por **Federico Gonzalez Brizzio**

Introducción a la programación paralela

Federico Gonzalez Brizzio

Tabla de contenidos

1 Presentación **10**
1.1 Objetivos del libro 10
1.2 Organización del recorrido 11
1.3 Cómo trabajar con este libro 12
1.4 Herramientas mínimas para empezar 12
1.5 Sobre el autor 13
1.6 Ilustraciones y revisión 13
1.7 Reporte de errores y sugerencias 14

I Fundamentos del paralelismo **15**

2 Conceptos básicos **16**
2.1 Objetivos del capítulo 16
2.2 Por qué estudiar paralelismo 17
2.3 Computación secuencial 17
2.4 Velocidad de ejecución y ciclo del procesador 19
2.5 Los límites de la computación secuencial 21
2.6 Concurrencia, paralelismo y distribución 21
2.7 Niveles de paralelismo 22
2.8 Observación del uso de CPU 23
2.9 Cierre de la unidad 25
2.10 Ejercicios del capítulo 25

3 Arquitectura y métricas **27**
3.1 Objetivos del capítulo 27
3.2 Clasificación por mecanismo de control 27
3.3 Clasificación por organización física 29

3.4 Jerarquía de memoria y localidad . . . 30
3.5 False sharing y NUMA . . . 32
3.6 Speed-up . . . 33
3.7 Eficiencia . . . 34
3.8 Un ejemplo numérico de escalamiento . . . 35
3.9 Escalamiento fuerte y escalamiento débil . . . 35
3.10 Ley de Amdahl . . . 36
3.11 Ley de Gustafson . . . 36
3.12 Memory bound y compute bound . . . 37
3.13 Modelo Roofline . . . 38
3.14 Cierre de la unidad . . . 38
3.15 Ejercicios del capítulo . . . 39

4 Modelos de programación paralela **40**
4.1 Objetivos del capítulo . . . 40
4.2 Por qué trabajar con modelos . . . 40
4.3 Modelo coordinador-trabajadores . . . 41
4.4 Divide and conquer . . . 43
4.5 Pipelining . . . 45
4.6 MapReduce . . . 46
4.7 Modelo de actores . . . 47
4.8 Comparación conceptual entre modelos . . . 48
4.9 Cierre de la unidad . . . 49
4.10 Ejercicios del capítulo . . . 50

5 APIs de programación paralela **51**
5.1 Objetivos del capítulo . . . 51
5.2 Por qué estudiar estas APIs . . . 51
5.3 Pthreads . . . 52
5.4 OpenMP . . . 53
5.5 MPI . . . 55
5.6 Diferencias y ámbitos de aplicación . . . 57
5.7 Vista comparativa . . . 57
5.8 Multiplicación de matrices como caso de estudio . . . 58
5.9 Cierre del capítulo . . . 60

5.10 Ejercicios del capítulo 60

II Herramientas y aplicaciones **62**

6 Paralelismo en Python **63**
6.1 Objetivos del capítulo 63
6.2 Qué permite comparar Python 64
6.3 Herramientas centrales de este capítulo 64
6.4 Limitaciones de Python para paralelismo 65
6.5 Threads y procesos: qué cambia en Python y qué cambia respecto de C 66
6.6 Patrones básicos para paralelizar loops en Python 67
6.6.1 Creación manual de workers 67
6.6.2 Pools y executors 68
6.7 Secuencial, procesos y Numba sobre problemas clásicos 69
6.8 Un caso práctico transversal: Sobel en CPU 74
6.9 Cómo elegir una estrategia 77
6.10 Errores frecuentes al paralelizar en Python 78
6.11 Una mirada inicial al debugging y profiling 79
6.12 Una tabla de síntesis del capítulo 79
6.13 Ejercicios del capítulo 80

7 Vectorización, broadcasting y optimización sobre arreglos **82**
7.1 Objetivos del capítulo 82
7.2 Planteo del capítulo 83
7.3 Qué es la vectorización 83
7.4 SIMD como fundamento 85
7.5 Qué es el broadcasting 85
7.6 Relación entre vectorización y broadcasting 87
7.7 El lugar de NumPy 87
7.8 Reformular también es optimizar 88
7.9 Un ejemplo de localidad: matrices y transposición 88
7.10 Broadcasting como técnica puntual de reformulación 91
7.11 Tensores en CPU: continuidad con PyTorch 92
7.12 Un caso práctico transversal: Sobel con arreglos y tensores 93

7.13 Criterios para el análisis práctico 96
7.14 Una tabla de síntesis del capítulo 97
7.15 Ejercicios del capítulo 98

8 Computación con GPU **100**
8.1 Objetivos del capítulo 100
8.2 El lugar de las GPU en este recorrido 101
8.3 Qué caracteriza a una GPU 101
8.4 CUDA como modelo de programación 103
8.5 Alternativas a CUDA 104
8.6 Un kernel mínimo con Numba 105
8.7 Un ejemplo bidimensional: multiplicación de matrices 108
8.8 Tipos de memoria en GPU 109
8.9 Coalescing 110
8.10 Warp divergence 110
8.11 Occupancy 110
8.12 Configuración del kernel y rendimiento observado 111
8.13 PyTorch sobre GPU 111
8.14 Paralelismo implícito en CPU 112
8.15 Dispositivo, transferencias y costo total 113
8.16 Ejecución asíncrona y sincronización 115
8.17 Operaciones por lotes y paralelismo de datos 116
8.18 Experimentos prácticos con PyTorch 117
8.19 Numba CUDA y PyTorch GPU 119
8.20 Continuidad del caso práctico transversal: Sobel con PyTorch GPU 120
8.21 Qué conviene observar al analizar una implementación GPU 123
8.22 Debugging y profiling en GPU 123
8.23 Cierre de la unidad 124
8.24 Ejercicios del capítulo 125

III Cierre y proyección **126**

9 Cierre y conclusión **127**
9.1 Criterios que conviene conservar 128

9.2 Del fundamento a la práctica contemporánea 130
9.3 Continuidad del estudio 130

10 Anexo: trabajos prácticos integradores **132**
10.1 Criterios generales de medición 132
10.2 Trabajo práctico 1: suma paralela de un vector 133
10.2.1 Objetivo 133
10.2.2 Punto de partida 133
10.2.3 Consignas 134
10.2.4 Resultados esperados 134
10.2.5 Informe 135
10.2.6 Entrega 135
10.3 Trabajo práctico 2: multiplicación de matrices 136
10.3.1 Objetivo 136
10.3.2 Implementaciones requeridas 136
10.3.3 Consignas 136
10.3.4 Resultados esperados 137
10.3.5 Preguntas de análisis 137
10.3.6 Informe 138
10.3.7 Entrega 138
10.4 Trabajo práctico 3: filtro de Sobel para detección de bordes 138
10.4.1 Objetivo 138
10.4.2 Conversión a escala de grises 139
10.4.3 Ejemplo mínimo de Sobel 140
10.4.4 Etapa 1: CPU secuencial, Numba CPU y NumPy 140
10.4.5 Etapa 2: Numba GPU 141
10.4.6 Etapa 3: PyTorch CPU y PyTorch GPU 141
10.4.7 Métrica de salida 142
10.4.8 Resultados esperados 142
10.4.9 Preguntas de análisis 142
10.4.10 Informe 143
10.4.11 Entrega 143
10.5 Trabajo práctico 4: procesamiento de video 4K con PyTorch 143
10.5.1 Objetivo 143

10.5.2 Filtros posibles 144
10.5.3 Implementaciones requeridas 144
10.5.4 Manejo de memoria 145
10.5.5 Benchmarking 146
10.5.6 Informe 146
10.5.7 Entrega 147

1 Presentación

Este libro fue elaborado para la asignatura Sistemas Paralelos de la Universidad Nacional de Tierra del Fuego (UNTDF), Argentina, y está dirigido a estudiantes universitarios que necesitan una introducción ordenada a los fundamentos del procesamiento paralelo, sus arquitecturas, sus modelos de programación y sus herramientas de implementación.

La computación paralela tiene hoy un lugar relevante en el desarrollo de software, el procesamiento de datos, la simulación científica y la inteligencia artificial. Incluso tareas cotidianas, como procesar imágenes o analizar grandes volúmenes de información, dependen cada vez más de arquitecturas con múltiples núcleos, unidades vectoriales y aceleradores especializados. Por ese motivo, comprender sus principios ya no constituye un interés restringido a ámbitos muy específicos, sino una parte importante de la formación en informática.

Estudiar paralelismo implica aprender a reconocer cómo se divide un problema, qué costos introduce la coordinación entre tareas, qué límites impone la arquitectura disponible y de qué manera se interpreta el rendimiento obtenido. Dicho de otro modo, no alcanza con observar que una implementación funciona: conviene entender bajo qué condiciones mejora su desempeño y cómo se evalúan sus resultados.

Con ese marco, el recorrido propuesto combina definiciones, ejemplos, implementaciones y métricas. A lo largo de los capítulos se desarrollan los conceptos necesarios para pasar de una comprensión general del campo a una aproximación más concreta a sus principales estrategias y herramientas de trabajo.

1.1. Objetivos del libro

Al finalizar este recorrido se espera que el estudiante pueda:

- distinguir computación secuencial, concurrente, paralela y distribuida;
- reconocer las principales arquitecturas, modelos de memoria y criterios básicos de organización del cómputo paralelo;
- interpretar métricas como speed-up y eficiencia al analizar resultados experimentales;
- identificar modelos y APIs de programación paralela adecuados para distintos tipos de problemas;
- desarrollar implementaciones introductorias en Python para explorar paralelismo explícito en CPU, reformulación sobre arreglos y ejecución sobre GPU;
- analizar críticamente las ventajas y los límites de distintas estrategias de paralelización.

1.2. Organización del recorrido

El libro sigue una secuencia progresiva que va de los fundamentos conceptuales a las estrategias de implementación. En los primeros capítulos se introducen las distinciones básicas entre computación secuencial, concurrencia, paralelismo y distribución, y luego se presentan las arquitecturas, los modelos de memoria y las métricas que permiten analizar rendimiento. Esa base resulta necesaria para comprender por qué ciertas decisiones de diseño escalan mejor que otras y qué límites impone cada plataforma.

Sobre ese marco se desarrollan después los modelos clásicos de programación paralela y algunas APIs históricas del área, como Pthreads, OpenMP y MPI. Estas referencias no aparecen solo como repertorio técnico, sino como formas de organizar problemas, comunicación y sincronización según el tipo de arquitectura disponible.

La segunda mitad se concentra en Python como lenguaje de trabajo para explorar distintos niveles de abstracción. Primero se estudia el paralelismo explícito en CPU mediante hilos, procesos y compilación JIT. Luego se introduce una estrategia complementaria: reformular el cálculo sobre arreglos y tensores completos mediante vectorización, broadcasting y trabajo con bibliotecas como NumPy y PyTorch en CPU. A continuación se analiza el paso hacia GPU, primero desde un modelo más cercano al hardware y luego desde herramientas de más alto nivel.

El cierre del libro recupera esa progresión para sintetizar criterios de elección, límites de escalabilidad y relaciones entre estructura del problema, forma de implementación y arquitectura de ejecución.

1.3. Cómo trabajar con este libro

Conviene recorrer el libro en orden. Los capítulos posteriores retoman definiciones, métricas, ejemplos y criterios introducidos al comienzo, de modo que saltear secciones suele dificultar la comprensión global del recorrido. Antes de profundizar en bibliotecas o APIs, resulta importante comprender qué problema intenta resolver el paralelismo y por qué su rendimiento no depende solamente de agregar más hilos, más procesos o más hardware.

La lectura se aprovecha mejor cuando se combina con implementación y medición. Muchos de los conceptos desarrollados en el libro se vuelven más claros al comparar una versión secuencial con una versión paralela, al registrar tiempos de ejecución y al analizar por qué dos soluciones correctas pueden exhibir desempeños muy diferentes.

También conviene prestar atención al tipo de problema que se trabaja en cada capítulo. En algunos casos, el interés estará puesto en repartir tareas de manera explícita; en otros, en reorganizar el cálculo para aprovechar mejor la memoria y las bibliotecas optimizadas; y en otros, en reconocer cuándo una GPU ofrece una ventaja real. Mantener esa perspectiva ayuda a evitar una lectura reducida a herramientas aisladas.

Por último, resulta importante documentar el contexto de ejecución. En sistemas paralelos, variables como la cantidad de cores, el sistema operativo, la disponibilidad de GPU, la memoria y las bibliotecas instaladas influyen directamente sobre los resultados. Registrar esas condiciones forma parte del análisis técnico y permite interpretar con mayor cuidado las comparaciones que aparecen a lo largo del libro.

1.4. Herramientas mínimas para empezar

Para trabajar con este libro alcanza, como base, con una instalación funcional de Python y un editor de código. A lo largo del recorrido se incorporarán bibliotecas como NumPy, Numba y PyTorch, según el tipo de problemas y ejemplos que se desarrollan en los capítulos.

