación del kernel y su rendimiento porcentual en la Tabla 2.1:

$$\eta_{prog\,\%} = \frac{1}{T_{prog}} \cdot 100\% \quad \textbf{(2.1)}$$

| Procesador | Tiempo de compilación (segundos) | Rendimiento |
|---|---|---|
| Intel Core i5-13600K | 83 | 1.20% |
| AMD Ryzen 9 5900X | 97 | 1.03% |
| AMD Ryzen 9 7950X | 53 | 1.88% |

*Tabla 2.1. Tiempo de Compilación en Segundos y Rendimiento por Procesador.*

## 2.1 Cálculo de aceleración (Speedup)

Se define la **aceleración** o *speedup* como **(2.2)**:

$$Speedup = \frac{\eta_{Mejor}}{\eta_{Original}} \quad \textbf{(2.2)}$$

Tomando como base el tiempo del Ryzen 9 5900X, cuya aceleración resulta unitaria debido a que evidentemente se compara con sí mismo:

- Aceleración del **Ryzen 9 7950X** respecto al 5900X:

$$Speedup = \frac{1.88\%}{1.03\%} = 1.83019$$

- Aceleración del **i5-13600K** respecto al 5900X:

$$Speedup = \frac{1.20\%}{1.03\%} = 1.16867$$

## 2.2 Eficiencia en el uso de núcleos

Se define la **eficiencia en uso de núcleos** como el cociente entre la aceleración y el número de núcleos, tal como se aprecia en **(2.3)**. Dada esta definición, es posible calcular la eficiencia para sistemas con los procesadores planteados (Tabla 2.2)

$$Eficiencia = \frac{Speedup_n}{n} \quad \textbf{(2.3)}$$

| Procesador | Speedup (vs 1 núcleo teórico) | Núcleos | Eficiencia-Núcleos |
|---|---|---|---|
| Intel Core i5-13600K | 1.16867 | 14 | 0.0835 |
| AMD Ryzen 9 5900X | 1 | 12 | 0.0833 |
| AMD Ryzen 9 7950X | 1.83019 | 16 | 0.1144 |

*Tabla 2.2. Eficiencia en términos de la cantidad de núcleos para distintos procesadores*

Es importante aclarar que el i5-13600K tiene 6 núcleos de rendimiento más 8 núcleos de eficiencia, totalizando 14 núcleos físicos. Por lo que al ser una arquitectura híbrida, el análisis de eficiencia puede variar según cómo se cuenten.

## 2.3 Análisis de eficiencia por costo

Para evaluar la **eficiencia económica**, se propone utilizar como indicador el costo en USD del producto, utilizando la misma ecuación vista en **(2.3)**.

Utilizando precios promedio estimados del mercado (referencia: Amazon) se encuentra la eficiencia según el costo en la Tabla 2.3.

| Procesador | Costo estimado (USD) | Speedup | Eficiencia-Costo |
|---|---|---|---|
| Intel Core i5-13600K | 230 | 1.16867 | 0.00508 |
| AMD Ryzen 9 5900X | 338 | 1 | 0.00296 |
| AMD Ryzen 9 7950X | 672 | 1.83019 | 0.00272 |

*Tabla 2.3. Eficiencia en términos de costo de producto para distintos procesadores*

## 2.4 Conclusiones parciales

- El **Ryzen 9 7950X** muestra el mejor tiempo de compilación absoluto y la mayor aceleración, como es esperable por su mayor cantidad de núcleos y arquitectura más reciente.
- En términos de **eficiencia por núcleo**, el **Ryzen 9 7950X** demuestra también tener un mejor aprovechamiento del paralelismo disponible.

- Al analizar la **eficiencia económica**, el **Intel Core i5-13600K** ofrece la mejor relación rendimiento-precio, lo cual podría ser determinante al planificar compras en contextos académicos o presupuestos limitados.

# 3. Time Profiling en Sistemas de Cómputo

El **time profiling** es una técnica fundamental en el análisis de rendimiento de software. Permite medir el tiempo que un programa dedica a ejecutar cada una de sus funciones o secciones, identificando así cuellos de botella y puntos críticos que pueden ser optimizados. A diferencia de los benchmarks, que evalúan el rendimiento global de un sistema o componente, el profiling se enfoca en el comportamiento interno de un programa específico durante su ejecución.

