



Universidad  
Nacional  
de Córdoba



Facultad de  
Ciencias Exactas  
Físicas y Naturales

# UNIVERSIDAD NACIONAL DE CÓRDOBA

## FACULTAD DE CIENCIAS EXACTAS, FÍSICAS Y NATURALES

### SISTEMAS DE COMPUTACIÓN

#### Trabajo Práctico N°1: Rendimiento y Time Profiling

Grupo: ByteShifters

Nombre	DNI
Gil Cernich, Manuel	43137748
Pallardó, Agustín	43147213
Saporito, Franco	39495234

Docentes

Jorge, Javier  
Solinas, Miguel

Marzo del 2024

# Objetivo

El objetivo de esta tarea es poner en práctica los conocimientos sobre performance y rendimiento de los computadores. El trabajo consta de dos partes, la primera es utilizar benchmarks de terceros para tomar decisiones de hardware y la segunda consiste en utilizar herramientas para medir la performance de nuestro código.

## **Armar una lista de benchmarks.**

¿Cuáles les serían más útiles a cada uno?

¿Cuáles podrían llegar a medir mejor las tareas que ustedes realizan a diario?

Pensar en las tareas que cada uno realiza a diario y escribir en una tabla de dos entradas las tareas y que benchmark la representa mejor.

¿Cuál es el rendimiento de estos procesadores para compilar el kernel de linux ?

Intel Core i5-13600K

AMD Ryzen 9 5900X 12-Core

Cual es la aceleración cuando usamos un AMD Ryzen 9 7950X 16-Core

<https://openbenchmarking.org/test/pts/build-linux-kernel-1.15.0>

**En un informe deberán responder a las siguientes preguntas y mostrar con capturas de pantalla la realización del tutorial descripto en time profiling adjuntando las conclusiones sobre el uso del tiempo de las funciones.**

# Profiling

El rendimiento en la programación se refiere a la velocidad y eficiencia con la que un programa o algoritmo realiza una tarea, en otras palabras es la capacidad de un sistema para ejecutar operaciones de manera rápida y efectiva. El rendimiento se mide en términos de tiempo de ejecución y el uso de la memoria.

Cuando hablamos de tiempo de ejecución, podemos hacer referencia a la eficiencia, es decir cuanto tiempo tarda un programa en completar una tarea. Por ejemplo, si hablamos de eficiencia de un algoritmo de búsqueda, un buen algoritmo es aquel que toma el mínimo tiempo de ejecución y uso de memoria posible.

El **profiling** es un proceso mediante el cual se recopila información detallada sobre cómo se ejecuta un programa. Su objetivo principal es identificar bottlenecks, ineficiencias y áreas de mejora en el código. El **profiling** implica medir y analizar aspectos como el tiempo de ejecución, el uso de memoria y la frecuencia de llamadas a funciones y proporciona una visión profunda del comportamiento del programa, lo que ayuda a los desarrolladores a optimizarlo.

En el siguiente trabajo utilizaremos **gprof** para ejecutar un código en C, desde el cual obtendremos información de cómo se ejecuta el código, que funciones consumen más tiempo y recursos y poder llegar a analizar donde se pueden realizar mejoras. También utilizaremos **perf** la cual es una herramienta basada en muestreo que se integra con el kernel de Linux. Ofrece detalles sobre eventos de hardware y software, como caché, fallos de página y ciclos de CPU.

Este análisis se hará sobre una computadora con:

- **Intel Core i5-7200U 4-Core**
- **8GB Memoria RAM**
- **SSD NVMe 240GB**

## Analisis con gprof

Vamos a realizar el análisis sobre un código en C muy simple el cual son diferentes funciones, a las que se les asignó una instrucción **for** vacía para poder consumir tiempo de ejecución.

```
#include <stdio.h>

void new_func1(void);

void func1(void)
{
    printf("\n Inside func1 \n");
    int i = 0;
    for( ; i<0xffffffff ; i++);
    new_func1();
    return;
}

static void func2(void)
{
    printf("\n Inside func2 \n");
    int i = 0;
    for( ; i<0xffffffff ; i++);
    return;
}

int main(void)
{
    printf("\n Inside main()\n");
    int i = 0;

    for( ; i<0xfffff11 ; i++);
    func1();
    func2();

    return 0;
}

void new_func1(void)
{
    printf("\n Inside new_func1()\n");
    int i = 0;

    for( ; i<0xfffff66 ; i++);

    return;
}
```

Para poder realizar el **profiling** debemos compilar y buildear nuestro código mediante gcc y la flag **-pg**.

