Programación con GPUs

Teórica 03 - Introducción a la Programación Paralela

Segundo Semestre, 2025

Contents

3.1 Introducción

Hasta ahora hemos visto cómo escribir en C para resolver problemas algorítmicos utilizando la CPU de una computadora. Sin embargo, el objetivo de la materia es la de aprender a escribir programas que puedan ejecutarse en múltiples unidades de procesamiento en forma **paralela**.

Los microprocesadores que tenemos en nuestras computadoras personales y celulares se basan en una unidad central de procesamiento (CPU) que ejecuta un cierto número de *threads* (hilos) en paralelo, que ejecutan un código secuencial de instrucciones. A lo largo de la historia, estas CPUs se fueron llevando a límites de rendimiento cada vez mayores, donde gracias a la miniaturización de los componentes, la mayor cantidad de núcleos, la mayor velocidad del reloj, 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.

Esto produjo que, la mayor parte de las aplicaciones, se vieran beneficiadas de estos avances de hardware para incrementar la velocidad de las propias aplicaciones donde, esencialmente, el mismo software funcionaba mucho 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 de la escalabilidad vertical es que tiene un límite, ya que eventualmente llegaremos a un punto donde no se pueden acelerar más las unidades de procesamiento y debemos encontrar otras formas de optimización. Es aquí donde entra en juego la **escalabilidad horizontal** que se refiere a la capacidad de aumentar el rendimiento de un sistema agregando más unidades de procesamiento (CPUs o GPUs) al sistema en lugar de hacer las unidades más veloces.

Para ilustrar los conceptos básicos de la programación paralela y escalable, necesitamos programar en un lenguaje que soporte la paralelización masiva como C, C++ o Python. Para ello elegimos uno de los modelos de programación paralela más populares que es CUDA (*Compute Unified Device Architecture*) que es una extensión del lenguaje C para ilustrar los conceptos básicos de la programación paralela. El modelo de programación CUDA fue desarrollado por NVIDIA que permite aprovechar la potencia de cómputo de varias unidades (GPU) de procesamiento a la vez.

La idea de parlelizar no es nueva, pero históricamente los centros de procesamiento paralelo estaban limitados a supercomputadoras y clusters bajo la órbita de gobiernos y grandes empresas que podían costearlos (Sutter and Larus [1]). Esta paralelización NO implica necesariamente un reemplazo de las CPUs sino un complemento ya que por un lado las CPUs son buenas para tareas de baja latencia y su poder de procesamiento seguramente seguirá aumentando.

El *ratio* de rendimiento entre una CPU y una GPU puede ser de 1:10 (o más) para ciertas operaciones. Las CPUs están optimizadas para ejecutar código secuencial de forma *performante* relizando, en la medida de lo posible, un paralelismo interno de instrucciones que es transparente para el usuario. Las CPUs, además, poseen internamente grandes cachés que les permiten manejar la inherente latencia del acceso a dispositivos externos lentos, mientras que las GPUs por el otro lado tienen cachés mucho más pequeñas ya que sólo necesitan acceder a memoria (muy rápida) y tener una gran optimización para mover grandes cantidades de datos tanto *in* como *out* de la DRAM (*Dynamic Random Access Memory*) ya que originalmente fueron diseñadas para procesar gráficos en tiempo real, especialmente en la industria de los videojuegos donde el *frame buffering* es un requerimiento crítico.

3.2 Complejidad algorítmica

En el repaso de algoritmos (y materias anteriores esta) vieron sólo algoritmos secuenciales. Sin embargo antes de avanzar con la programación paralela, tenemos que estudiar *complejidad algorítmica* más en detalle.

La complejidad algorítmica es un parámetro con el cual podemos medir la eficiencia de un algoritmo en términos de la cantidad de recursos que va a utilizar para completar la tarea. Por un lado tenemos la complejidad en **tiempo**, que podemos pensarla como la cantidad de ciclos de CPU que necesita un algoritmo para obtener el resultado esperado, y por el otro lado, tenemos la complejidad en **espacio**, que podemos pensarlo como cantidad de memoria que necesita un algoritmo para obtener el resultado esperado.

