Implementación de métodos computacionales

(Gpo 601)

Actividad Integradora 3.

“Programación Paralela”

Asesor:

Pedro Oscar Pérez Murueta.

Alumno:

Yael Charles Marin - A01711111

12 de junio 2025

Introducción.

Se quería desarrollar un algoritmo que determinara la suma de todos los números primos menores a 5 millones. Con el objetivo de evaluar cuál implementación ofrecía el mejor rendimiento, se crearon tres versiones distintas usando tecnologías multihilo y CUDA, basadas en el mismo algoritmo para verificar si un número es primo y, en caso afirmativo, sumarlo. Las implementaciones fueron las siguientes.

Desarrollo de las soluciones.

1.- Implementación secuencial.

Se creó una función auxiliar llamada **isPrime()** que recibe un número como parámetro y verifica si es primo. Para ello, utiliza un ciclo que itera desde 2 hasta la raíz cuadrada del número, comprobando si es divisible **exactamente** por alguno de esos valores. Si encuentra un divisor, retorna **false**; de lo contrario, retorna **true**, indicando que el número es **primo**.

Posteriormente, se utiliza la función **sum()**, la cual recibe un arreglo con los números del 1 al 5 millones. Con ayuda de un acumulador y un ciclo que recorre el arreglo desde la posición 0 hasta size - 1, se invoca **isPrime()** para cada número. Si el resultado es **true**, se suma dicho número al **acumulador**. Finalmente, se retorna la suma total para ser impresa en pantalla.

2.- Implementación multihilo.

Contrario a la implementación anterior, esta versión se diferencia por aprovechar todos los hilos disponibles del procesador en la computadora que ejecuta el programa. Para lograrlo, se utiliza la función hardware_concurrency(), la cual retorna la cantidad de núcleos de hardware disponibles para la ejecución concurrente, permitiendo así distribuir el trabajo entre múltiples hilos y mejorar el rendimiento.

Lo característico de esta implementación es que distribuye el trabajo entre **múltiples** hilos, asignando a cada uno un rango específico de valores sobre los cuales debe verificar qué números son **primos** y **sumarlos**. Para ello, se utilizan las funciones **isPrime()** y **sum()**, esta última modificada para recibir una posición **inicial** y **final** que define el segmento del arreglo (con números del 1 al 5 millones) que cada **hilo** debe **procesar**.

Una vez se llama a todos los hilos, se utiliza la función **thread::join()** para asegurarse de que cada hilo complete su tarea antes de continuar con la ejecución del programa. Esto garantiza que los **resultados parciales** acumulados por cada hilo estén listos para ser sumados, obteniendo así el resultado total de la suma de todos los números primos menores a 5 millones.

3.- Implementación con CUDA.

Para la implementación con CUDA se optó por utilizar **512** hilos por bloque y **32** bloques, lo que da un total de 16,384 hilos ejecutándose en la GPU. En cuanto a la lógica del programa, primero se define en el CPU un array con los números del 1 a 5 millones, así como otro array del tamaño de la cantidad de bloques. Posteriormente, mediante la función **cudaMalloc()**, se reserva memoria en la GPU para ambos arrays. Luego, se copia el arreglo de números desde la CPU hacia la GPU utilizando **cudaMemcpy()** con el parámetro **"cudaMemcpyHostToDevice"**.

Se mantuvieron **isPrime()** y **sum()**, con algunas modificaciones. La función **isPrime()** se definió como **__device__** para que pudiera ser ejecutada desde la **GPU** por cada hilo. Por otro lado, **sum()** se definió como **__global__**, lo que permite llamarla desde el **CPU** al ser la función principal que ejecuta el cálculo en la **GPU**. Dentro de **sum()**, una vez que cada hilo termina, almacena su valor acumulado en una memoria compartida **(__shared__)**. Luego, usando **__syncthreads()**, se sincronizan todos los hilos del bloque para evitar caer en una condición de carrera. Finalmente, los resultados parciales por bloque se almacenan en el arreglo **results**.

Cabe destacar que, en este punto, los resultados parciales aún se encuentran almacenados en la memoria de la **GPU** y no son accesibles directamente. Por ello, se utiliza **cudaMemcpy()** con el parámetro **cudaMemcpyDeviceToHost** para copiar el arreglo de resultados desde la **GPU** hacia la memoria del **CPU**. Una vez en la **CPU**, se recorre el arreglo con un ciclo para sumar todos los resultados parciales y así obtener el resultado final de la suma de los números primos.