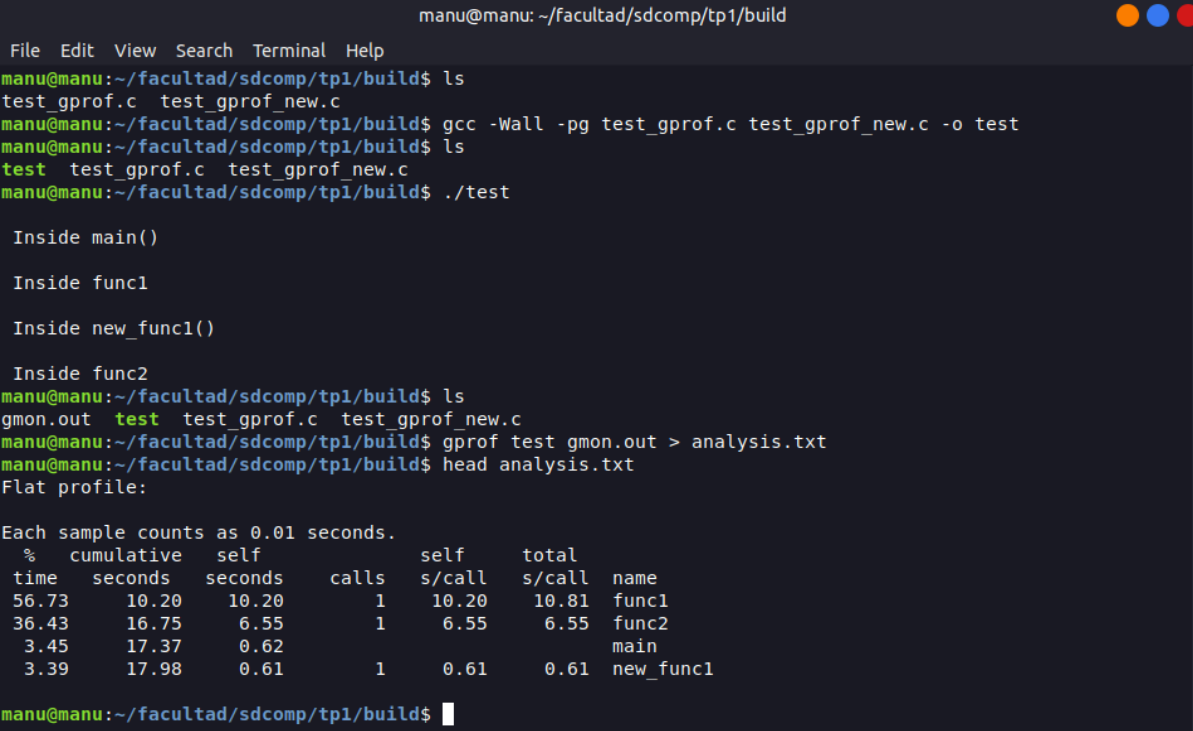
- **pg**: Generate extra code to write profile information suitable for the analysis program gprof. You must use this option when compiling the source files you want data about, and you must also use it when linking.

Una vez creado nuestro binario, lo ejecutamos para que realice sus instrucciones y a su vez el profiling se lleve a cabo, para que al finalizar se genere nuestro archivo **gmon.out** el cual va a contener toda nuestra información sobre el rendimiento como por ejemplo:

- Tiempo de CPU dedicado a cada función.
- Número de llamadas a cada función.
- Funciones de llamada para cada función.
- Funciones llamadas por cada función.

Para poder analizar este **gmon.out** utilizaremos una herramienta llamada **gprof** la cual nos proporciona informes detallados sobre el rendimiento del programa.