Siempre que sea posible, conviene trabajar en un entorno nativo y, de preferencia, en Linux, aunque buena parte de las prácticas también puede realizarse en otros sistemas operativos. Cuando no se disponga de una GPU local, algunas pruebas vinculadas con aceleración por hardware podrán ejecutarse en servicios como Google Colab o en otros entornos equivalentes.

En el contexto de la asignatura, también podrán utilizarse recursos institucionales disponibles para ciertas actividades específicas, como el nodo de alta performance de la UNTDF.

1.5. Sobre el autor

Federico Gonzalez Brizzio es profesor adjunto en la carrera de Informática de la UNTDF, donde además de la asignatura Sistemas Paralelos, tiene a su cargo las cátedras de Laboratorio de Software, y Desarrollo Web I y II. Es Licenciado en Informática por la Universidad Nacional de la Patagonia San Juan Bosco, con un Máster en Smart Cities por la Universidad de Girona, un posgrado en Gobierno Abierto por la Organización de los Estados Americanos (OEA) y actualmente realiza su doctorado en Inteligencia Artificial en la Universidad de Barcelona. Es cofundador de Panalsoft, una empresa con sede en Ushuaia, Argentina, orientada a la transformación digital de organizaciones públicas y privadas. Trabaja en el Institute of Science and Technology Austria (ISTA), un centro internacional de investigación científica ubicado en Klosterneuburg, a las afueras de Viena.

1.6. Ilustraciones y revisión

Las ilustraciones incluidas en el libro fueron diseñadas íntegramente con ChatGPT 5.4. En su elaboración no hubo intervención humana sobre la construcción visual propiamente dicha más allá del prompt inicial que orientó cada imagen y de la posterior decisión editorial de incorporarla al capítulo correspondiente. Las figuras deben leerse como parte de una metodología de producción asistida, integrada deliberadamente al proceso de escritura.

La revisión del libro también se realizó mediante una modalidad de trabajo de a pares. Ese equipo de revisión estuvo formado por el autor, Federico Gonzalez Brizzio, y por GitHub Copilot, utilizando para esa tarea modelos Claude Sonnet 4.6 y GPT 5.4. Esta revisión se aplicó a la relectura técnica y editorial de los capítulos, con el propósito de detectar inconsistencias, ajustar formulaciones, mejorar la claridad expositiva y controlar la coherencia del recorrido pedagógico. La responsabilidad autoral final sobre el contenido, su selección y su organización permanece, en todos los casos, en la figura del autor.

1.7. Reporte de errores y sugerencias

Si encuentra errores, inconsistencias o tiene sugerencias para mejorar el contenido, puede reportarlas por alguno de estos canales:

- Correo electrónico: fgonzalez@untdf.edu.ar
- GitHub: https://github.com/fedegonzal/books
- Website: https://fedegonzalez.com

Parte I

Fundamentos del paralelismo

2 Conceptos básicos

Antes de analizar arquitecturas o escribir código paralelo, conviene comprender con precisión los conceptos básicos que organizan el campo. En esta unidad se distinguen distintos modos de ejecución, se revisan límites históricos de la computación secuencial y se introducen ideas centrales como núcleo de procesamiento, sincronización y niveles de paralelismo.

La computación paralela ocupa hoy un lugar central en la informática, porque muchos problemas actuales ya no pueden resolverse de manera eficiente programando algo secuencialmente de forma tradicional. Procesamiento de imágenes y video, simulaciones, análisis de grandes volúmenes de datos, inteligencia artificial y servicios distribuidos exigen aprovechar mejor los recursos de hardware disponibles. Por ese motivo, conviene estudiar el paralelismo no como un tema aislado, sino como una respuesta técnica a necesidades concretas de rendimiento.

También es importante notar que el problema involucra al mismo tiempo hardware y software. No alcanza con disponer de múltiples núcleos, clústeres o GPU: es necesario diseñar algoritmos capaces de dividir el trabajo, coordinar tareas y sincronizar resultados. Dicho de forma simple, los sistemas paralelos se ubican en la intersección entre arquitectura de cómputo, y modelos de programación.

2.1. Objetivos del capítulo

- distinguir computación secuencial, concurrente, paralela y distribuida;
- explicar de manera introductoria cómo ejecuta instrucciones un procesador;
- relacionar la evolución del hardware con la necesidad de paralelizar;
- presentar los principales niveles de paralelismo.

2.2. Por qué estudiar paralelismo

El paralelismo ya forma parte del hardware cotidiano. Computadoras personales, notebooks, teléfonos móviles y placas de video incorporan distintos niveles de ejecución simultánea. En consecuencia, incluso en desarrollos que no pertenecen a la computación de alto rendimiento, conviene entender cómo se reparten tareas, qué costos introduce la sincronización y qué límites aparecen cuando se busca acelerar un programa.

Estudiar sistemas paralelos implica construir criterios para distinguir cuándo un problema puede beneficiarse del paralelismo, qué arquitectura resulta más adecuada y cómo evaluar si la mejora obtenida justifica los recursos empleados.

Por ese motivo, conviene pensar el paralelismo también como una forma de analizar problemas. Antes de escribir una implementación paralela, suele ser necesario preguntarse qué partes del trabajo pueden ejecutarse de manera independiente, cuáles dependen de resultados previos y en qué momento será necesario recombinar los resultados parciales. Esa mirada de descomposición es tan importante como la herramienta elegida para programar.

2.3. Computación secuencial

La computación secuencial es la forma tradicional de ejecución de programas. Las instrucciones se realizan una tras otra, en un orden determinado, y en cada instante solo una rama de ejecución avanza efectivamente. Aunque existan estructuras condicionales o repetitivas, el programa sigue un recorrido único en cada paso.

Comprender esta modalidad es importante porque el paralelismo no reemplaza por completo a la ejecución secuencial. Más bien se construye sobre ella, identificando qué partes de un problema pueden descomponerse y cuáles deben mantenerse en orden.

Un ejemplo simple ayuda a ver esta diferencia. Si se desea sumar los valores `(40 + 20 + 10 + 60)`, una implementación secuencial realiza una acumulación lineal: primero suma `(40 + 20)`, luego agrega `10` y finalmente `60`. En cambio, una descomposición paralela puede organizar el mismo cálculo en dos sumas parciales independientes, por ejemplo `(40 + 20)` y `(10 + 60)`, para luego combinar ambos resultados. El problema es el mismo, pero cambia la forma de organizar el trabajo.

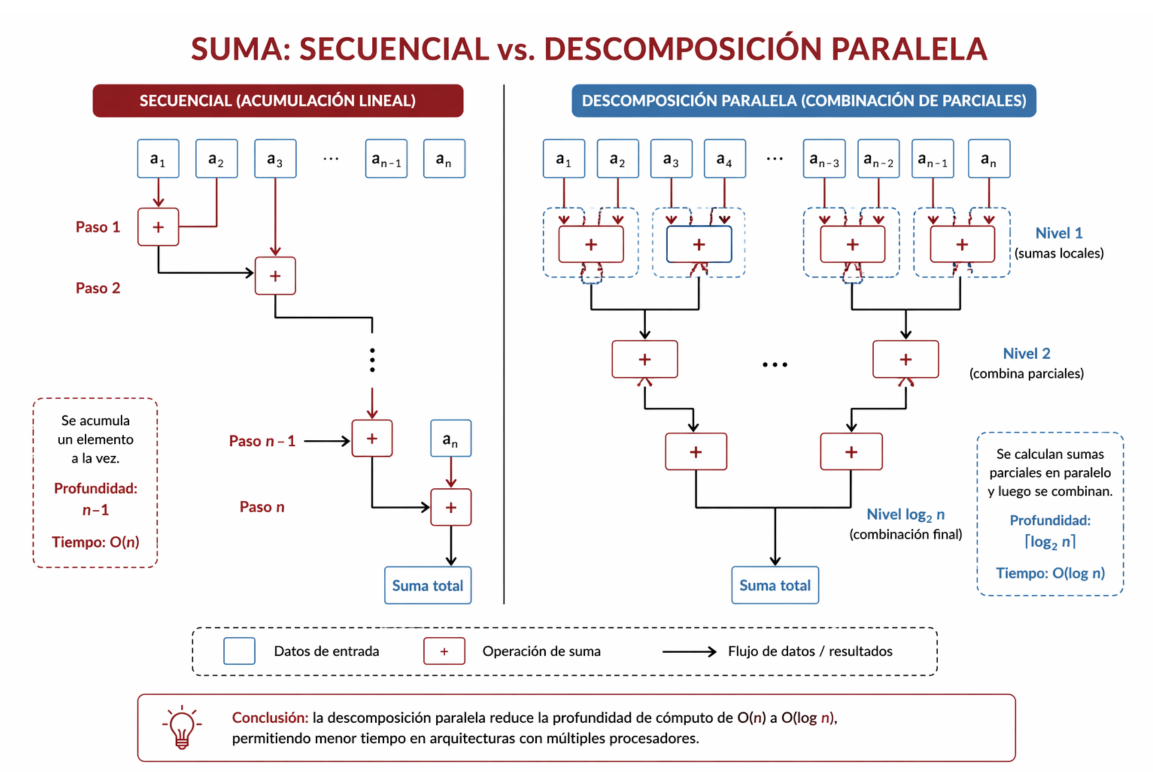

Figura 2.1: Comparación entre una suma secuencial y una descomposición paralela basada en resultados parciales.

Un pseudocódigo mínimo permite ver con más claridad esa reorganización:

```
total = 40 + 20
total = total + 10
total = total + 60
```

Frente a una descomposición en árbol de dos etapas:

```
parcial_1 = 40 + 20
parcial_2 = 10 + 60
total = parcial_1 + parcial_2
```

En la segunda formulación, `parcial_1` y `parcial_2` podrían calcularse de manera simultánea. La sincronización aparece en el momento de combinar ambos resultados para producir el valor final.

Esta observación parece menor, aunque en realidad introduce una idea central: paralelizar no consiste únicamente en ejecutar más cosas al mismo tiempo, sino en reorganizar un problema para exponer partes independientes sin perder corrección.

2.4. Velocidad de ejecución y ciclo del procesador

La velocidad de ejecución de un programa no depende solo de cuántas instrucciones contiene, sino también de cómo el procesador las organiza internamente. Una CPU trabaja siguiendo ciclos de reloj, que marcan el ritmo con el que se coordinan sus operaciones básicas. En cada ciclo, distintas partes del procesador pueden avanzar en tareas como buscar instrucciones, decodificarlas, ejecutarlas o escribir resultados.

De manera simplificada, el ciclo de ejecución de una instrucción puede describirse mediante cuatro etapas: búsqueda de la instrucción en memoria (*fetch*), interpretación de la operación que debe realizarse (*decode*), ejecución de la operación (*execute*) y escritura del resultado (*write back*). Los procesadores modernos aplican muchas optimizaciones sobre este esquema básico, pero la idea general permite entender un punto importante: el rendimiento no depende únicamente del programa escrito, sino también de la forma en que el hardware logra solapar, ordenar y completar instrucciones.

Durante buena parte de la historia de la computación personal y de servidores, la mejora del rendimiento estuvo asociada sobre todo al aumento de la frecuencia de reloj y al perfeccionamiento de los procesadores mononúcleo. En ese contexto, la expectativa habitual era que cada nueva generación de hardware ejecutara el mismo software más rápido, aun sin cambios importantes en los programas. Esta etapa estuvo acompañada por la influencia de la llamada ley de Moore, que describía el crecimiento sostenido de la cantidad de transistores integrados en un chip y funcionó durante muchos años como marco general para pensar la evolución del hardware.

Sin embargo, ese crecimiento no podía traducirse indefinidamente en aumentos lineales de frecuencia. A medida que los procesadores alcanzaron velocidades cada vez mayores, comenzaron a hacerse más visibles problemas de temperatura, consumo energético y disipación térmica. En otras palabras, ya no resultaba viable sostener la mejora del rendimiento simplemente haciendo que un único núcleo trabajara cada vez más rápido. Ese límite marcó un cambio técnico e histórico importante en la industria.

La respuesta consistió en orientar la evolución de los procesadores hacia la integración de varios núcleos dentro de un mismo chip. En lugar de depender solo de un núcleo más veloz, comenzó a ganar importancia la posibilidad de ejecutar varias tareas o varias partes de un mismo problema en paralelo. En el mercado de consumo, esta transición se volvió especialmente visible con familias de procesadores de doble núcleo, entre ellas Intel Core 2 Duo, que ayudaron a consolidar la idea de que el rendimiento futuro dependería cada vez más de explotar concurrencia y paralelismo, y no únicamente de aumentar megahertz.