Medición de rendimiento.

Para el cálculo del tiempo que tardó cada programa se utilizó la librería de **chrono**, y lo más importante es que a cada programa se le puso dentro de un ciclo que repite **10 veces** el proceso de identificar los números primos y sumarlos, esto para obtener un **promedio** del tiempo que tarda en ejecutarse, siguiendo esta regla los tiempos de ejecución fueron los siguientes.

	Secuencial	Hilos	Paralelo
Tarea	927.752	130.734	0.018

Fig. 1.1. Medición del tiempo en milisegundos para cada implementación.

Posteriormente para obtener un mejor criterio sobre cual implementación fue la más óptima se utilizaron fórmulas para calcular el speedup y la eficiencia.

$$S_p = \frac{T_1}{T_p} \qquad E = \frac{S_p}{h}$$

S_p Speedup obtenido usando p procesadores.

E Porcentaje que se está aprovechando.

T_1 Tiempo que tarda la ejecución secuencial.
T_p Tiempo que tarda la ejecución paralela.

Fig. 1.2. Fórmula del speedup.

S_p Speedup obtenido.
h número de hilos con los que se contó durante la ejecución.

Fig. 1.3. Fórmula de la eficiencia.

Haciendo uso de estas fórmulas se determinó la eficiencia y speedup para la implementación con CUDA y multihilo.

	Speed Up	Eficiencia	Num. hilos
Tarea	7.096485994	44.35%	16

Fig. 1.4. Speedup y eficiencia de la solución multihilo.

	Speed Up	Eficiencia	Num. hilos
Tarea	51541.77778	314.59%	16384

Fig. 1.5. Speedup y eficiencia de la solución con CUDA.

Comparación de Desempeño.

Desde que vemos cual fue el tiempo promedio de ejecución de CUDA se podría intuir que sería mucho mejor en comparación con el multihilo, ya que usando CUDA se obtuvo un tiempo de ejecución **7265 veces** mejor y tomando como referencia el speedup y la eficiencia obtenidas en cada una de las implementaciones, CUDA obtuvo un speedup **7263 veces** mejor que la forma multihilo y una eficiencia **7 veces** mayor.

En conclusión la implementación con **CUDA** demostró un desempeño claramente superior frente a la solución multihilo en CPU, con resultados tan increíbles como en el tiempo de ejecución donde la versión multihilo requirió, en promedio, 130,734 ms, en cambio **CUDA** apenas tardó **0,018 ms**, lo que supone una mejora de más de **7 000 veces** y también la mejora en la eficiencia que logró **CUDA** obteniendo **7 veces** más de la eficiencia obtenida con multihilo.

Estos resultados se explican principalmente debido a la gran cantidad de hilos con los que se cuenta en **CUDA**, ya que a comparación de los núcleos del CPU que comparados uno a uno tienen más potencia bruta, cuando comparas la gran cantidad que posee el **GPU** terminan ganando en potencia y esto fue confirmado al observar la mejora aplastante que se consiguió con la solución en **CUDA**, por lo tanto para un problema que era muy paralelizable resulta mucho mejor realizar la implementación con **CUDA** si se desea aprovechar mejor los recursos con los que se cuenta.

Fuentes bibliográficas:

Parzibyte. (2024, 19 julio). *Redondear números en C*. Parzibyte's Blog. https://parzibyte.me/blog/2018/11/08/redondear-numeros-en-c/

Sherief, A. (2021, 26 enero). An overview of CUDA, part 2: Host and device code. DEV Community. https://dev.to/zenulabidin/an-overview-of-cuda-part-2-host-and-device-code-69d

Harris, M., & Harris, M. (2022, 21 agosto). Using Shared Memory in CUDA C/C++. *NVIDIA Technical Blog*. https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

OpenAI. (2025). ChatGPT [Modelo de lenguaje de gran tamaño]. https://chat.openai.com/chat

std::thread::hardware_concurrency - cppreference.com. (s. f.). https://en.cppreference.com/w/cpp/thread/thread/hardware_concurrency.html