Ambas complejidades se expresan en función del tamaño del *input* del algoritmo y nos permiten comparar diferentes algoritmos independientemente del lenguaje en que estén implementados. Esto significa que si tenemos dos lenguajes de programación diferentes, la complejidad tanto en **tiempo** como en **espacio** de un mismo algoritmo será la misma, aunque el tiempo de ejecución y la cantidad de memoria pueden ser diferentes ya que los lenguajes de programación tienen diferentes niveles de optimización.

3.2.1 Complejidad en Tiempo

Cuando diseñamos un algoritmo, es importante tener en cuenta la cantidad de recursos que se van a utilizar, tratando de estimar cuál será su complejidad en tiempo y en espacio de acuerdo a la solución que hayamos propuesto. Hay diferentes métricas de estimación, aunque las más comunes son estimar **el caso promedio** y, aún más importante, **el peor caso posible**. Por ejemplo, agregar un elemento al final de un *array* dinámico en cualquier lenguaje de programación de alto nivel, tiene una **complejidad promedio** constante, ya que cuando se crea un array dinámico, el lenguaje lo crea de un tamaño dado (aún si el usuario no lo especifica). Por el otro lado, si el *array* dinámico estuviera lleno, estaríamos en el *peor caso* posible, ya que el lenguaje, tendría que conseguir más memoria para añadir ese nuevo elemento y luego copiar todos los elementos a la nueva posición de memoria.

En esta materia vamos a ver sólo la notación de complejidad para el peor caso, que se representa con la notación **Big O**. En palabras simples, lo que representa la complejidad algorítmica en el peor caso es: *la cantidad máxima de tiempo de CPU o cantidad máxima de memoria que se requiere para resolver un problema en función del tamaño de la entrada.* Esta función **Big O** nos da una cota asintótica superior de la cantidad de recursos que se necesitan para resolver un problema en función del tamaño de la entrada.

En la figura 1, podemos observar diferentes funciones de complejidad algorítmica en función del tamaño de la entrada (`n`).

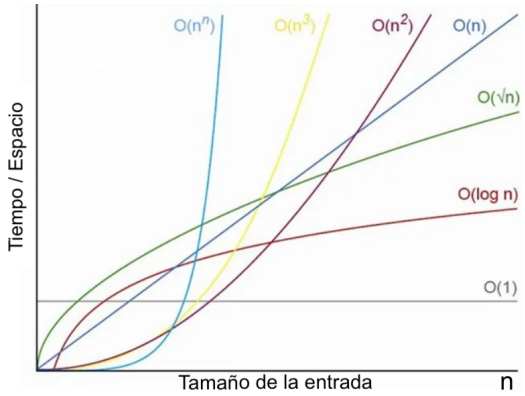

Figure 1: Big O - Comparación de complejidades

Diferentes implementaciones de un mismo algoritmo para resolver un problema pueden tener diferentes complejidades. Por ejemplo, existen varios algoritmos que resuelven el problema de ordenar un *array* (arreglo) de elementos de tamaño n. El más popular por su facilidad de implementación es el algoritmo de burbujeo que, si bien ordena los elementos, tiene una complejidad de $O(n^2)$, pero si utilizamos un algoritmo de ordenamiento más eficiente, nos encontramos con el *merge sort* que tiene una complejidad de $O(n \cdot \log\ n)$. Ambos algoritmos resuelven el mismo problema (ordernar un array), pero uno es mucho más eficiente que el otro. En la figura 2 podemos ver la comparación de ambas complejidades.

Descubrir un algoritmo más eficiente que otro no fácil.