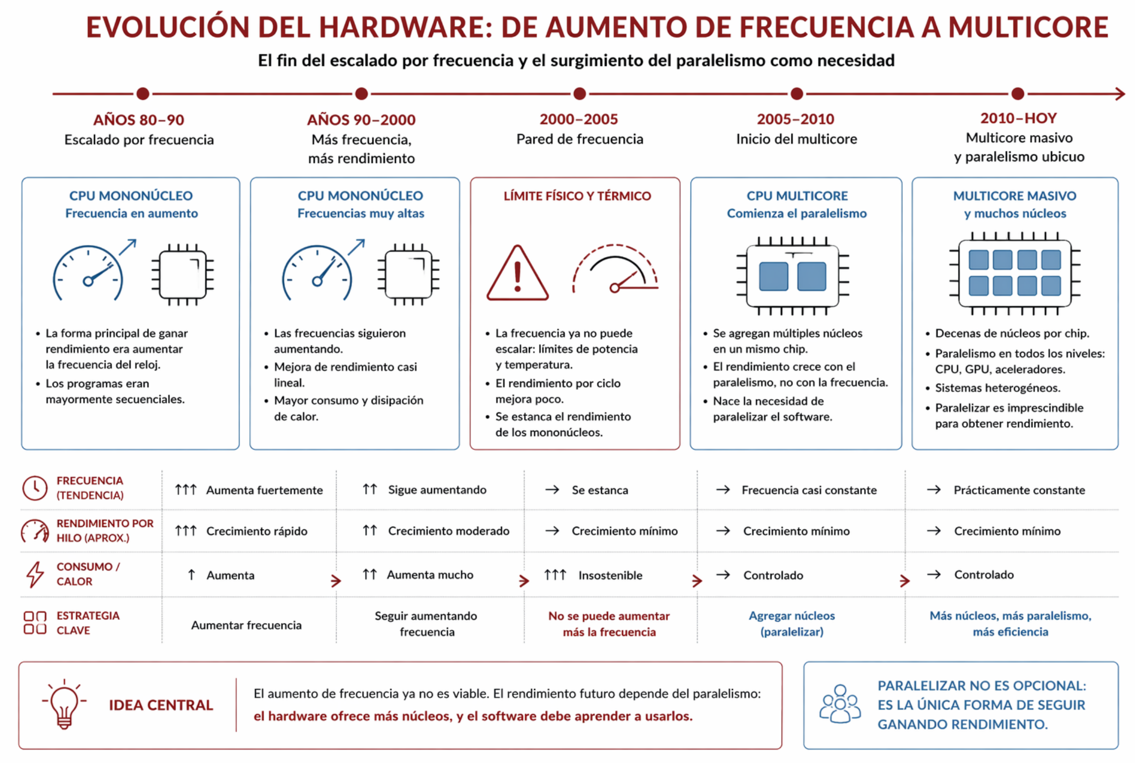

Figura 2.2: Cambio de estrategia en el hardware: del aumento de frecuencia al aprovechamiento de múltiples núcleos.

Este cambio tuvo consecuencias directas sobre el software. Mientras en la etapa mononúcleo gran parte de la mejora podía obtenerse por renovación de hardware, la era multicore exigió programas capaces de distribuir trabajo entre varios núcleos. Por ese motivo, estudiar paralelismo dejó de ser una cuestión reservada a supercomputadoras o a nichos especializados y pasó a convertirse en un problema general del desarrollo de software contemporáneo.

2.5. Los límites de la computación secuencial

La disponibilidad de varios núcleos no elimina, por sí sola, los límites de la computación secuencial. Aunque el hardware permita ejecutar varias tareas al mismo tiempo, muchos programas contienen partes que deben mantenerse en orden, ya sea porque dependen de resultados previos o porque concentran decisiones que no pueden repartirse fácilmente.

A esto se suman los costos de coordinación. Dividir trabajo entre varios núcleos exige crear tareas, sincronizar resultados, compartir datos y, en muchos casos, esperar a que una parte del programa alcance a otra. Por ese motivo, una implementación paralela no siempre mejora en proporción directa a la cantidad de recursos disponibles. A veces la ganancia es modesta y, en situaciones mal diseñadas, incluso puede ocurrir que el overhead reduzca o anule la ventaja esperada.

Este punto resulta decisivo porque obliga a abandonar una idea ingenua: la de suponer que más hardware garantiza automáticamente más velocidad. En realidad, el rendimiento depende de la estructura del problema, del porcentaje de trabajo que puede ejecutarse en paralelo y del costo introducido por la coordinación. Más adelante se verá que estas restricciones pueden expresarse de manera más formal mediante nociones como speed-up, eficiencia y fracción secuencial.

2.6. Concurrencia, paralelismo y distribución

La concurrencia se refiere a la coexistencia de varias tareas en ejecución o en progreso, aun cuando no siempre se resuelva un mismo problema. El paralelismo, en cambio, apunta específicamente a dividir un problema en partes que puedan ejecutarse simultáneamente para obtener un resultado común.

La computación distribuida agrega otra dimensión: los procesadores o nodos pueden encontrarse físicamente separados y cooperar mediante intercambio de datos. En la práctica contemporánea, estos enfoques suelen combinarse. Por ese motivo, conviene distinguir sus diferencias conceptuales sin perder de vista que en sistemas reales aparecen mezclados.

Una comparación breve permite aclarar la distinción. Si en una computadora se descarga un archivo mientras se reproduce música y se edita un documento, hay concurrencia: varias tareas avanzan en el mismo intervalo, aunque no cooperan para resolver un único problema. En cambio, si una suma grande de datos se divide en bloques para que varios núcleos calculen subtotales y luego se

integren en un único resultado, hay paralelismo. En ambos casos puede haber actividad simultánea, pero solo en el segundo existe cooperación directa sobre una misma tarea.

Figura 2.3: Diferencias conceptuales entre concurrencia, paralelismo y computación distribuida.

2.7. Niveles de paralelismo

Conviene distinguir distintos niveles de paralelismo porque no todos operan en la misma escala ni exigen el mismo tipo de intervención por parte de quien programa. Algunos aparecen cerca del hardware y funcionan de manera casi transparente, mientras que otros requieren decisiones explícitas sobre cómo dividir trabajo, datos y sincronización.

Entre los niveles más cercanos al hardware suele mencionarse el paralelismo a nivel de bit, que aparece cuando una arquitectura amplía la cantidad de bits que puede procesar en una sola operación. También resulta importante el paralelismo a nivel de instrucción, donde el procesador reorganiza y solapa internamente la ejecución de varias instrucciones. En este punto conviene incorporar la idea de pipelining en hardware. Un pipeline divide la ejecución de una instrucción en etapas, por ejemplo búsqueda, decodificación, ejecución y escritura, y permite que distintas

instrucciones ocupen simultáneamente distintas etapas del procesador. De ese modo, mientras una instrucción se ejecuta, otra puede estar siendo decodificada y una tercera puede estar siendo traída desde memoria. No se trata todavía de paralelismo entre programas diferentes, sino de una forma de aumentar el rendimiento interno del procesador mediante solapamiento.

En un nivel más visible para el software aparecen formas de paralelismo que sí exigen decisiones de diseño. El paralelismo de tareas divide un problema en subtareas que pueden asignarse a distintas unidades de procesamiento. El paralelismo de datos, en cambio, aplica la misma operación sobre particiones independientes de un conjunto de entrada. La vectorización puede entenderse como una forma particularmente importante de este segundo enfoque, ya que aprovecha operaciones repetitivas sobre estructuras de datos para ejecutarlas de manera más eficiente, muchas veces apoyándose en capacidades del hardware como SIMD.

Distinguir estos niveles es importante porque la mejora de rendimiento no siempre proviene del mismo lugar. A veces surge de mecanismos internos del procesador, como el pipelining o el paralelismo a nivel de instrucción, que resultan casi invisibles para quien programa. En otros casos depende de decisiones explícitas, como repartir tareas entre varios workers o reorganizar datos para aplicar la misma operación muchas veces. Reconocer estas diferencias ayuda a elegir mejor las herramientas, formular expectativas realistas y entender por qué dos programas pueden comportarse de manera distinta aun cuando se ejecuten sobre el mismo hardware.

2.8. Observación del uso de CPU

Al analizar una implementación conviene observar la carga de cada core y no solo un porcentaje global de CPU. Esa diferencia es importante porque un programa puede mostrar un uso alto del procesador en términos generales y, sin embargo, seguir concentrando casi todo su trabajo en un único núcleo. Mirar la actividad por core permite detectar con más claridad si una implementación realmente distribuye el trabajo o si mantiene un comportamiento predominantemente secuencial.

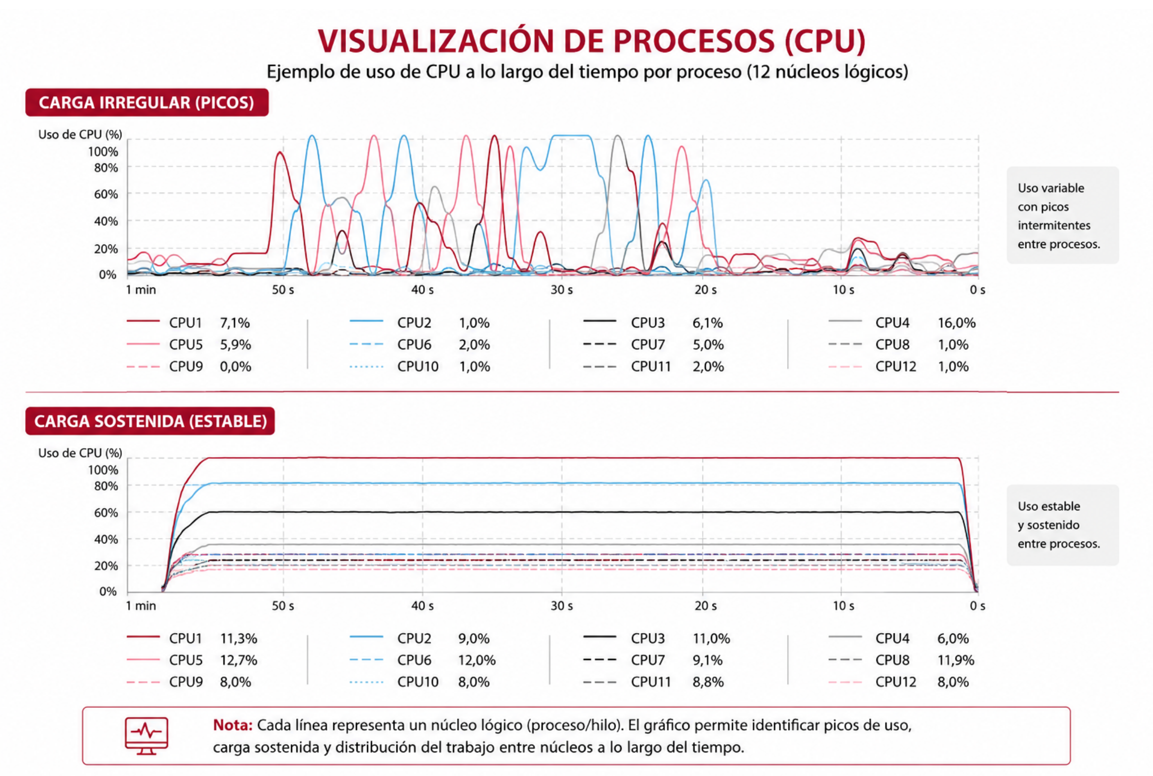

Figura 2.4: Ejemplo de visualización del uso de CPU por núcleo lógico a lo largo del tiempo, comparando una carga irregular con picos y una carga sostenida más estable.

En Linux, una primera herramienta útil es `top`, disponible desde la línea de comandos. Permite ver procesos activos, consumo de CPU y memoria, y sirve como punto de partida para observar qué ocurre mientras se ejecuta un programa. También existen alternativas más visuales, como `htop` o `btop`, que facilitan la lectura porque muestran barras por núcleo, colores y una organización más clara de los procesos en ejecución. Para los fines de este libro, cualquiera de estas herramientas resulta suficiente siempre que permita responder una pregunta simple: si el programa intenta paralelizar trabajo, ¿se reparte efectivamente entre varios cores o no?

En macOS, la referencia más directa es el Monitor de Actividad, que ofrece una vista gráfica del uso del sistema y permite inspeccionar procesos, consumo de CPU y carga general. Si se prefiere trabajar desde terminal, también puede usarse `top`, aunque en la práctica la herramienta visual suele ser más cómoda para una observación inicial. En Windows, el Administrador de tareas cumple un papel equivalente y permite ver el uso global de CPU, la actividad por proceso y, en la pestaña de rendimiento, la distribución de carga del procesador. Según la versión del sistema, también puede recurrirse al Monitor de recursos para un análisis algo más detallado.

Lo importante no es dominar una herramienta específica, sino adquirir un criterio de observación. Si un programa secuencial mantiene ocupado principalmente un solo core, ese comportamiento es esperable. Si una versión paralela usa varios núcleos, conviene mirar si la carga se reparte de manera relativamente equilibrada o si algunos cores trabajan mucho más que otros. Esa observación no reemplaza la medición de tiempos, pero ayuda a interpretar mejor los resultados y a detectar tempranamente problemas de diseño o de implementación.