- **gprof**: "gprof" produces an execution profile of C, Pascal, or Fortran77 programs. "gprof" reads the given object file (the default is "a.out") and establishes the relation between its symbol table and the call graph profile from gmon.out.



```
manu@manu: ~/facultad/sdcomp/tp1/build
File Edit View Search Terminal Help
manu@manu:~/facultad/sdcomp/tp1/build$ ls
test_gprof.c  test_gprof_new.c
manu@manu:~/facultad/sdcomp/tp1/build$ gcc -Wall -pg test_gprof.c test_gprof_new.c -o test
manu@manu:~/facultad/sdcomp/tp1/build$ ls
test  test_gprof.c  test_gprof_new.c
manu@manu:~/facultad/sdcomp/tp1/build$ ./test

Inside main()

Inside func1

Inside new_func1()

Inside func2
manu@manu:~/facultad/sdcomp/tp1/build$ ls
gmon.out  test  test_gprof.c  test_gprof_new.c
manu@manu:~/facultad/sdcomp/tp1/build$ gprof test gmon.out > analysis.txt
manu@manu:~/facultad/sdcomp/tp1/build$ head analysis.txt
Flat profile:

Each sample counts as 0.01 seconds.
 %   cumulative   self           calls   self   total    name
time  seconds    seconds               s/call  s/call  s/call
56.73    10.20     10.20                1    10.20   10.81  func1
36.43    16.75      6.55                 1     6.55    6.55  func2
 3.45    17.37      0.62                 1     0.61    0.61  main
 3.39    17.98      0.61                 1     0.61    0.61  new_func1

manu@manu:~/facultad/sdcomp/tp1/build$
```

Fig 1. Ejemplo de instrucciones para generar el análisis.

Es posible visualizar estos datos obtenidos con **gprof2dot** la cual es una herramienta para convertir la salida generada por **gprof** de un profiling en un gráfico dot que puede visualizarse mediante **graphviz**.

```
manu@manu: ~/facultad/sdcomp/tp1/build
File Edit View Search Terminal Help
manu@manu:~/facultad/sdcomp/tp1/build$ gprof2dot analysis.txt > analysis.dot
manu@manu:~/facultad/sdcomp/tp1/build$ dot -Tpng analysis.dot -o analysis.png
manu@manu:~/facultad/sdcomp/tp1/build$
```

Fig 2. Comandos para obtener el gráfico.

En el siguiente gráfico podemos observar como nuestra función **func1** ocupa el 60% de tiempo de ejecución por lo cual podría ser un posible punto de falla de rendimiento o un posible punto de optimización de código.

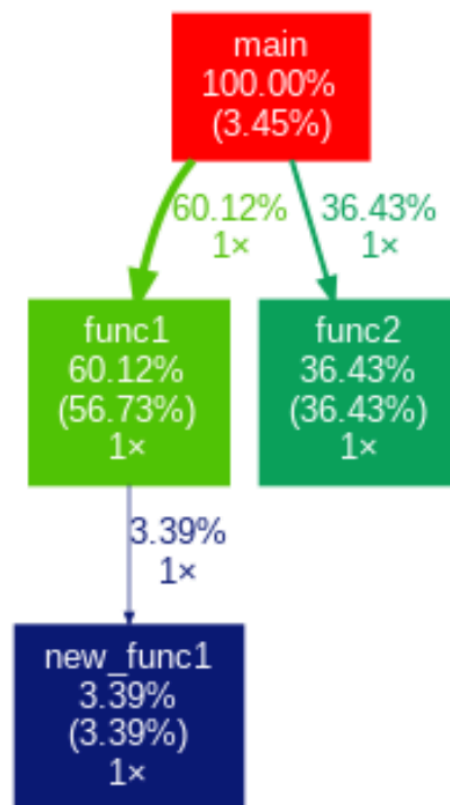


Fig 3. Gráfico generado por gprof2dot.

## Resultados con gprof

El análisis proporcionado por **gprof** nos da información sobre el rendimiento del programa, podemos ver como nuestra **func1** es la que consume la mayoría del tiempo de ejecución del programa con un aproximado del 56% del tiempo de ejecución, esto nos puede llegar a dar a entender que es una función crítica dentro del programa y deberíamos analizarla en detalle para buscarle algún tipo de optimización.