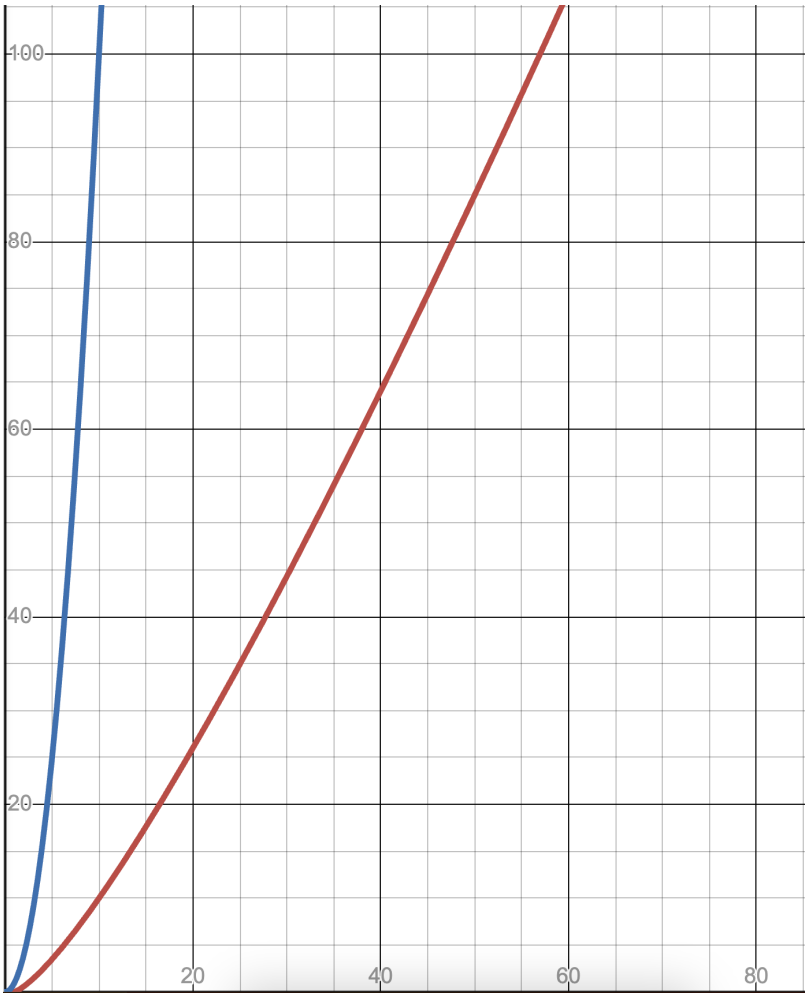

Figure 2: Big O - Comparación de complejidad algorítmica para Bubble Sort vs Quick Sort

3.2.2 Definición: Big O

En todos los gráficos que vimos la variable `n` es la cantidad de elementos que tenemos como *input* de nuestro algoritmo que se ejecutará con una cierta función de tiempo $f(n)$.

Definición: Diremos que el algoritmo se comporta con una complejidad asintótica de $O(g(n))$ si existe una constante C y un valor n_0 tal que el valor absoluto de $f(n)$ es menor o igual a C por el valor absoluto de $g(n)$ para todo n mayor a n_0. (Wilf [2]). Formalmente:

$$f(n) = O(g(n))\ (n \ \rightarrow \ \infty)\ si\ \exists\ C, n_0\ /\ |f(n)| \leq C\,|g(n)|\ (\forall\ n > n_0)$$

Las complejidades algorítmicas nos dan una idea de si un algoritmo puede ser realizable o no por una determinada unidad de procesamiento; por *realizable* queremos decir que la capacidad de resolver las instancias deseadas de un problema estén dentro de nuestros recursos disponibles. En la práctica, la factibilidad es muy dependiente del contexto y no es particularmente *portable* (portátil) entre diferentes problemas y situaciones. Un principio común se mantiene en casi todas las situaciones: **una tasa de crecimiento exponencial en el consumo de algún recurso limita la aplicación del uso de ese método a todas las instancias, excepto a las más pequeñas**. En otras palabras, si nuestros algoritmos tienden a tener complejidades

exponenciales (o más) en el tamaño de la entrada, no importa cuántos recursos tengamos, no podremos resolver instancias grandes del problema.

