

# Programación Paralela

## Teórica 01 - Introducción

Segundo Semestre, 2025

---

### Introducción

Los microprocesadores se basan en una unidad central de procesamiento (CPU) que ejecuta un cierto número de *threads* (hilos) en paralelo. Estas CPUs se fueron llevando a límites de rendimiento donde gracias a la miniaturización de los componentes, la mayor cantidad de núcleos, la mayor velocidad, mejores formas de enfriamiento y la mejora en la eficiencia energética, se logró aumentar la cantidad de procesamiento que se podía hacer en un solo chip.

Históricamente, la mayor parte de las aplicaciones se vieron beneficiadas de estos avances de hardware para incrementar la velocidad de las propias aplicaciones donde, esencialmente, el mismo software funcionaba más rápido a medida que se iban realizando mejoras en estas unidades de procesamiento secuenciales. Sin embargo esto muchas veces se lo conoce como escalabilidad vertical, donde se busca mejorar el rendimiento simplemente agregando más recursos. El problema con este enfoque es que eventualmente se llega a un límite de la cantidad de recursos que se pueden agregar.

Para ilustrar los conceptos básicos de la programación paralela y escalable, necesitamos elegir un lenguaje simple de programación que soporte paralelismo masivo. En este curso vamos a utilizar CUDA C para nuestros ejemplos y ejercicios. CUDA C extiende el lenguaje de programación C con una nueva sintaxis e interfaces que hace que los programadores puedan trabajar con sistemas que tengan tanto CPUs como GPUs masivamente paralelas.

Si bien la escalabilidad vertical seguirá siendo importante y la cantidad de núcleos posiblemente siga aumentando mejorando el paralelismo de las CPUs, hay aplicaciones que podrían ver un incremento de *performance* (rendimiento) mucho mayor si se utilizaran programas que se ejecutan en paralelo. Si bien esta idea no es nueva [1], estos programas paralelos estaban limitados a supercomputadoras y clusters de computadoras.

El *ratio* de rendimiento entre una CPU y una GPU puede ser de 1:10 se debe a que las CPUs están optimizadas para ejecutar código secuencial de forma *performante* realizando un paralelismo interno de instrucciones, pero dando la impresión de una ejecución secuencial. Poseen internamente grandes cachés que les permiten manejar la típica latencia del acceso a dispositivos externos lentos. Las GPUs por el otro lado tienen cachés mucho más pequeñas, pero están optimizadas para mover grandes cantidades de datos tanto *in* como *out* de la DRAM (*Dynamic Random Access Memory*) de la GPU porque fueron creadas para la industria de los juegos donde el *frame buffering* es un requerimiento crítico.

### ¿Por qué no se utilizan GPUs para todo?

Las GPUs están diseñadas para la ejecución paralela y de alto *throughput* de operaciones matemáticas. Sin embargo, las CPUs son mucho mejores para realizar tareas de baja latencia donde sobrepasan holgadamente a la *performance* de una GPU. Por eso es que las computadoras de escritorio traen CPUs ya que muchas de las tareas de interacción con los usuarios tienen que ser de baja latencia.

### ¿Qué es CUDA?

CUDA es un modelo de programación desarrollado por NVIDIA que permite la utilización de ambas unidades de procesamiento (CPU y GPU) a la vez para ejecutar una aplicación. Con esto entonces podemos aprovechar la potencia de cómputo de varias unidades de procesamiento a la vez.

Sin embargo no hay que ser ingenuos y pensar que la única razón por la que se desarrolla esto es para aumentar la velocidad. En principio el factor económico es uno de los factores más importantes, ya que lo que se desea es utilizar la base instalada de procesadores, ya que el costo del desarrollo de software sólo se ve justificado por una población grande de clientes. Este había sido un problema grande ya que la creación de supercomputadoras masivamente paralelas, sólo había estado al alcance de pocos. Pero dado que el mercado de GPUs es enorme casi que podemos decir que prácticamente todas las PCs tienen algún tipo de GPUs, lo cual hace que existan más de 1000 millones de GPUs en el mundo que pueden ser utilizadas con CUDA.

## ¿Por qué paralelizar?

Hay aplicaciones que trabajan con datos que pueden ser procesados en paralelo que pueden ser masivamente paralelizadas ejecutando en varias GPUs. Por ejemplo en biología donde a veces las observaciones están limitadas por las observaciones que se pueden hacer a nivel molecular, es posible crear modelos que simulen las actividades subyacentes de estas moléculas a través de modelos computacionales.

Sin embargo no todas las aplicaciones pueden ser paralelizadas, sino que normalmente un porcentaje del tiempo de ejecución de un aplicación puede ser paralelizado. Sin embargo la ley de Amdahl nos dice que el rendimiento obtenida por la optimización de una parte de un programa está limitada por la cantidad de tiempo que esa parte está en uso. Con lo cual, cuantas más secciones de un software podamos paralelizar, más rápido podremos lograr que se ejecuten, llegando a mejoras de  $100\times$  o más en el rendimiento de un programa.

Hay que tener en cuenta, de todas formas, que la CPU es muy eficiente para correr aplicaciones secuenciales, con lo cual la GPU es un complemento de la CPU y no un reemplazo.

## Desafíos en la programación paralela

Se podría pensar entonces que la programación paralela es una solución a todos los problemas de rendimiento.

Sin embargo, por un lado es complicado diseñar algoritmos paralelos que tengan el mismo nivel de complejidad computacional que los algoritmos secuenciales. Por el otro lado la programación paralela tiene límites de velocidad de acceso a memoria. La performance de los algoritmos paralelos son muy sensibles a los datos de entrada, ya que esencialmente la paralelización se basa en el procesamiento de datos de manera paralela. Por último, pero no menos importante, los algoritmos paralelos suelen expresarse de manera de recurrencias, lo cual requiere que se piensen los diferentes problemas de formas no intuitivas.

## Modelos y Lenguajes de Programación Paralela

En el pasado hubo muchos lenguajes de programación que fueron propuestos para abordar el problema de la programación paralela. Algunos de estos lenguajes son:

- **OpenMP**: es una API de programación en paralelo que se basa en directivas de compilador y funciones de biblioteca para permitir la paralelización de aplicaciones en sistemas de memoria compartida.
- **MPI**: es una biblioteca de paso de mensajes que permite la comunicación entre procesos en un sistema distribuido.
- **OpenACC**: Es un estándar de programación para la computación paralela desarrollado por Cray, CAPS, Nvidia y PGI. El estándar está diseñado para simplificar la programación paralela de sistemas heterogéneos CPU/GPU.
- **OpenCL**: Es un estándar abierto para la programación de sistemas heterogéneos que consta de CPUs, GPUs y otros dispositivos de cómputo.
- **CUDA**: Es una plataforma de computación paralela y un modelo de programación desarrollado por NVIDIA para sus GPUs.

## Bibliografía

## References

- [1] H. Sutter and J. Larus, "Software and the concurrency revolution: Leveraging the full power of multicore processors demands new tools and new thinking from the software industry.," *Queue*, vol. 3, no. 7, pp. 54–62, Sep. 2005, ISSN: 1542-7730. DOI: 10.1145/1095408.1095421. [Online]. Available: <https://doi.org/10.1145/1095408.1095421>.