2.9. Cierre de la unidad

Con este marco ya se cuenta con una base conceptual suficiente para avanzar hacia el análisis de plataformas y métricas. En particular, el capítulo permitió distinguir tipos de ejecución, entender por qué el multicore modificó el problema del rendimiento y ver que paralelizar exige descomponer, coordinar y sincronizar trabajo.

En el próximo capítulo se estudiarán clasificaciones de arquitecturas paralelas y medidas de rendimiento como speed-up y eficiencia. Ese paso permitirá no solo describir plataformas, sino también analizar por qué una implementación paralela puede rendir mucho mejor, apenas mejor o incluso peor de lo esperado.

2.10. Ejercicios del capítulo

- Distinga computación secuencial, concurrente, paralela y distribuida.
- Explique por qué la aparición de procesadores multicore modificó las exigencias sobre el software.
- Describa la diferencia entre paralelismo de tareas y paralelismo de datos.
- Explique por qué conviene entender el paralelismo también como una forma de descomponer problemas.
- Proponga un ejemplo cotidiano de tarea concurrente y otro de tarea paralela.
- Observe el monitor de actividad de su sistema y describa qué información ofrece sobre el uso de los distintos cores.
- Reescriba conceptualmente una suma secuencial de cuatro valores como una suma en árbol de dos etapas e indique dónde aparece la sincronización.

- Justifique por qué disponer de más cores no garantiza, por sí solo, una mejora lineal del rendimiento.

3 Arquitectura y métricas

Una vez introducidos los conceptos básicos del paralelismo, conviene estudiar dos dimensiones que condicionan cualquier implementación real. Por un lado, la arquitectura disponible define cómo se organizan procesadores, memorias y mecanismos de comunicación. Por otro, las métricas permiten evaluar si una versión paralela realmente mejora el desempeño o si solo agrega complejidad y costo.

Estas dos dimensiones no deben analizarse por separado. Una misma estrategia de programación puede comportarse de manera muy distinta según el tipo de memoria, la jerarquía de caché, el costo de sincronización o el ancho de banda disponible.

3.1. Objetivos del capítulo

- presentar clasificaciones básicas de plataformas de cómputo paralelo;
- distinguir memoria compartida, memoria distribuida y modelos híbridos;
- formalizar las nociones de speed-up y eficiencia;
- introducir leyes y conceptos que explican los límites de escalabilidad;
- relacionar jerarquía de memoria y rendimiento observado.

3.2. Clasificación por mecanismo de control

Una clasificación clásica de las arquitecturas paralelas es la taxonomía de Flynn. Este esquema organiza las plataformas según la relación entre flujo de instrucciones y flujo de datos. Aunque se trata de una clasificación histórica, sigue siendo útil para introducir diferencias fundamentales entre tipos de procesamiento.

- SISD: una única secuencia de instrucciones opera sobre un único flujo de datos. Corresponde al modelo secuencial tradicional. Un ejemplo representativo sería una computadora mononúcleo clásica ejecutando un programa secuencial, o incluso un microcontrolador simple que resuelve una tarea en una única línea de ejecución.
- SIMD: una misma instrucción se aplica en paralelo sobre múltiples datos. Este esquema resulta especialmente relevante para vectorización, procesamiento de imágenes y muchas operaciones internas de las GPU. Como ejemplo, pueden pensarse las extensiones vectoriales de procesadores modernos, como SSE o AVX en CPU, o una GPU cuando aplica la misma operación sobre miles de píxeles o elementos de un tensor.
- MISD: múltiples instrucciones actúan sobre un mismo flujo de datos. Se mantiene sobre todo como categoría teórica y tiene escasa presencia en sistemas de propósito general. Cuando se lo ilustra con ejemplos, suele recurrirse a ciertos sistemas de control tolerantes a fallos, donde varias unidades procesan la misma entrada con lógicas distintas para aumentar confiabilidad, aunque no se trate de un modelo habitual en computadoras de uso general.
- MIMD: múltiples instrucciones operan sobre múltiples datos. Es la categoría más común en computación paralela de propósito general y abarca desde procesadores multicore hasta numerosos sistemas distribuidos. Una notebook o una PC actual con varios núcleos ya responde a este esquema, y también lo hacen un servidor multiprocesador o un clúster donde distintos nodos ejecutan tareas diferentes sobre datos distintos.

En términos generales, SIMD resulta adecuado cuando hay gran regularidad en los datos y la operación aplicada es la misma. MIMD, en cambio, ofrece mayor flexibilidad y permite abordar problemas con tareas heterogéneas, dependencias más complejas o estrategias de sincronización variadas.

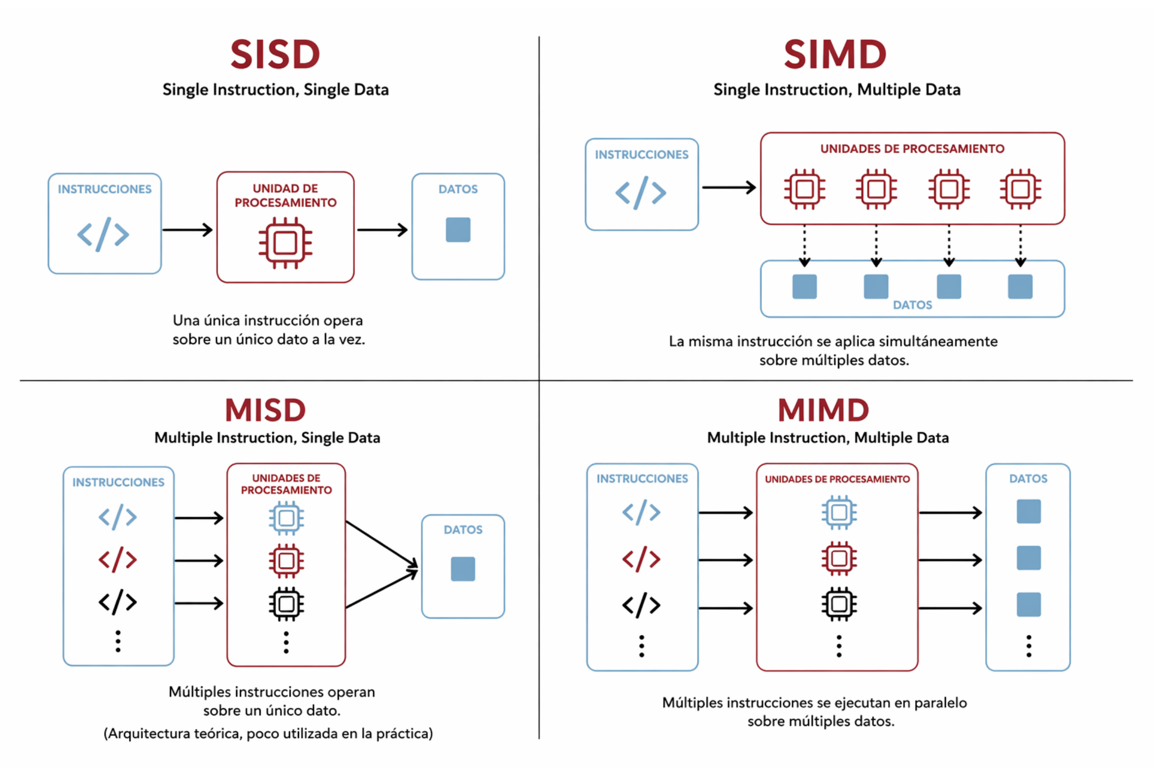

Figura 3.1: Taxonomía de Flynn para clasificar arquitecturas según la cantidad de flujos de instrucciones y de datos.

3.3. Clasificación por organización física

Otra clasificación central atiende al modo en que se organiza la memoria del sistema. Este criterio es decisivo porque modifica tanto la forma de programar como el costo de la comunicación.

En sistemas de memoria compartida, varios núcleos o procesadores acceden a un mismo espacio de direcciones. Esto simplifica el intercambio de datos, pero exige coordinar accesos concurrentes y controlar problemas de sincronización.

En sistemas de memoria distribuida, cada nodo posee memoria propia. Los datos no se comparten automáticamente y deben enviarse de manera explícita entre procesos o máquinas. Esta organización aparece con frecuencia en clústeres y entornos de cómputo científico.

También existen modelos híbridos, en los que cada nodo tiene memoria compartida local pero se comunica con otros nodos como si formara parte de un sistema distribuido. Este esquema es

habitual en servidores modernos y en numerosos clústeres de alto rendimiento.

Esta distinción ayuda a anticipar decisiones de diseño. Como se verá más adelante, la elección de una estrategia de paralelización suele estar condicionada por la arquitectura. Herramientas como OpenMP se ajustan de manera natural a memoria compartida, mientras que MPI responde mejor a memoria distribuida. En modelos híbridos, ambas estrategias suelen combinarse.

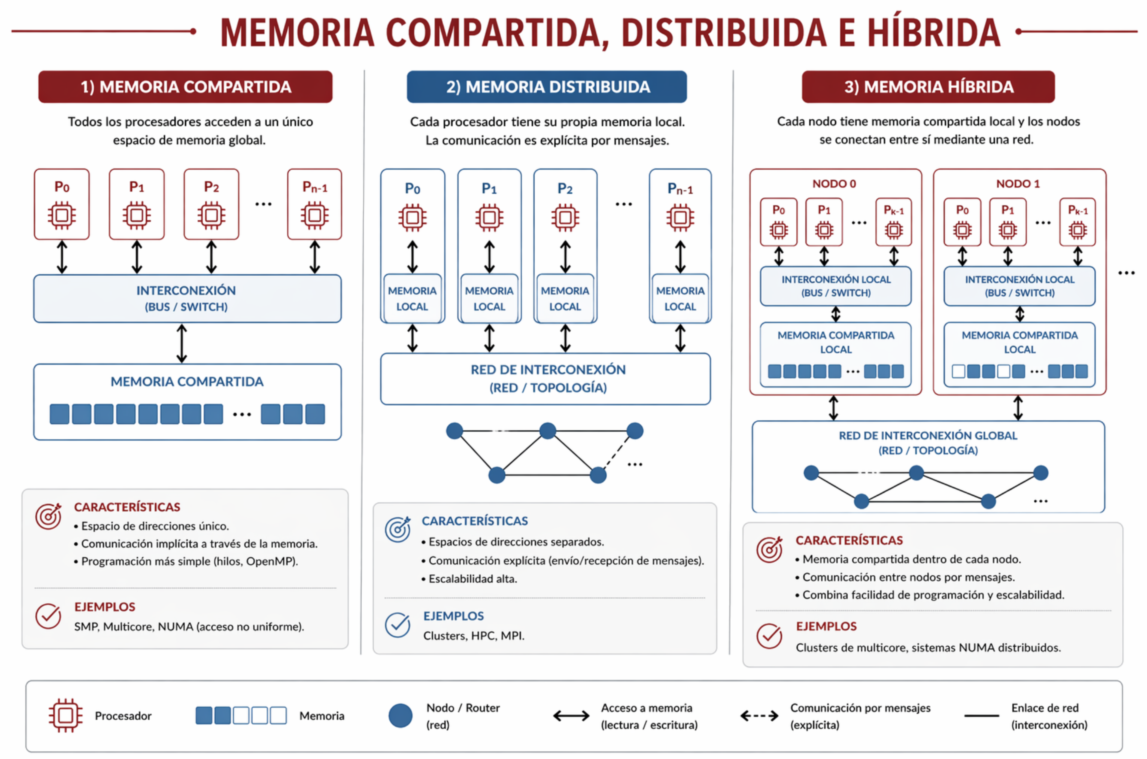

Figura 3.2: Comparación entre arquitecturas de memoria compartida, distribuida e híbrida.

3.4. Jerarquía de memoria y localidad

En arquitectura paralela no alcanza con conocer la cantidad de cores. También importa cómo acceden a los datos. La memoria principal es mucho más lenta que los registros y las memorias caché, por lo que el rendimiento depende en gran medida de la capacidad de reutilizar datos cercanos en tiempo o en espacio.