Entre sus principales objetivos se encuentran:

- Cuantificar el uso del procesador por parte de cada función.
- Detectar rutinas ineficientes o innecesarias.
- Visualizar la jerarquía de llamadas y dependencias entre funciones.
- Evaluar el impacto de cambios o refactorizaciones en el rendimiento.

## 3.1. Herramientas comunes de profiling

En entornos Unix/Linux, existen múltiples herramientas para realizar time profiling, entre ellas:

- `gprof`: Instrumenta el código durante la compilación para generar estadísticas sobre las funciones ejecutadas, su frecuencia y el tiempo relativo que consumen.
- `perf`: Utiliza contadores de hardware y mecanismos de muestreo para capturar eventos de bajo nivel del sistema, permitiendo un análisis detallado sin modificar el código fuente.
- `valgrind` **(modo `callgrind`)**: Aunque no mide tiempo real, permite evaluar el comportamiento lógico y estructural del programa en cuanto a llamadas y costos simulados.
- `gprof2dot` **+** `graphviz`: Utilizados como herramientas auxiliares para visualizar de forma gráfica el grafo de llamadas generado por `gprof`.

## 3.2. Aplicaciones del profiling

El profiling es especialmente útil en contextos como:

- Desarrollo de firmware y software embebido, donde los recursos son limitados.
- Algoritmos complejos que requieren optimización temporal.
- Análisis de programas que presentan lentitud aparente sin una causa evidente.
- Comparación de versiones de código o estrategias de implementación.

### 3.3. Implementación práctica

Como parte de este trabajo, se llevó a cabo una experiencia de profiling sobre un programa en lenguaje C. Para ello:

- Se implementó un conjunto de funciones artificiales con estructuras repetitivas (bucles grandes) para simular carga computacional.
- Se compiló el código con soporte para `gprof` y se generó un archivo de perfil (`gmon.out`), a partir del cual se extrajo un informe detallado del uso temporal por función.
- Se exploraron diferentes flags de `gprof` para visualizar tanto perfiles planos como gráficos de llamadas.
- También se empleó `perf` para complementar el análisis, accediendo a métricas del sistema más cercanas al hardware.
- Finalmente, se generaron visualizaciones gráficas con `gprof2dot`, que facilitaron la interpretación estructural del flujo de ejecución.

Este enfoque permitió reforzar los conceptos teóricos sobre profiling mediante una aplicación directa y controlada, para consolidar así la comprensión de cómo medir y optimizar el rendimiento de software real. El mismo se encuentra desarrollado en el siguiente repositorio de GitHub por cada integrante del equipo.

## 4. Time Profiling en Embebidos: Efecto de la Frecuencia

En esta sección se analiza cómo la **frecuencia del reloj del procesador** afecta los tiempos de ejecución de un mismo código en un sistema embebido. Para ello se realizaron pruebas de profiling sobre un **ESP8266**, un microcontrolador ampliamente utilizado en sistemas de bajo costo, que permite cambiar dinámicamente su frecuencia entre 80 MHz y 160 MHz.

### 4.1 Objetivo del análisis

El objetivo principal fue cuantificar la diferencia en tiempo de ejecución de una rutina fija cuando se modifica la frecuencia del CPU. Esto permite:

- Verificar empíricamente la relación inversa entre frecuencia y tiempo.
- Evaluar el impacto de operar a menor frecuencia en términos de rendimiento.
- Proyectar decisiones de diseño donde se debe balancear eficiencia energética con respecto a la velocidad.

### 4.2 Descripción de la rutina evaluada

Se diseñó una función que realiza una **suma de enteros** y una **suma con números en punto flotante**, simulando una carga de trabajo matemática mixta, típica de aplicaciones de firmware como controladores o procesamiento de datos.