Por lo tanto, la factibilidad ha llegado a significar que la tasa de crecimiento del recurso está acotada por un una complejidad polinomial en el tamaño de la entrada. Esto nos da la noción común de que un problema tiene una solución secuencial factible sólo si tenemos un algoritmo de tiempo polinomial, es decir, sólo si cualquier instancia de tamaño n del problema se puede resolver en tiempo $n^{O(1)}$. Aunque ampliamente reconocido como muy simplista, la dicotomía entre algoritmos polinomiales y no polinomiales ha demostrado ser un discriminador poderoso entre aquellos cálculos que son factibles en la práctica y aquellos que no lo son (Greenlaw, Hoover, and Ruzzo [3]).

Lógicamente hay problemas para los cuales no se han encontrado algoritmos polinomiales, por ejemplo, la factorización entera de un número muy grande es un problema que no puede resolverse en tiempo polinomial con las CPUs y GPUs actuales. Por eso es que la seguridad de muchos sistemas de encriptación se basan en ello. Si bien excede lárgamente el contenido de esta materia, hay algoritmos cuánticos que pueden resolver este problema en tiempo $O((\log n)^{O(1)})$ ([4]).

3.2.3 Complejidad en Espacio

La complejidad en espacio es la cantidad de memoria requerida para resolver un algoritmo. Es, también, una función que depende de la entrada del algoritmo y de la cantidad de memoria adicional que se necesite para resolver el problema.

Como en la complejidad en tiempo, se utiliza la notación **Big O** ya que en este caso también se trata de una cota asintótica superior de la cantidad de memoria que se necesita para resolver un problema en función del tamaño de la entrada. Esto incluye tanto la memoria utilizada estáticamente como la memoria utilizada dinámicamente, sin embargo, no se tiene en cuenta la memoria utilizada como *input*. Por ejemplo, un algoritmo que cuenta la cantidad de números pares de un vector tiene una complejidad en tiempo de $O(n)$ y una complejidad en espacio de $O(1)$ ya que sólo necesita una variable para contar la cantidad de números pares.

Algorithm 1 Contar números pares

Define: countPares(a)
Input: $a = \{a_0, a_1, a_2, \ldots, a_n\}$
Initialization: $count = 0$
1: **Output:** $count$
2: **for** $i = 0$ to n **do**
3: **if** $a_i \mod 2 = 0$ **then**
4: $count = count + 1$

Vemos aquí que la complejidad en tiempo es $O(n)$ ya que sólo hay un ciclo que recorre todos los elementos del vector de tamaño n pero la complejidad en espacio es $O(1)$ ya que sólo utilizamos una variable `count` para guardar el contador con la cantidad de números pares. Esto significa que si el tamaño del vector a se duplica, la cantidad de memoria utilizada no cambia, pero la cantidad de ciclos de CPU necesarios para completar la tarea se duplica.

3.3 ¿Por qué no se utilizan GPUs para todo?

En la 3.1 se mencionó que el *ratio* de rendimiento entre una CPU y una GPU puede ser de 1:10 (e incluso mayor), entonces una pregunta válida sería **¿por qué no se utilizan GPUs para todo?**. La respuesta a esto no es única, por un lado las GPUs están diseñadas para tener alto *throughput* de operaciones matemáticas y de transferencia a memoria y suelen ser buenas para tareas de alta latencia que requieren pasajes grandes de información desde y hacia la memoria (óptimos para el procesamiento paralelo). Por el otro lado las CPUs están optimizadas para realizar tareas de baja latencia y tareas inherentemente secuenciales donde sorbepasan holgadamente a la *performance* de una GPU. Por esto es que estos sistemas se llaman híbridos, ya que hay un hilo conductor secuencial y otros que se ejecutan en paralelo (a demanda).