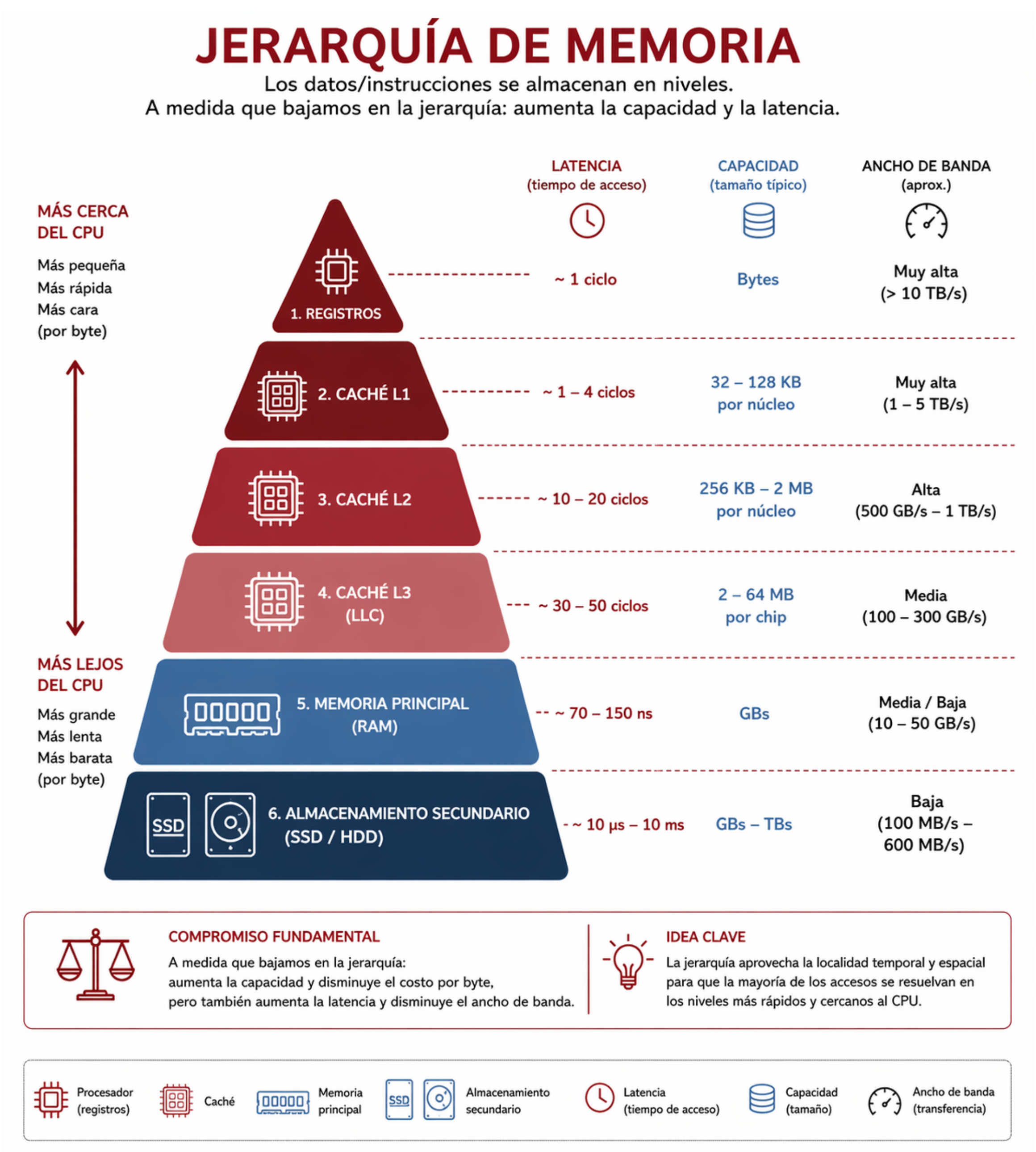

Figura 3.3: Jerarquía de memoria y relación entre cercanía al procesador, capacidad y latencia de acceso.

La localidad temporal aparece cuando un dato utilizado recientemente vuelve a usarse en poco tiempo. Si una variable, una fila de una matriz o un bloque de datos acaba de cargarse en caché y el programa la necesita nuevamente enseguida, es más probable que ese acceso pueda resolverse sin volver a buscar la información en memoria principal. Dicho de forma simple, la localidad temporal

mejora cuando un programa reutiliza pronto aquello que ya usó.

La localidad espacial aparece cuando, al acceder a una posición de memoria, es probable que pronto se necesiten posiciones cercanas. Esto ocurre, por ejemplo, al recorrer un vector de manera secuencial: después de leer un elemento, suelen leerse los siguientes, que están almacenados en direcciones contiguas. Como la caché transfiere datos en bloques o líneas, aprovechar posiciones vecinas aumenta la probabilidad de obtener un acierto de caché y reduce el costo de acceso.

En programación paralela esta cuestión es especialmente importante porque dos implementaciones con el mismo número de hilos pueden rendir de forma muy distinta según cómo recorran los datos. A veces, reorganizar un arreglo o cambiar el orden de recorrido mejora más que agregar workers, justamente porque mejora la localidad y reduce fallos de caché. Más adelante, cuando se estudie la multiplicación de matrices, aparecerá un ejemplo representativo: trabajar con una versión transpuesta de una de las matrices puede volver más regular el acceso a memoria y mejorar el aprovechamiento de caché, aun cuando el algoritmo siga realizando la misma cantidad de operaciones aritméticas.

3.5. False sharing y NUMA

Un problema frecuente en memoria compartida es el false sharing. Ocurre cuando dos hilos modifican variables distintas que, sin embargo, residen en la misma línea de caché. Aunque cada hilo trabaje sobre un dato diferente, el hardware interpreta que ambos compiten por la misma región de memoria y fuerza invalidaciones o recargas innecesarias. El resultado es una degradación de rendimiento que puede ser importante aun cuando el algoritmo parezca correctamente paralelizado.

Un ejemplo simple ayuda a verlo. Supóngase un arreglo de contadores donde cada hilo actualiza una posición distinta, por ejemplo `counters[0]`, `counters[1]`, `counters[2]` y `counters[3]`. A primera vista no habría conflicto, porque cada hilo escribe en una variable diferente. Sin embargo, si esas posiciones quedan alojadas en una misma línea de caché, cada escritura puede invalidar la copia observada por otros núcleos. El programa sigue siendo correcto desde el punto de vista lógico, pero el rendimiento cae porque el hardware debe sincronizar constantemente esa región de memoria.

Identificar esta situación no siempre es sencillo, porque el error no suele aparecer como un fallo visible del programa. Más bien se manifiesta como una pérdida de rendimiento difícil de explicar: al aumentar la cantidad de hilos, el tiempo no mejora como se esperaba o incluso empeora, aun cuando el reparto de trabajo parezca razonable. Una señal típica es que el problema disminuya si se separan físicamente los datos, por ejemplo dejando espacio adicional entre contadores o asignando a cada hilo bloques más grandes y menos entremezclados de memoria.

Otro concepto clave es NUMA, sigla de Non-Uniform Memory Access. En este tipo de arquitectura, todos los núcleos pueden acceder a toda la memoria, pero no con la misma latencia. Cada procesador o grupo de núcleos tiene memoria local de acceso más rápido y memoria remota de acceso más costoso. Por ese motivo, en plataformas NUMA la ubicación de los datos influye directamente sobre el rendimiento.

También aquí conviene pensar en un ejemplo. Si un proceso o un conjunto de hilos corre principalmente en un socket del sistema, pero los datos que utiliza fueron reservados en memoria asociada a otro socket, buena parte de los accesos serán remotos. El programa puede seguir funcionando sin errores, pero con tiempos sensiblemente peores que los esperados, porque una parte importante del costo se desplaza hacia la comunicación con memoria no local.

En la práctica, una situación NUMA suele sospecharse cuando el rendimiento cambia de manera notable según dónde se ejecuten los hilos o cómo se distribuyan los datos, aun manteniendo el mismo algoritmo. Si una implementación mejora al fijar afinidad de hilos, al inicializar datos desde el mismo conjunto de núcleos que luego los usa o al reducir accesos remotos, es razonable pensar que la arquitectura NUMA está influyendo sobre el resultado. En este tipo de plataformas, no alcanza con repartir tareas: también conviene preguntarse cerca de qué procesador quedaron ubicados los datos.

False sharing y NUMA muestran una idea central: el paralelismo real no depende solo de repartir trabajo, sino también de cómo ese trabajo interactúa con la memoria física.

3.6. Speed-up

El speed-up mide cuánto mejora el tiempo de ejecución de un programa al pasar de una versión secuencial a una versión paralela. Si se llama T_s al tiempo secuencial y T_p al tiempo paralelo usando p procesadores, entonces:

$$S(p) = \frac{T_s}{T_p}$$

Si una implementación secuencial tarda 24 segundos y la paralela 6 segundos, el speed-up es 4. Este valor indica que la versión paralela ejecuta el trabajo cuatro veces más rápido que la secuencial de referencia.

Sin embargo, el speed-up rara vez es lineal. En un escenario ideal, duplicar la cantidad de procesadores duplicaría la velocidad. En la práctica aparecen costos de creación de tareas, comunicación, sincronización, acceso a memoria y balanceo desigual de carga. Por ese motivo, un aumento en p no garantiza un crecimiento proporcional de $S(p)$.

3.7. Eficiencia

La eficiencia relaciona el speed-up con la cantidad de procesadores utilizados. Permite estimar cuánto del paralelismo disponible se está aprovechando efectivamente.

$$E(p) = \frac{S(p)}{p}$$

En este libro se expresará siempre en porcentaje. Por lo tanto, conviene escribir:

$$E(p) = \frac{S(p)}{p} \times 100$$

En condiciones normales, una eficiencia del 100 % representa un escalamiento lineal ideal. Valores menores indican que una parte del potencial paralelo se pierde en sobrecargas o en secciones no paralelizables.

$$E(4) = \frac{5}{4} \times 100 = 125\,\%$$

Este resultado indica un caso de eficiencia superlineal. Puede ocurrir ocasionalmente cuando la versión paralela no solo reparte trabajo, sino que además aprovecha mejor la jerarquía de memoria, reduce fallos de caché, utiliza vectorización o reorganiza los datos de una manera más

favorable que la versión secuencial. Por ese motivo, una eficiencia superior al 100 % no debe interpretarse automáticamente como un error, pero sí exige revisar con cuidado qué cambió entre ambas versiones.

3.8. Un ejemplo numérico de escalamiento

La siguiente tabla muestra un caso hipotético de escalamiento fuerte para un programa cuyo tiempo secuencial es 100 segundos.

Procesadores p	Tiempo paralelo T_p	Speed-up $S(p)$	Eficiencia $E(p)$
1	100 s	1.00	100 %
2	55 s	1.82	91 %
4	32 s	3.13	78 %
8	22 s	4.55	57 %
16	18 s	5.56	35 %

La tabla permite observar dos fenómenos. En primer lugar, el tiempo sigue disminuyendo al agregar procesadores. En segundo lugar, la eficiencia cae de manera sostenida. Esto muestra que la implementación mejora, pero lo hace con rendimientos decrecientes. Justamente esa diferencia entre acelerar y escalar bien es uno de los ejes centrales del análisis de performance.

3.9. Escalamiento fuerte y escalamiento débil

Antes de introducir las leyes clásicas de escalabilidad, conviene distinguir dos formas de plantear los experimentos. En el escalamiento fuerte, el tamaño del problema se mantiene fijo y se aumenta la cantidad de procesadores. La pregunta central es cuánto más rápido puede resolverse la misma tarea al agregar recursos. En este escenario, la fracción secuencial y los costos de coordinación suelen volverse cada vez más visibles.

En el escalamiento débil, en cambio, el tamaño del problema crece junto con la cantidad de procesadores. La pregunta ya no es solo si una tarea fija termina antes, sino si el sistema permite resolver problemas más grandes en tiempos razonables. Esta distinción ayuda a interpretar por

qué una misma plataforma puede mostrar límites claros en un experimento de tamaño fijo y, aun así, resultar útil cuando el objetivo es aumentar la escala del problema.

3.10. Ley de Amdahl

La ley de Amdahl formaliza el límite del speed-up cuando una parte del programa debe ejecutarse en forma secuencial. Si se llama α a la fracción secuencial del programa, el speed-up máximo con p procesadores se expresa como:

$$S_A(p) = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

Si el 10 % del programa es inevitablemente secuencial, es decir $\alpha = 0.1$, con 8 procesadores se obtiene:

$$S_A(8) = \frac{1}{0.1 + \frac{0.9}{8}} = \frac{1}{0.2125} \approx 4.71$$

Incluso con una cantidad muy grande de procesadores, el límite teórico estaría dado por:

$$\lim_{p \to \infty} S_A(p) = \frac{1}{\alpha}$$

En este ejemplo, el máximo teórico sería 10. Esta ley es valiosa porque muestra que el cuello de botella no desaparece por agregar hardware: una fracción secuencial pequeña puede imponer un techo fuerte al rendimiento.

3.11. Ley de Gustafson

La ley de Gustafson ofrece otra perspectiva. En lugar de suponer un problema de tamaño fijo, plantea que al aumentar la cantidad de procesadores también puede crecer el tamaño del problema resuelto en un tiempo razonable. Su formulación habitual es:

$$S_G(p) = p - \alpha(p-1)$$

Si nuevamente se toma $\alpha = 0.1$ y $p = 8$:

$$S_G(8) = 8 - 0.1(7) = 7.3$$

Mientras Amdahl enfatiza los límites del escalamiento con tamaño fijo, Gustafson ayuda a pensar por qué el paralelismo sigue siendo útil cuando el objetivo no es solo acelerar una tarea pequeña, sino resolver problemas más grandes en tiempos aceptables.