A continuación, se presenta el código utilizado:

```
#include <Arduino.h>

void ejecutarTest(int frecuenciaMHz) {
  Serial.println();
  Serial.println("========================================");
  Serial.print("⚙️  Iniciando benchmark en ");
  Serial.print(frecuenciaMHz);
  Serial.println(" MHz");
  Serial.println("========================================");

  setCpuFrequencyMhz(frecuenciaMHz);
  delay(500);  // Esperar estabilización del sistema

  // --------- Bucle de enteros ---------
  Serial.println("🔢 [Test 1] Suma de enteros en ejecución...");

  unsigned long tInicio = millis();
  volatile long sumaEnteros = 0;
  long repeticionesEnteros = 100000000;

  for (long i = 0; i < repeticionesEnteros; i++) {
      sumaEnteros += i;
  }

  unsigned long tFin = millis();
  Serial.print("⏱️  Duración: ");
  Serial.print(tFin - tInicio);
  Serial.println(" ms");

  // --------- Bucle de floats ---------
  Serial.println("🌊 [Test 2] Now with the floats...");

  tInicio = millis();
  volatile float sumaFloats = 0.0;
  long repeticionesFloats = 10000000;

  for (long i = 0; i < repeticionesFloats; i++) {
      sumaFloats += 0.1;
  }

  tFin = millis();
  Serial.print("⏱️  Duración: ");
  Serial.print(tFin - tInicio);
  Serial.println(" ms");
```

```
  Serial.println("✅ Benchmark completado para esta frecuencia.");
  Serial.println("----------------------------------------\n");

  delay(2000);  // Pausa entre pruebas
}

void setup() {
  Serial.begin(115200);
  while (!Serial) {}

  delay(1000);
  Serial.println("\n💻 Benchmark de rendimiento ESP32 💻");
  Serial.println("🧪 Grupo: ForkAround&FindOut");
  Serial.println("📊 Evaluación comparativa de frecuencias\n");

  int frecuencias[] = {80, 160};

  for (int i = 0; i < 2; i++) {
      ejecutarTest(frecuencias[i]);
  }

  Serial.println("🏁 Todos los tests han finalizado correctamente.");
}

void loop() {
  // Nada por aquí
}
```

## 4.3 Resultados obtenidos

Se ejecutó la rutina con el microcontrolador configurado a **80 MHz** y a **160 MHz**. Estos resultados, ilustrados en la Figura 4.1, reflejan una relación directa entre la frecuencia del CPU y el tiempo necesario para ejecutar una rutina computacional.