Un ejemplo típico de una tarea que es inherentemente secuencial es la ejecución de un sistema operativo, ya que todas las tareas que se ejecutan en un sistema operativo son secuenciales y dependen unas de otras. Otroe ejemplo es la orquestación GPU-CPU, ya que la CPU es la que orquesta las tareas que se ejecutan en la GPU y no al revés. Además la CPU es buenísima para tareas de baja latencia como la ejecución del software de una computadora personal.

Por el otro lado, las GPUs son muy buenas para tareas que pueden ser paralelizadas y que requieren un alto *throughput* de operaciones matemáticas y de transferencia a memoria. Un ejemplo típico de una tarea que puede ser paralelizada es el procesamiento de imágenes, ya que cada píxel de una imagen puede ser procesado de manera independiente. Otro ejemplo es el entrenamiento de un modelo de machine learning implica realizar millones o incluso miles de millones de operaciones matriciales (multiplicaciones de matrices) y cálculos matemáticos para ajustar los parámetros del modelo.

Al desarrollar CUDA C, NVIDIA no sólo ha permitido paralelizar aplicaicones, sino que también ha permitido bajar el costo del desarrollo del software paralelo que estaba restringido a supercomputadoras masivamente paralelas. Esto se debe a que el mercado actual de GPUs es enorme ya que casi todas las PCs actuales tienen algún tipo de GPU instalada, habiendo más de 1000 millones de GPUs en el mundo que pueden ser utilizadas con CUDA, NVIDIA facilitó el desarrollo de aplicaciones paralelas, logrando que el desarrollo de software paralelo sea más accesible para todos.

3.4 Ley de Amdahl

Como veremos en la sección siguiente, no todas las aplicaciones pueden ser paralelizadas, y en muchas aplicaciones, sólo un porcentage del tiempo de ejecución de una aplicación puede ser paralelizado. El algoritmo utilizado es quien define la mejora de velocidad y no necesariamente el número de procesadores; en algún momento llegaremos a un límite donde finalmente por más que tengamos más recursos no se podrá paralelizar más el algoritmo. A esto se lo conoce como ley de Amdahl.

La ley de Amdahl es una fórmula aritmética simple que se utiliza para estimar la potencial mejora de velocidad que puede obtenerse al paralelizar una parte del código de un programa secuencial en múltiples procesadores. La idea de la fórmula es que hay una parte que puede ser paralelizada y otra parte que no puede ser paralelizada, y la mejora de velocidad se calcula en función del porcentaje de código que se puede paralelizar y el número de procesadores utilizados. [5]

La ley de Amdahl se expresa matemáticamente como:

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

donde:

- S es la velocidad de mejora del programa.
- P es la fracción del programa que se puede paralelizar (entre 0 y 1).
- N es el número de procesadores utilizados.

Es bastante sencillo ver que si la parte paralelizable es 0% (es decir, $P = 0$), no existe mejora de velocidad ($S = 1$). Por otro lado, si la parte paralelizable es 100% (es decir, $P = 1$), la mejora de velocidad es igual al número de procesadores utilizados ($S = N$).

Supongamos que $P = 0.5$ (es decir, el 50% del programa puede ser paralelizado) y que utilizamos $N = 2$ (es decir, utilizamos dos procesadores). En este caso, la mejora de velocidad sería:

$$S = \frac{1}{(1-0.5) + \frac{0.5}{2}} = \frac{1}{0.5 + 0.25} = \frac{1}{0.75} \approx 1.33$$

3.5 Desafíos en la programación paralela

Podríamos pensar entonces que la programación paralela puede ser una solución general para resolver todos los problemas de *performance* y debería utilizarse siempre. Aunque como vimos con la ley de Amdahl, no todos los problemas son 100% paralelizables, y en general sólo podremos intentar paralelizar una parte de la aplicación, mientras que otra parte deberá seguir siendo secuencial.

El primer problema, es que es difícil diseñar algoritmos paralelos, ya que requiere pensar los problemas de formas anti-intuitivas que, muchas veces, no son sencillas de resolver para que tengan el mismo nivel de complejidad computacional que los algoritmos secuenciales. 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 limitada por la velocidad de acceso a memoria. Pero aún si salváramos estos problemas que son, méramente técnicos, nos encontraríamos con la pregunta más profunda: **¿todos los problemas son teóricamente paralelizables?**.