También podemos ver que un 36% de tiempo de ejecución es consumido por **func2** que aunque no sea tan dominante como **func1** aun se podría proporcionar algún tipo de optimización y llegar a resultados más favorables en términos de tiempo de ejecución.

Para la **new\_func1** si bien lleva poco tiempo de ejecución hay que tener en cuenta que está anidada dentro de **func1**.

```
manu@manu:~/facultad/sdcomp/tp1/build$ cat analysis_p.txt
Flat profile:

Each sample counts as 0.01 seconds.
 %   cumulative   self           self       total
time  seconds    seconds   calls   s/call   s/call   name
56.73    10.20    10.20        1    10.20    10.81  func1
36.43    16.75     6.55        1     6.55     6.55  func2
 3.45    17.37     0.62         1     0.62     0.62  main
 3.39    17.98     0.61         1     0.61     0.61  new_func1
```

Fig 4. Análisis de tiempos obtenidos.

## Analisis con perf

**Perf** es una herramienta muy útil para el análisis de rendimiento en sistemas basados en linux, este es un profiler (como lo es gprof) la cual hace una medición y lleva un registro de los diversos eventos de hardware y software luego de ejecutar un código.

A continuación tenemos un ejemplo de uso en el cual ejecutamos el mismo binario generado y analizado con **gprof**, para utilizar **perf** es necesario ejecutarlo con permisos de administrador por ello se utiliza sudo.

```
manu@manu: ~/facultad/sdcomp/tp1/build
File Edit View Search Terminal Help
manu@manu:~/facultad/sdcomp/tp1/build$ sudo perf record ./test

Inside main()

Inside func1

Inside new_func1()

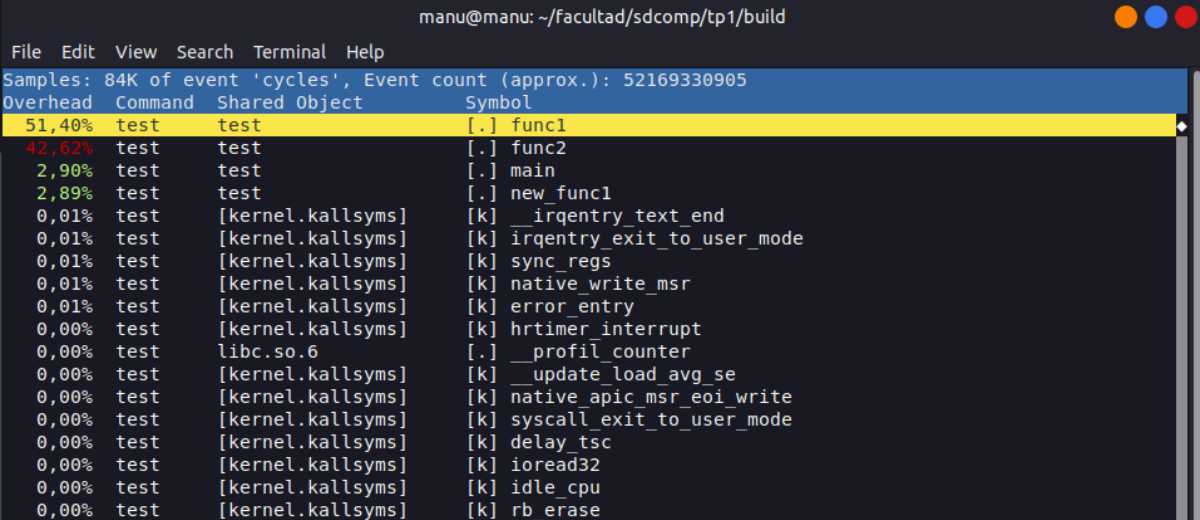
Inside func2
[ perf record: Woken up 12 times to write data ]
[ perf record: Captured and wrote 3,239 MB perf.data (84269 samples) ]
manu@manu:~/facultad/sdcomp/tp1/build$ sudo perf report
manu@manu:~/facultad/sdcomp/tp1/build$
```

Fig 5. Uso de la herramienta perf.

## Resultados con perf

Como vimos anteriormente con **gprof** los resultados obtenidos con **perf** son muy similares, las pequeñas variaciones pueden ser por la metodología o la forma de obtener los datos de tiempos de ejecución.