3.12. Memory bound y compute bound

No todos los programas están limitados por el mismo recurso. Un problema compute bound dedica la mayor parte del tiempo al cálculo aritmético. En estos casos, disponer de más capacidad de cómputo suele traducirse en mejoras significativas.

Un problema memory bound, en cambio, está dominado por el costo de mover datos entre memoria y procesadores. Aquí el cuello de botella no está en la cantidad de operaciones, sino en la velocidad con que pueden leerse y escribirse datos.

Una suma simple sobre un vector muy grande suele acercarse a un comportamiento memory bound: cada elemento requiere poco cálculo y mucho movimiento de datos. En cambio, la multiplicación densa de matrices tiende a ser más compute bound, porque reutiliza datos y realiza muchas operaciones por cada acceso a memoria. Esta diferencia ayuda a entender por qué algunas tareas escalan mejor que otras, aun cuando ambas estén correctamente paralelizadas.

Una comparación mínima en Python permite fijar esta diferencia:

```
total = 0.0
for value in values:
    total += value
```

En este caso, cada iteración hace poco cálculo por cada dato leído. El costo de acceder a memoria pesa mucho en el tiempo total.

```
for i in range(n):
    for j in range(n):
        for k in range(n):
            c[i][j] += a[i][k] * b[k][j]
```

Aquí la situación es distinta: cada acceso a los datos participa en muchas operaciones aritméticas. Aunque este ejemplo introductorio no agota el análisis, ayuda a ver por qué una tarea puede estar más limitada por memoria o más limitada por cómputo.

3.13. Modelo Roofline

El modelo Roofline ofrece una forma sintética de relacionar capacidad de cómputo y ancho de banda de memoria. Su idea central es que el rendimiento máximo de un programa queda limitado por dos techos: uno dado por la potencia de cálculo del procesador y otro por la velocidad con que los datos pueden llegar desde memoria.

En términos introductorios, el modelo permite distinguir si una implementación está frenada por cómputo o por memoria. Si la intensidad aritmética del programa es baja, es probable que el ancho de banda sea el factor dominante. Si es alta, el límite puede pasar a ser la capacidad de cálculo. Este marco será útil más adelante para interpretar resultados experimentales de CPU, vectorización y GPU.

3.14. Cierre de la unidad

Con todos estos elementos, resulta más claro por qué una implementación paralela puede comportarse por debajo de lo esperado. El problema puede estar en una fracción secuencial importante, en exceso de sincronización, en mala localidad de caché, en false sharing, en accesos remotos sobre arquitectura NUMA o en saturación del ancho de banda de memoria.

Por ese motivo, medir no consiste solo en registrar tiempos. También implica interpretar qué parte de la arquitectura está imponiendo el límite. Ese vínculo entre estructura del hardware y rendimiento observado es el núcleo del análisis de performance en sistemas paralelos.

Este capítulo presentó una base más rigurosa para evaluar plataformas paralelas. A partir de ahora ya no alcanza con afirmar que un programa usa varios hilos o varios procesos: también conviene preguntar sobre qué arquitectura corre, qué costos de memoria introduce y hasta dónde puede escalar razonablemente.

En el próximo capítulo se estudiarán modelos de programación paralela. Esa transición permitirá pasar desde la arquitectura y las métricas hacia estrategias de diseño concretas para organizar tareas, datos y comunicación.

3.15. Ejercicios del capítulo

- Describa las categorías principales de la taxonomía de Flynn y señale en qué casos resultan más relevantes SIMD y MIMD.
- Explique la diferencia entre memoria compartida, memoria distribuida y modelo híbrido.
- Defina localidad temporal y localidad espacial.
- Explique con sus palabras qué problema intenta describir la ley de Amdahl.
- Distinga problemas memory bound y compute bound.
- Explique por qué el modelo Roofline ayuda a interpretar la diferencia entre una implementación limitada por memoria y otra limitada por cómputo.
- Calcule el speed-up de un programa que tarda 24 segundos en forma secuencial y 6 segundos en forma paralela.
- Calcule la eficiencia del caso anterior si se usaron 4 procesadores.
- Suponga que la fracción secuencial de un programa es 0.2. Calcule el speed-up máximo teórico según Amdahl para 8 procesadores.
- Interprete la siguiente situación: al pasar de 8 a 16 procesadores, el tiempo baja poco y la eficiencia cae con fuerza.
- Proponga una situación en la que el rendimiento de un programa paralelo pueda verse afectado por accesos remotos en una arquitectura NUMA o por false sharing, y explique brevemente cuál sería el problema.

4 Modelos de programación paralela

Disponer de hardware paralelo no garantiza por sí mismo una solución eficiente. También es necesario decidir cómo descomponer el problema, cómo distribuir tareas y cómo coordinar resultados. Los modelos de programación paralela ofrecen justamente marcos de organización para pensar estas decisiones.

4.1. Objetivos del capítulo

- presentar algunos paradigmas clásicos de programación paralela;
- describir el tipo de problemas para los que cada modelo puede resultar útil;
- vincular cada paradigma con las nociones de comunicación, sincronización y distribución del trabajo.

4.2. Por qué trabajar con modelos

Los modelos de programación paralela funcionan como esquemas conceptuales y prácticos. Permiten reconocer patrones recurrentes en problemas distintos y facilitan la elección de una estrategia de implementación. En lugar de empezar desde cero ante cada desafío, conviene apoyarse en estructuras conocidas que ya organizan la coordinación entre procesos o tareas.

Antes de revisar cada paradigma, conviene identificar algunas preguntas guía:

- ¿el problema puede dividirse en subtareas relativamente independientes o requiere una coordinación central fuerte?;
- ¿el trabajo se parece más a una secuencia de etapas, a una descomposición recursiva o a una aplicación repetida de la misma operación sobre muchos datos?;

- ¿la comunicación entre tareas será frecuente o esporádica?;
- ¿interesa más maximizar rendimiento, simplificar implementación o facilitar escalabilidad distribuida?;
- ¿el problema tiene una estructura estable y regular, o cambia dinámicamente durante la ejecución?

Estas preguntas no producen una respuesta automática, pero permiten justificar mejor por qué un modelo resulta más adecuado que otro.

4.3. Modelo coordinador-trabajadores

El modelo coordinador-trabajadores organiza la ejecución a partir de un proceso principal que distribuye trabajo a varios procesos secundarios. En la bibliografía también puede aparecer como modelo maestro-trabajadores, o bajo la denominación histórica master-slave. En este libro se utilizará la expresión coordinador-trabajadores porque describe con claridad la relación funcional entre las partes: un componente coordina, asigna tareas y reúne resultados, mientras que los trabajadores ejecutan las unidades de trabajo recibidas.

Este modelo es útil cuando las tareas son muchas, relativamente similares entre sí y pueden asignarse desde un punto central sin excesivo costo. Un caso típico es el procesamiento por lotes de archivos, imágenes o registros, donde un proceso coordinador reparte unidades de trabajo independientes a varios workers.

Puede pensarse en la conversión de un conjunto grande de imágenes. El proceso coordinador mantiene la lista de archivos pendientes y asigna una imagen a cada worker disponible. Cada worker aplica la misma transformación, por ejemplo cambiar formato o reducir tamaño, y luego devuelve el resultado o informa que terminó. El coordinador continúa repartiendo trabajo hasta completar todo el lote.

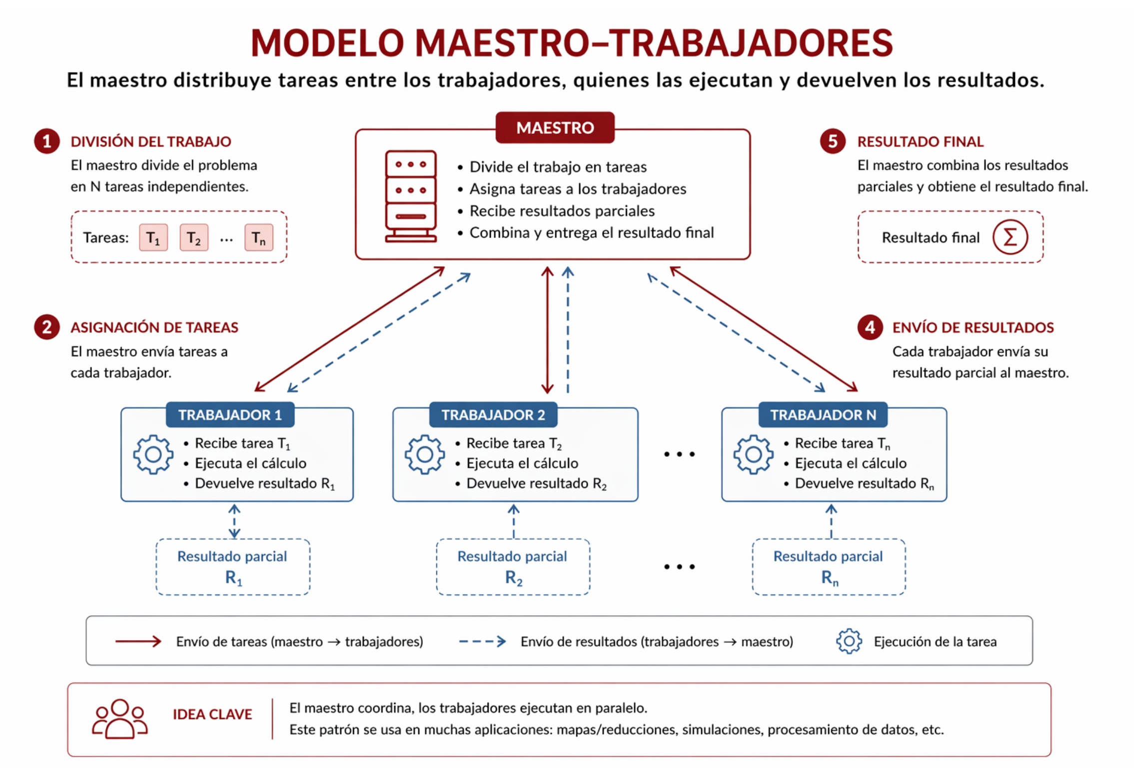

Figura 4.1: Esquema del modelo maestro-trabajadores para repartir tareas y reunir resultados.

Un esquema mínimo de pseudocódigo puede escribirse así:

```
funcion coordinador(tareas, workers):
    pendientes = copiar(tareas)

    mientras haya_tareas(pendientes):
        para cada worker disponible en workers:
            tarea = tomar_siguiente(pendientes)
            enviar(worker, tarea)

        recibir_resultados()
```

Su ventaja principal es la claridad organizativa. Su riesgo principal es que el coordinador se convierta en cuello de botella o en punto único de falla si concentra demasiadas decisiones o demasiada comunicación.

4.4. Divide and conquer

El modelo divide and conquer resuelve un problema descomponiéndolo en subproblemas más pequeños. Cada parte se resuelve por separado y luego los resultados parciales se combinan para obtener la solución final. En muchas implementaciones, esta descomposición se aplica de manera recursiva, pero la idea central del paradigma es la partición del problema y la posterior recomposición. Ejemplos clásicos son QuickSort y ciertas estrategias de multiplicación de matrices.

Este paradigma es útil cuando el problema admite una descomposición natural y relativamente equilibrada.

Un ejemplo más simple es la suma de un vector dividido en bloques. En lugar de recorrer todos los elementos en una sola secuencia, el conjunto de datos puede separarse en partes de tamaño semejante. Cada parte se procesa por separado y, al final, los subtotales se combinan para obtener el resultado global. Aunque muchas implementaciones de este paradigma usan recursión, la idea central puede entenderse también como una división del problema en partes comparables que luego se recomponen.