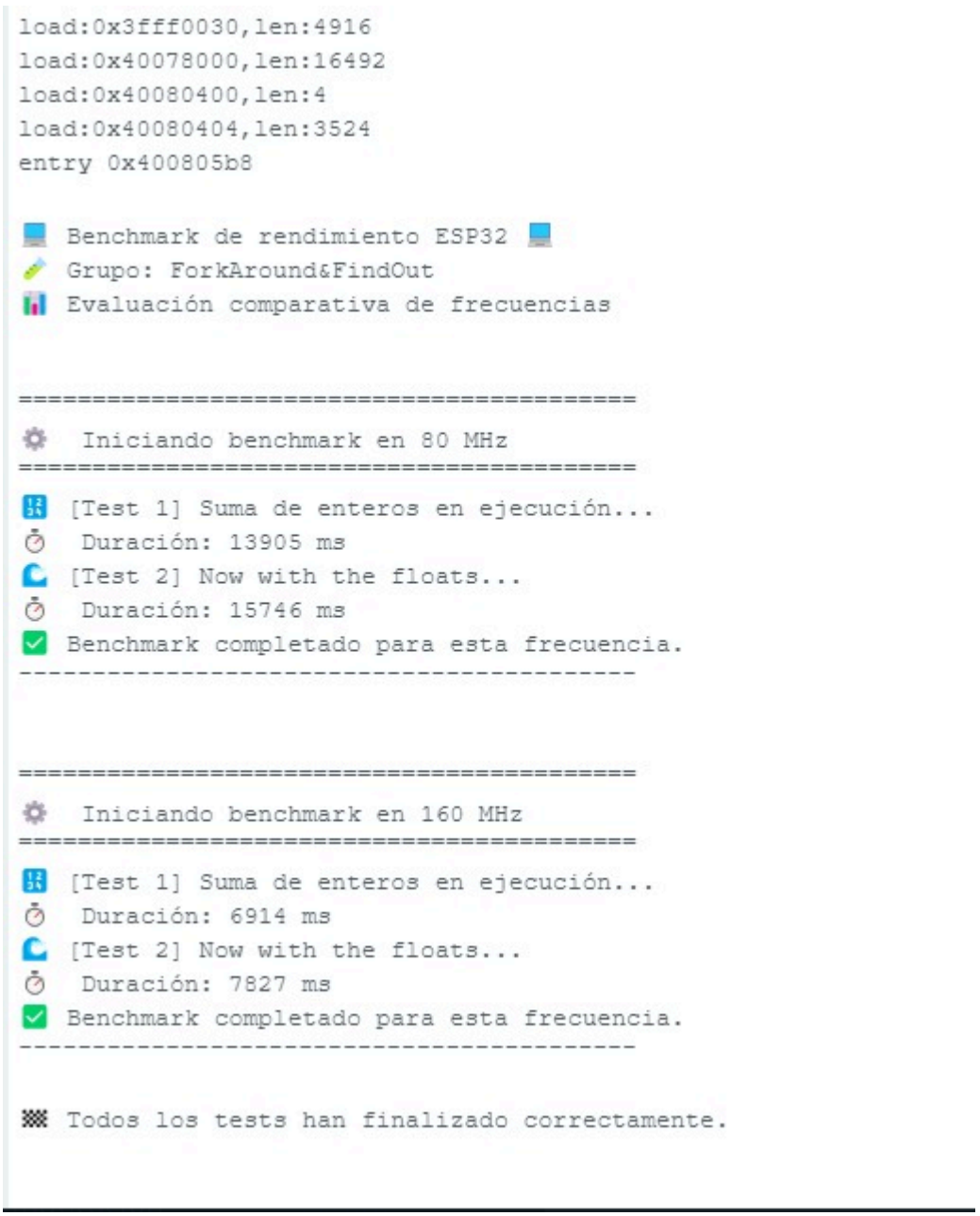


*Figura 4.1. Resultado del canal serial ante la ejecucion del codigo.*

| Frecuencia (MHz) | Test Realizado | Tiempo de ejecución (s) |
|---|---|---|
| 80 | Suma de Enteros | 13,91 |
| 80 | Suma de Floats | 15,75 |
| 160 | Suma de Enteros | 6,91 |
| 160 | Suma de Floats | 7,83 |

*Tabla 4.1. Recopilación de Resultados*

Tal como se observa en la Tabla 4.1 entonces, al duplicar la frecuencia del procesador se reduce aproximadamente a la mitad el tiempo necesario para ejecutar ambas rutinas.

## 4.4. Conclusiones

- La prueba confirma que el rendimiento de una rutina matemática intensiva mejora linealmente con la frecuencia del procesador.
- Esto valida el uso de técnicas de profiling incluso en sistemas embebidos sin herramientas avanzadas, utilizando funciones estándar de medición temporal.
- Este tipo de análisis resulta útil tanto para identificar cuellos de botella como para tomar decisiones de optimización, como por ejemplo reducir la frecuencia del reloj en tramos de ejecución no críticos para ahorrar energía sin afectar el funcionamiento general del sistema.

# Conclusión

Este trabajo permitió explorar de forma práctica cómo evaluar y optimizar el rendimiento de sistemas de cómputo. En la **primera etapa**, se clasificaron distintos tipos de **benchmarks** (sintéticos, reducidos, de núcleo y reales), se analizó el tiempo de compilación del kernel de Linux en procesadores modernos, y se aplicaron métricas como **aceleración** y **eficiencia por núcleo versus costo**. Además, se realizó un tutorial detallado de **time profiling con** `gprof`, reforzando la capacidad de detectar cuellos de botella en el código.

En la **segunda etapa**, se trasladó el análisis a un entorno embebido con un **ESP32**, donde se comprobó empíricamente cómo la **frecuencia del procesador** impacta directamente en los tiempos de ejecución de una rutina computacional, validando la utilidad del profiling incluso en sistemas de recursos limitados.