Acá también podemos ver como **func1** nos ocupa un aproximado de 52% de tiempo de ejecución y **func2** un 42%.



Overhead	Command	Shared Object	Symbol
51,40%	test	test	[.] func1
42,62%	test	test	[.] func2
2,90%	test	test	[.] main
2,89%	test	test	[.] new_func1
0,01%	test	[kernel.kallsyms]	[k] __irqentry_text_end
0,01%	test	[kernel.kallsyms]	[k] irqentry_exit_to_user_mode
0,01%	test	[kernel.kallsyms]	[k] sync_regs
0,01%	test	[kernel.kallsyms]	[k] native_write_msr
0,01%	test	[kernel.kallsyms]	[k] error_entry
0,00%	test	[kernel.kallsyms]	[k] hrtimer_interrupt
0,00%	test	libc.so.6	[.] __profil_counter
0,00%	test	[kernel.kallsyms]	[k] __update_load_avg_se
0,00%	test	[kernel.kallsyms]	[k] native_apic_msr_eoi_write
0,00%	test	[kernel.kallsyms]	[k] syscall_exit_to_user_mode
0,00%	test	[kernel.kallsyms]	[k] delay_tsc
0,00%	test	[kernel.kallsyms]	[k] ioread32
0,00%	test	[kernel.kallsyms]	[k] idle_cpu
0,00%	test	[kernel.kallsyms]	[k] rb_erase

Fig 6. Resultados obtenidos por perf.



# Benchmarks

Un benchmark es una prueba diseñada para medir y comparar el rendimiento de sistemas informáticos o componentes. Proporciona datos objetivos sobre la velocidad y eficiencia de hardware o software, ayudando en la toma de decisiones sobre qué dispositivos o programas son los más adecuados para ciertas tareas.

## Lista de benchmarks

Dependiendo de lo que se busca medir, existen diferentes tipos de benchmarks.

Algunos de ellos son:

- **Benchmarks de CPU:**
  - Estos evalúan el rendimiento del procesador en términos de velocidad de cálculo y capacidad de manejar múltiples tareas simultáneamente.
  - Ejemplos incluyen SPEC CPU, Geekbench, PassMark CPU
- **Benchmarks de GPU:**
  - Se enfocan en medir la capacidad de procesamiento gráfico de nuestra tarjeta gráfica, la cual es importante para aplicaciones de diseño 3D o para videojuegos.
  - Ejemplos incluyen 3DMark, GFXBench
- **Benchmarks de unidades de almacenamiento:**
  - Los benchmarks de unidades de almacenamiento evalúan, por lo general, la velocidad de lectura y escritura. Hay que reconocer que hoy en día existen sistemas mecánicos y eléctricos para almacenar información, y entre estos se pueden experimentar diferentes resultados.
  - Ejemplos incluyen PassMark, CrystalDiskMark, UserBenchmark
- **Benchmarks de RAM:**
  - Evalúan la velocidad de acceso y transferencia de datos de la memoria RAM.
  - Ejemplos incluyen MemTest86, AIDA64 y SiSoftware Sandra.
- **Benchmarks de red:**
  - Miden el rendimiento y velocidad de las conexiones de red, útiles para diagnosticar problemas de velocidad o estabilidad.
  - Ejemplos incluyen comando ping, iPerf o paginas como speedtest.com

## Benchmarks cotidianos

Para las tareas cotidianas del día a día existen diferentes tipos de benchmarks, a continuación nombraremos tareas cotidianas nuestras y el que nos parece que es el benchmark más útil para ello.

<b>Rendimiento de red</b>	<a href="https://fast.com/">https://fast.com/</a>
<b>Duración de Batería</b>	PCMark
<b>Prueba de memoria RAM</b>	MemTest86
<b>Prueba de discos</b>	CrystalDiskInfo
<b>Prueba de rendimiento código C</b>	SPEC CPU

## Rendimiento de procesadores para compilar el Kernel de Linux

El rendimiento es una medida de la eficacia con la que un sistema, componente o software realiza una determinada tarea. Se puede entender como la capacidad del sistema para completar una tarea específica, y este rendimiento se puede medir mediante diferentes métricas dependiendo del contexto.