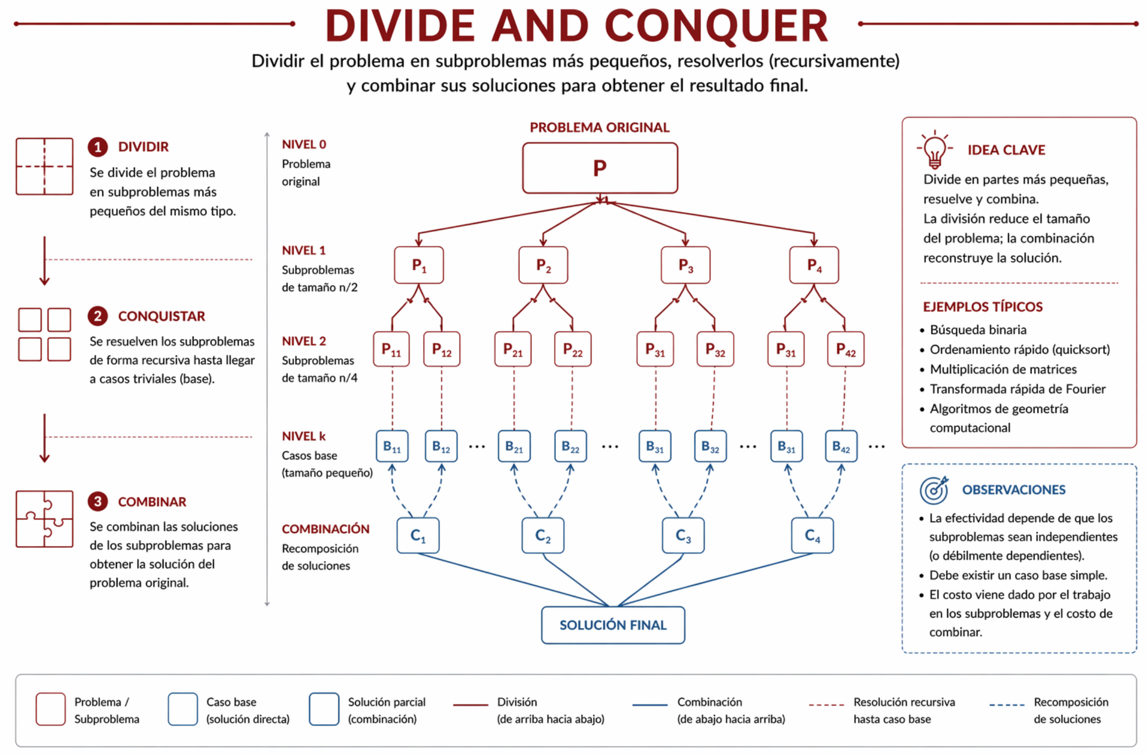

Figura 4.2: Patrón divide and conquer: partición del problema, resolución de subproblemas y recomposición final.

Un esquema mínimo de pseudocódigo puede escribirse así:

```
funcion suma_por_bloques(datos, cantidad_bloques):
    bloques = particionar(datos, cantidad_bloques)
    subtotales = []

    para cada bloque en bloques:
        subtotales.agregar(sumar(bloque))

    return sumar(subtotales)
```

En una implementación paralela, cada bloque podría procesarse al mismo tiempo y la fase final solo tendría que combinar los subtotales obtenidos.

Conviene usar este modelo cuando la división del problema produce subproblemas de tamaño

semejante y con poca dependencia entre sí. Si las particiones quedan muy desbalanceadas o si la fase de combinación resulta costosa, la ventaja del modelo disminuye.

4.5. Pipelining

El modelo de pipelining divide el procesamiento en etapas. Cada etapa realiza una parte del trabajo y pasa el resultado a la siguiente. Mientras una etapa procesa un nuevo elemento, otra puede estar terminando el elemento anterior. Se genera así un flujo continuo, similar a una línea de producción.

Su principal desafío consiste en gestionar correctamente las dependencias entre etapas para evitar cuellos de botella.

Este paradigma resulta especialmente útil cuando el trabajo puede describirse como una secuencia estable de transformaciones. Un caso reconocible es el procesamiento de datos provenientes de sensores, video o registros: una etapa lee, otra limpia, otra transforma y una última almacena o visualiza resultados.

Un caso simple aparece en el procesamiento de formularios enviados desde un sitio web. Una primera etapa recibe los datos, una segunda valida los campos, una tercera normaliza el formato y una cuarta guarda la información en una base de datos. Mientras un formulario se encuentra en la etapa de almacenamiento, otro puede estar siendo validado y un tercero puede estar ingresando al sistema.

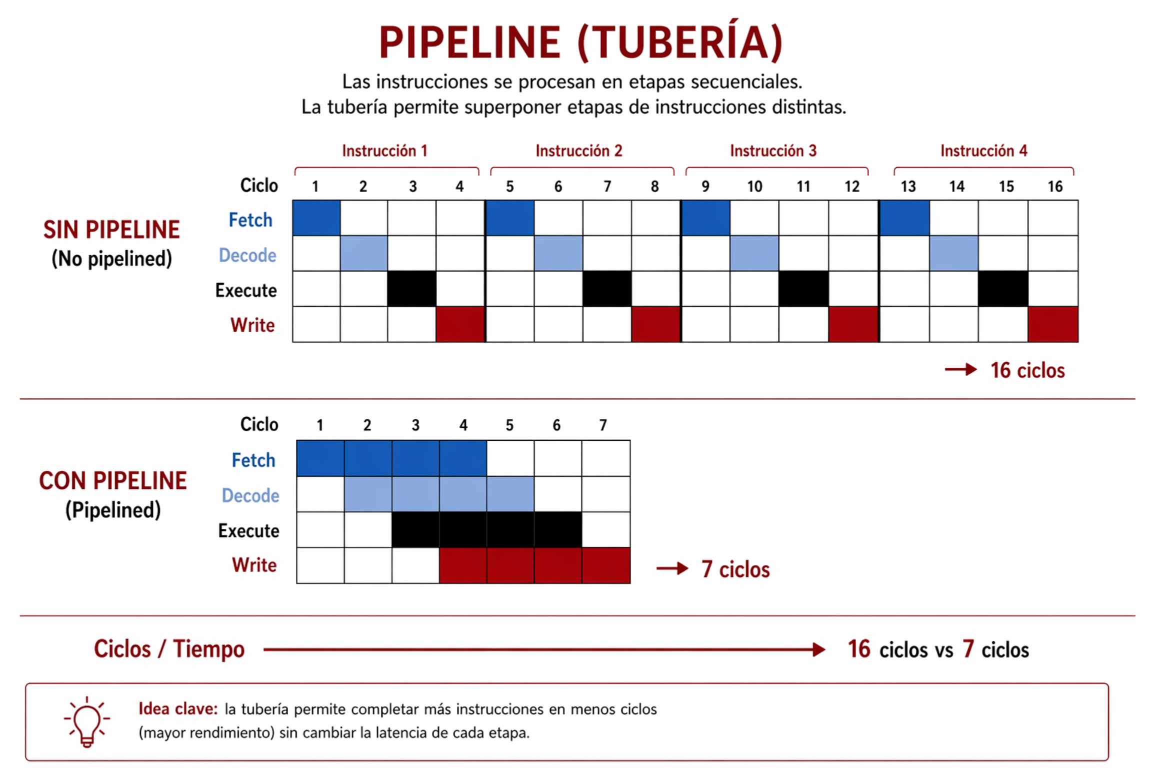

Figura 4.3: Modelo pipeline con etapas consecutivas que procesan datos en flujo.

Un esquema mínimo de pseudocódigo puede representarse de este modo:

```
funcion pipeline(formulario):
    datos = recibir(formulario)
    datos_validados = validar(datos)
    datos_normalizados = normalizar(datos_validados)
    guardar(datos_normalizados)
```

4.6. MapReduce

MapReduce organiza el cálculo en dos grandes fases: una etapa de mapeo, que aplica una operación sobre múltiples elementos, y una etapa de reducción, que combina los resultados parciales. Se trata de un modelo muy asociado al procesamiento distribuido de grandes volúmenes de datos y a herramientas como Hadoop o Apache Spark.

Aunque en este libro se presenta de forma introductoria, su valor conceptual es grande porque muestra una estrategia de paralelización centrada en datos y agregación de resultados.

El conteo de palabras en una gran colección de documentos permite verlo con claridad. La fase de map recorre cada documento y produce pares del tipo palabra, 1 cada vez que encuentra una aparición. Luego, la fase de reduce agrupa todas las ocurrencias de una misma palabra y suma esos valores parciales para obtener el total final. Más allá de la herramienta concreta, lo importante es observar que el modelo organiza con claridad la relación entre paralelismo de datos y agregación.

Conviene usar este enfoque cuando la tarea puede expresarse como una operación repetida sobre muchas entradas independientes seguida de una combinación de resultados. No suele ser la mejor opción cuando existen dependencias complejas, interacción frecuente entre tareas o estructuras de datos muy cambiantes.

4.7. Modelo de actores

En el modelo de actores, cada componente se entiende como una entidad independiente con estado propio, que se comunica con otros actores mediante mensajes. No se basa necesariamente en una jerarquía fija, sino en interacciones coordinadas entre componentes autónomos. Se trata, ante todo, de un modelo de concurrencia y organización del sistema. Sin embargo, se incluye en este capítulo porque ofrece una forma de estructurar problemas en los que muchas actividades pueden avanzar en paralelo sin compartir memoria directamente.

Un caso reconocible es una plataforma compuesta por servicios que reciben eventos, procesan pedidos y envían respuestas sin compartir memoria directamente. Cada actor mantiene su estado local y responde a mensajes, lo que favorece desacoplamiento y escalabilidad.

Un ejemplo posible es un sistema de simulación en el que cada partícula o entidad del modelo se representa como un actor. Cada una mantiene su propio estado y envía mensajes a otras cuando necesita informar un cambio, por ejemplo una colisión, una proximidad o una modificación de velocidad. De ese modo, la simulación no depende de una memoria compartida central para coordinar todas las interacciones. Si muchas entidades evolucionan al mismo tiempo, distintos actores pueden procesar mensajes de manera concurrente, y en muchos entornos esa concurrencia también puede traducirse en ejecución paralela.

Este modelo conviene cuando interesa reducir acoplamiento entre componentes y tolerar mejor la complejidad de la concurrencia. No siempre su objetivo principal es acelerar un cálculo del mismo modo que divide and conquer o MapReduce, pero sí puede servir para organizar sistemas donde muchas tareas o eventos deben resolverse de manera simultánea. A cambio, obliga a pensar cuidadosamente el diseño de mensajes, la coordinación entre actores y el seguimiento del estado distribuido.

4.8. Comparación conceptual entre modelos

Si se observa el conjunto, coordinador-trabajadores y divide and conquer son modelos orientados con más claridad al reparto del trabajo. Pipelining organiza una cadena de etapas. MapReduce enfatiza transformación de datos y agregación. Actores, en cambio, privilegia la coordinación entre componentes autónomos y por eso queda más cerca de los modelos de concurrencia que de los esquemas clásicos de paralelización de datos.

Ningún modelo resuelve por sí solo todos los problemas. En sistemas reales es frecuente combinar varios. Por ejemplo, un clúster puede usar un esquema coordinador-trabajadores para distribuir bloques de datos y, dentro de cada bloque, aplicar una estrategia divide and conquer o un pipeline de procesamiento.

La siguiente tabla resume el rasgo dominante de cada modelo y el tipo de problema en el que suele resultar más útil.

Modelo	Idea central	Conviene usarlo cuando	Caso reconocible
Coordinador-trabajadores	un coordinador reparte trabajo y reúne resultados	hay una lógica central clara de asignación y control	procesamiento de lotes independientes
Divide and conquer	el problema se divide en partes y luego se combinan resultados	la tarea admite particiones naturales y balanceadas	suma por bloques, ordenamiento, matrices
Pipelining	el trabajo se organiza en etapas consecutivas	los datos fluyen por fases con operaciones diferenciadas	procesamiento de imágenes o streaming

Modelo	Idea central	Conviene usarlo cuando	Caso reconocible
MapReduce	se aplica una función a muchos datos y luego se agregan resultados	hay grandes volúmenes de datos con operaciones homogéneas	conteo de palabras, agregaciones masivas
Actores	componentes autónomos intercambian mensajes	el sistema requiere desacoplamiento, concurrencia y evolución dinámica	servicios distribuidos o sistemas reactivos

4.9. Cierre de la unidad

Este capítulo subraya un punto decisivo: programar algoritmos paralelos suele ser más difícil que programar algoritmos secuenciales. La sincronización, la comunicación entre procesos y la necesidad de evitar errores como esperas innecesarias o incoherencias de datos introducen complejidad adicional.

Por ese motivo, los modelos no solo ayudan a organizar soluciones, sino también a razonar sobre sus riesgos y limitaciones.

En este sentido, elegir un modelo adecuado no es solo una cuestión de estilo. Una buena elección puede reducir comunicación, simplificar sincronización y mejorar la escalabilidad. Una mala elección puede producir cuellos de botella, tareas desbalanceadas o una implementación innecesariamente difícil de mantener.

Con estos paradigmas ya es posible pasar de la discusión arquitectónica a una discusión de diseño. El capítulo mostró que programar en paralelo no consiste solo en lanzar hilos o procesos, sino en decidir cómo se estructura el trabajo y cómo se relacionan las tareas entre sí.

En el próximo capítulo estas ideas se traducirán a APIs clásicas de programación paralela. Allí se verá cómo modelos como memoria compartida, memoria distribuida y reparto de tareas se concretan en herramientas específicas de implementación.

4.10. Ejercicios del capítulo

- Describa las características básicas del modelo coordinador-trabajadores.
- Explique en qué se diferencia un pipeline de una estrategia divide and conquer.
- Indique qué papel cumplen los mensajes en el modelo de actores.
- Justifique por qué MapReduce se asocia especialmente con problemas de gran volumen de datos.
- Proponga un caso en el que coordinador-trabajadores sea una mejor elección que actores y explique por qué.
- Analice una situación de procesamiento de datos o de simulación y justifique qué modelo elegiría entre coordinador-trabajadores, divide and conquer, pipeline, MapReduce o actores.