3.6 Los límites de la programación paralela

En la teoría de complejidad computacional, P es una clase [1] de complejidad que contiene todos los problemas de decisión que pueden ser resueltos por una máquina de Turing determinista en tiempo polinomial. Sin embargo, Nicholas Pippenger realizó una investigación exhaustiva sobre circuitos con profundidad polilogarítmica y tamaño polinomial y sugirió una interesante clase de complejidad a estudiar que sería la clase de problemas solucionables por máquinas paralelas en tiempo polilogarítmico [2] usando un número polinomial de procesadores. Esta clase se conoce como **NC** (por *Nick's class*). (Stockmeyer [6])

Por otro lado los problemas $P - completos$ son de interés porque parecen carecer de soluciones altamente paralelas. Es decir, los diseñadores de algoritmos han fallado en encontrar algoritmos NC para cualquier problema $P - completo$. En consecuencia, la promesa de la computación paralela, es decir, que aplicar más procesadores a un problema parece ser imposible en toda la clase de problemas $P - completos$. Dejando abierta la siguiente pregunta: (Greenlaw, Hoover, and Ruzzo [3])

$$P \stackrel{?}{=} NC$$

Esta clase P se puede pensar como los problemas tratables (tesis de Cobham), por lo que NC se puede pensar como los problemas que se pueden resolver eficientemente en una computadora paralela. NC es un subconjunto de P porque los cálculos paralelos polilogarítmicos se pueden simular mediante cálculos secuenciales polinomiales. Pero no se sabe si $NC \stackrel{?}{=} P$, pero **la mayoría de los investigadores sospechan que esto es falso**, lo que significa que probablemente hay algunos problemas tratables que son **"intrínsecamente secuenciales"** y no se pueden acelerar significativamente utilizando paralelismo. Así como la clase NP-completo se puede pensar como "probablemente intratable", así que **la clase P-completo, cuando se utilizan reducciones NC, se puede pensar como "probablemente no paralelizable" o "probablemente *intrínsecamente secuencial*"**. Pero por más que parezca extraño, no hay ninguna demostración formal de que $P \neq NC$ o que $P = NC$.

Con lo cual, la respuesta a la pregunta de si todos los problemas son paralelizables es que **NO, no todos los problemas son paralelizables ya que hay ciertos problemas que son intrínsecamente secuenciales** (o no se ha encontrado una forma de paralelizarlos).

3.7 Modelos y Lenguajes de Programación Paralela

Hay varios lenguajes de programación paralela propuestos para abordar el problema de la programación paralela, aunque aquí sólo veremos CUDA, aquí tienen una lista de algunos de los lenguajes más populares:

- **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.

[1] En matemáticas, una clase es una colección de conjuntos (u otros objetos matemáticos) que pueden ser definidos unívocamente por una propiedad compartida por todos los miembros de ese conjunto

[2] Un problema es polilogarítmico si su complejidad es $O((\log n)^k)$ para algún k

- **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.

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.

[2] H. S. Wilf, *Algorithms and complexity*. AK Peters/CRC Press, 2002.

[3] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo, *Limits to parallel computation: P-completeness theory*. Oxford university press, 1995.

[4] A. Ekert and R. Jozsa, "Quantum computation and shor's factoring algorithm," *Reviews of Modern Physics*, vol. 68, no. 3, p. 733, 1996.

[5] B. Tuomanen, *Hands-On GPU Programming with Python and CUDA: Explore high-performance parallel computing with CUDA*. Packt Publishing Ltd, 2018, ISBN: 978-1-78899-391-3.

[6] L. Stockmeyer, "Classifying the computational complexity of problems," *The Journal of Symbolic Logic*, vol. 52, no. 1, pp. 1–43, 1987. DOI: 10.2307/2273858.