Para este problema se solicita calcular el rendimiento para compilar el kernel de Linux de los siguientes procesadores:

- **Intel Core i5-13600K**
- **AMD Ryzen 9 5900X 12-Core**

En la siguiente [página web](#) podemos ver diferentes tipos de procesadores y sus métricas al compilar el kernel de linux con las configuraciones por default.

## Timed Linux Kernel Compilation 6.1

Build: defconfig

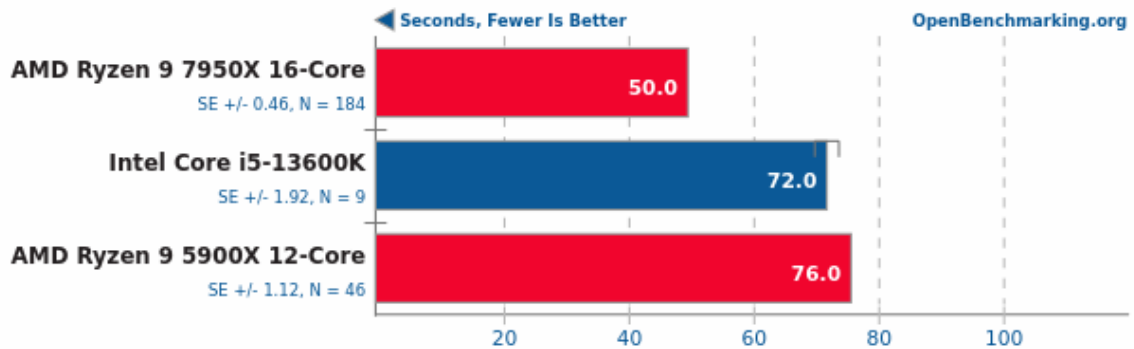


Fig 7. Tiempos de compilación de kernel de linux.

## Cálculo rendimientos

El rendimiento es una forma de medir la capacidad que tiene un sistema para realizar una cierta tarea. Se suele medir en función al tiempo en que demora en realizar cierta tarea. En estos casos, estuvimos analizando el tiempo de compilación del kernel 6.1 de Linux, donde los valores suelen comenzar del minuto hacia adelante, o un poco menos en procesadores más potentes.

Hay una forma de calcular el rendimiento matemáticamente, para lo cuál se necesitan ciertos datos, como lo son la frecuencia del procesador, su período, los ciclos por instrucción, la cantidad de instrucciones de la tarea a realizar.

En las siguientes fórmulas se puede apreciar cómo obtener estos valores:

$$f_{\text{CPU}} = \frac{\text{n}^{\circ} \text{ ciclos}}{\text{segundo}}$$

$$T_{\text{CPU}} = \frac{1}{f_{\text{CPU}}}$$

$$CPI = \frac{\sum_{i=1}^n N^{\circ} \text{ Instruc}_i * CPI_i}{N^{\circ} \text{ InstrucTot}}$$

Para completar, el rendimiento de cierto procesador para realizar cierta tarea es equivalente a 1 dividido el tiempo en que demora en completarla, y este tiempo es equivalente a el número de instrucciones de la tarea, por el tiempo del período del procesador (o la frecuencia inversa), por los ciclos por instrucción.

$$T_{\text{instrucción}} = \text{CPI} * T_{\text{CPU}}$$

$$T_{\text{prog}} = N^{\circ} \text{ instrucciones} * \text{CPI} * T_{\text{CPU}}$$

$$\eta_{\text{prog}} = \frac{1}{T_{\text{prog}}} = \frac{1}{N^{\circ} \text{instruc} * \text{CPI} * T_{\text{CPU}}} = \frac{f_{\text{CPU}}}{N^{\circ} \text{instruc} * \text{CPI}} s^{-1}$$

Por ende, para un procesador que obtiene un valor de rendimiento más alto que otro, quiere decir que es más veloz para realizar esa tarea comparado con su oponente.

Para el procesador **Intel Core i5 13600K** con un tiempo de 72 [s] tenemos un rendimiento:

$$\eta_{\text{prog}} = \frac{1}{T_{\text{prog}}} = \frac{1}{72} = 0.0139 = 1.39\%$$

Para el procesador **AMD Ryzen 9 5900x 12-Core** con un tiempo de 76 [s] tenemos un rendimiento:

$$\eta_{\text{prog}} = \frac{1}{T_{\text{prog}}} = \frac{1}{76} = 0.0131 = 1.31\%$$

Para el procesador **AMD Ryzen 9 7950x 16-Core** con un tiempo de 50 [s] tenemos un rendimiento:

$$\eta_{\text{prog}} = \frac{1}{T_{\text{prog}}} = \frac{1}{50} = 0.02 = 2\%$$

Teniendo en cuenta solo los primeros dos procesadores, se concluye que según los benchmarks de [openbenchmarking.org](https://openbenchmarking.org), si se compila un kernel de Linux, será más veloz realizarlo en el procesador **Intel Core i5 13600k 14-Core**.

## Cálculo de aceleración

La aceleración es una forma de medir la mejora en rendimiento entre procesadores o cuando se cambia otro componente que altere su rendimiento. Se puede calcular con el tiempo de ejecución de una tarea con el sistema original, sobre su tiempo de ejecución con el sistema mejorado. Mientras mayor sea el valor, mejor será el sistema mejorado.

$$\text{Speedup} = \frac{\text{Rendimiento Mejorado}}{\text{Rendimiento Original}} = \frac{EX_{\text{CPU Original}}}{EX_{\text{CPU Mejorado}}}$$

La aceleración si utilizamos al **AMD Ryzen 9 5900x 12-Core** como referencia para calcular la nueva aceleración del **AMD Ryzen 9 7960x 16-Core**

#### **AMD Ryzen 9 5900x 12-Core**

$$speedup = \frac{76}{76} = 1$$

$$eficiencia = \frac{1}{12} = 0.083 = 8.3\%$$

#### **AMD Ryzen 9 7950x 16-Core**

$$speedup = \frac{76}{50} = 1.52$$

$$eficiencia = \frac{1.52}{16} = 0.095 = 9.5\%$$

Con estos datos podemos concluir que el procesador que mejor uso hace de sus núcleos es el **AMD Ryzen 9 7950x 16-Core** y que tiene un rendimiento 1.52% mayor a su comparación.

Podemos calcular la eficiencia del uso de sus núcleos en relación al costo del CPU dividiendo su rendimiento sobre el costo del CPU obtenidos en [PCPartPicker](#).

#### **Intel Core I5 13600K 14-Core**

$$ef = \frac{1.39}{295} = 0.00471 = 0.471\%$$

#### **AMD Ryzen 9 5900x 12-Core**

$$ef = \frac{1.31}{278} = 0.00472 = 0.472\%$$

#### **AMD Ryzen 9 7950x 16-Core**

$$ef = \frac{2}{585} = 0.0034 = 0.34\%$$

Podemos concluir que el **AMD Ryzen 9 5900x 12-Core** es el que tiene mejor costo en relación al uso eficiente de sus núcleos, aunque la diferencia no es notoria.

## **Frecuencias en microcontrolador**

La frecuencia es la velocidad de operación del reloj interno de un microcontrolador/procesador, esta se mide en ciclos por segundo o Hertz [Hz]. Es decir que determina la velocidad con la que el microcontrolador puede ejecutar instrucciones y procesar datos.

Para poder analizar cómo afecta el cambio de frecuencia al tiempo de ejecución, utilizamos un microcontrolador ESP8266 al cual se le puede cambiar la frecuencia del clock entre 80

[MHz] y 160 [MHz]. Utilizamos un código en C en el cual simulamos tiempo de ejecución con un ciclo que resuelve una operación matemática.

```
for (int i = 0; i < 1000; i++)  
{  
    float result = sqrt(i * 1.0);  
}
```

Al hacer los tests con estas dos diferentes frecuencias llegamos a los siguientes resultados.

- Time taken at 160MHz: 6925 microseconds
- Time taken at 80MHz: 13855 microseconds

Podemos ver que al duplicar la frecuencia del reloj se divide aproximadamente por la mitad el tiempo de ejecución, pero aunque las instrucciones se ejecuten en la mitad del tiempo esto también aumenta la temperatura y aumenta el consumo de energía, lo que puede afectar la estabilidad del sistema.