5 APIs de programación paralela

Luego de estudiar modelos de diseño, conviene presentar algunas APIs clásicas que han tenido un papel relevante en la programación paralela. Estas herramientas no solo permiten implementar soluciones, sino que también representan enfoques distintos según el tipo de arquitectura disponible.

Aunque en este libro el trabajo aplicado se desarrolla principalmente con Python, conviene conocer estas APIs porque siguen siendo una referencia importante para entender cómo se programa cerca del hardware y por qué muchas herramientas de más alto nivel toman ideas de estos modelos.

5.1. Objetivos del capítulo

- introducir Pthreads, OpenMP y MPI como referencias clásicas del área;
- vincular cada API con su ámbito de aplicación principal;
- reconocer diferencias entre memoria compartida, memoria distribuida e implementaciones híbridas;
- comparar mecanismos de comunicación, sincronización y nivel de abstracción;
- mostrar decisiones concretas de diseño a partir de un mismo problema.

5.2. Por qué estudiar estas APIs

Pthreads, OpenMP y MPI forman una tríada clásica porque condensan tres modos distintos de pensar la programación paralela. Pthreads expone control fino sobre hilos y sincronización. OpenMP simplifica la incorporación de paralelismo en memoria compartida mediante directivas. MPI organiza la cooperación entre procesos independientes que intercambian mensajes.

5.3. Pthreads

Pthreads es una API POSIX orientada a la programación con hilos en sistemas de memoria compartida. Su importancia histórica radica en ofrecer una forma estandarizada de crear, coordinar y sincronizar hilos de ejecución. En lenguajes como C, C++ y Fortran ha sido una herramienta importante para trabajar con paralelismo a bajo nivel.

Pthreads conviene cuando se necesita control detallado sobre la creación de hilos, la asignación de trabajo y la sincronización. Ese control fino es una fortaleza, pero también aumenta la complejidad. Un mutex permite asegurar exclusión mutua sobre una sección crítica, es decir, impedir que dos hilos modifiquen al mismo tiempo un mismo dato compartido. Las variables compartidas son justamente los datos accesibles por varios hilos, y por eso deben protegerse o coordinarse con cuidado. Las barreras obligan a que varios hilos esperen hasta alcanzar un mismo punto de ejecución antes de continuar, mientras que las variables de condición permiten suspender un hilo hasta que se cumpla cierto estado o evento.

Trabajar con estos mecanismos exige mucha atención porque un uso incorrecto puede producir race conditions, es decir, resultados que dependen del orden imprevisible en que acceden los hilos a los datos. También puede generar bloqueos, por ejemplo cuando dos hilos quedan esperando indefinidamente recursos o señales del otro. En términos de aprendizaje, Pthreads permite ver con claridad que paralelizar no consiste solo en repartir trabajo, sino también en coordinar accesos, esperas y dependencias de manera segura.

En términos de diseño, Pthreads resulta útil para entender qué significa realmente paralelizar en memoria compartida sin ayudas adicionales del compilador.

Un ejemplo mínimo permite observar la estructura básica: crear hilos, ejecutar una función y esperar su finalización con `pthread_join`.

```
#include <pthread.h>
#include <stdio.h>

void* worker(void* arg) {
    int id = *(int*)arg;
    printf("Worker %d\n", id);
    return NULL;
```

```
}

int main() {
    pthread_t threads[2];
    int ids[2] = {0, 1};

    for (int i = 0; i < 2; i++) {
        pthread_create(&threads[i], NULL, worker, &ids[i]);
    }

    for (int i = 0; i < 2; i++) {
        pthread_join(threads[i], NULL);
    }

    return 0;
}
```

El ejemplo no realiza todavía un cálculo paralelo relevante, pero muestra el patrón esencial: cada hilo ejecuta una función y el programa principal espera a que todos terminen antes de continuar. En un problema real, además de crear hilos, habría que decidir qué datos recibe cada uno y cómo se protegen los datos compartidos.

5.4. OpenMP

OpenMP es una API orientada a memoria compartida que simplifica la incorporación de paralelismo mediante directivas del compilador. Se usa principalmente con C, C++ y Fortran, y resulta adecuada para paralelizar bucles y secciones de código en equipos con múltiples núcleos.

En el recorrido del libro, OpenMP cumple una doble función: por un lado, es una referencia estándar del área; por otro, sirve como antecedente conceptual para entender herramientas en Python que ofrecen estrategias semejantes.

Su ventaja principal es que permite agregar paralelismo incremental sobre código secuencial existente. En muchos casos basta con identificar un bucle independiente y marcarlo como región

paralela. OpenMP ofrece menos control explícito que Pthreads porque el programador no suele crear y coordinar hilos uno por uno, sino que describe regiones paralelas, bucles o reducciones y deja que el compilador y el entorno de ejecución decidan cómo materializar esa estrategia. Dicho de otro modo, en Pthreads se programa de manera directa la lógica de creación, sincronización y reparto; en OpenMP se declara qué partes pueden ejecutarse en paralelo y con qué restricciones.

Por ese motivo, OpenMP se apoya fuertemente en el compilador y en la semántica de sus directivas. Una instrucción como `#pragma omp parallel for` no implementa por sí misma el paralelismo, sino que le indica al compilador cómo transformar ese bloque para ejecutarlo con varios hilos. Esa mediación simplifica mucho el trabajo y reduce código de sincronización explícita, pero también implica aceptar menor control fino sobre decisiones como la organización exacta de los hilos o ciertos detalles del reparto de trabajo.

Un ejemplo mínimo en C permite ver esa lógica:

```c
#include <omp.h>
#include <stdio.h>

int main() {
    int total = 0;

    #pragma omp parallel for reduction(+:total)
    for (int i = 0; i < 100; i++) {
        total += i;
    }

    printf("Total: %d\n", total);
    return 0;
}
```

En este fragmento, la directiva `#pragma omp parallel for` indica que las iteraciones del bucle pueden repartirse entre varios hilos. La cláusula `reduction` resuelve de forma segura la acumulación parcial. El interés pedagógico del ejemplo no está en el detalle sintáctico, sino en mostrar que OpenMP expresa paralelismo de forma declarativa y compacta.

Para compilar un programa con OpenMP suele ser necesario activar explícitamente el soporte del compilador. Con `gcc`, por ejemplo, una compilación básica puede escribirse así:

```
gcc -fopenmp ejemplo_openmp.c -o ejemplo_openmp
```

El comando exacto puede variar según el compilador y el sistema operativo, pero la idea importante es que las directivas de OpenMP requieren un compilador y un entorno de ejecución capaces de interpretarlas.

5.5. MPI

MPI, sigla de Message Passing Interface, es una API pensada para sistemas distribuidos. Cada proceso es independiente y la coordinación se realiza mediante envío y recepción explícita de mensajes. Esto la vuelve especialmente relevante para clústeres y contextos de computación científica.

En este libro se incluyen ejemplos introductorios con `mpi4py`, lo que permite acercar este modelo al entorno Python sin perder la lógica central del intercambio de mensajes entre procesos.

MPI es útil cuando los procesos no comparten memoria y la distribución de datos debe explicitarse. Esa es su principal diferencia con Pthreads y OpenMP: aquí la comunicación no está implícita en variables compartidas, sino que debe diseñarse conscientemente mediante primitivas como `send`, `recv`, `scatter`, `gather`, `broadcast` y `reduce`.

En términos conceptuales, algunas de las operaciones más importantes son estas:

- `broadcast`: un proceso envía el mismo dato a todos los demás;
- `scatter`: un proceso reparte fragmentos distintos de un conjunto de datos;
- `gather`: varios procesos devuelven resultados parciales a un proceso raíz;
- `reduce`: varios resultados parciales se combinan mediante una operación como suma, máximo o mínimo.

Un ejemplo mínimo con `mpi4py` muestra la lógica de `scatter` y `gather`:

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = [10, 20, 30, 40]
else:
    data = None

value = comm.scatter(data, root=0)
partial = value * 2
results = comm.gather(partial, root=0)

if rank == 0:
    print(results)
```

Aquí el proceso raíz distribuye cuatro valores, cada proceso calcula un resultado local y luego el conjunto de resultados vuelve al proceso raíz. Se trata de un patrón muy frecuente en problemas distribuidos.

También es habitual combinar `scatter` con `reduce` cuando interesa repartir datos y obtener directamente un agregado final sin reconstruir toda la colección intermedia.

Un programa con MPI debe ejecutarse lanzando varios procesos. En un entorno con `mpi4py`, una forma típica de hacerlo es:

```bash
mpiexec -n 4 python ejemplo_mpi.py
```

La opción `-n 4` indica que se crearán cuatro procesos. Cada uno ejecutará el mismo programa, pero tendrá un `rank` distinto, lo que permite asignar responsabilidades diferentes dentro de una misma ejecución distribuida.

5.6. Diferencias y ámbitos de aplicación

Pthreads y OpenMP se asocian, en general, a sistemas de memoria compartida. MPI, en cambio, responde de forma más natural a escenarios de memoria distribuida. También es posible combinar enfoques, por ejemplo mediante OpenMP más MPI, dando lugar a modelos híbridos.

Reconocer estas diferencias permite evitar un error frecuente: pensar que existe una única forma de paralelizar. En realidad, la arquitectura disponible condiciona de manera fuerte qué herramientas resultan más adecuadas.

En una estación de trabajo multicore, Pthreads u OpenMP suelen ser elecciones naturales. En un clúster donde cada nodo tiene memoria propia, MPI se vuelve la opción más razonable. En un sistema híbrido, una estrategia común consiste en usar MPI para repartir trabajo entre nodos y OpenMP dentro de cada nodo para explotar los cores locales.

5.7. Vista comparativa

Antes de revisar cada API por separado, conviene ordenar sus diferencias principales en una única comparación.

API	Arquitectura más natural	Comunicación o sincronización principal	Nivel de abstracción	Dificultad relativa	Fortaleza principal	Riesgo principal	Uso típico
Pthreads	memoria compartida	acceso compartido a memoria, mutexes, joins y variables de condición	bajo	alta	control fino	errores de concurrencia	control detallado de hilos

API	Arquitectura más natural	Comunicación o sincronización principal	Nivel de abstracción	Dificultad relativa	Fortaleza principal	Riesgo principal	Uso típico
OpenMP	memoria compartida	regiones paralelas, barreras implícitas y reducciones	medio	media	rapidez para paralelizar código existente	uso ingenuo de directivas	paralelización rápida de bucles y secciones
MPI	memoria distribuida	envío y recepción explícita de mensajes entre procesos	medio a bajo	media a alta	escalabilidad distribuida	costo de comunicación y diseño de intercambio	clústeres y procesos distribuidos

Esta tabla permite advertir una diferencia central: no todas las APIs atacan el mismo problema. Elegir una herramienta adecuada depende primero de la arquitectura y luego del grado de control o simplicidad que se necesite. También ayuda a pasar de la taxonomía a la decisión práctica: no se trata solo de saber qué API existe, sino de entender qué costo de implementación y qué modelo mental exige cada una.

5.8. Multiplicación de matrices como caso de estudio

La multiplicación de matrices aparece de manera recurrente en este libro porque reúne varias razones pedagógicas y técnicas a la vez. Desde el punto de vista conceptual, permite ver con claridad cómo se descompone un problema, cómo se distribuyen filas, columnas o bloques y cómo se recomponen luego los resultados parciales. Desde el punto de vista del costo, se trata de una tarea suficientemente intensiva como para que las decisiones de paralelización, acceso a memoria

y sincronización tengan efectos visibles en el rendimiento. Además, posee gran versatilidad matemática: aparece en álgebra lineal, simulación numérica, gráficos por computadora, procesamiento de señales, aprendizaje automático y numerosos problemas científicos e ingenieriles. Por ese motivo, funciona como un ejemplo útil para conectar fundamentos teóricos, diseño de algoritmos y aplicaciones reales.

Desde el punto de vista del diseño, la multiplicación de matrices puede leerse de varias maneras:

- con Pthreads, varias filas o bloques pueden asignarse manualmente a hilos distintos;
- con OpenMP, un bucle externo puede paralelizarse con directivas del compilador;
- con MPI, las matrices pueden repartirse por bloques entre nodos y luego combinar resultados parciales.

En una versión distribuida conceptual, un proceso raíz podría repartir bloques de filas de la matriz `A`, enviar la matriz `B` completa o partes relevantes a otros procesos y luego recolectar los bloques de la matriz resultado `C`. La dificultad ya no está solo en el cálculo local, sino en decidir cómo dividir y comunicar datos con un costo razonable.

Un esquema mínimo con MPI puede resumirse así:

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

if rank == 0:
    blocks_of_a = split_rows(matrix_a, size)
else:
    blocks_of_a = None

local_a = comm.scatter(blocks_of_a, root=0)
full_b = comm.bcast(matrix_b, root=0)
local_c = local_a @ full_b
result_blocks = comm.gather(local_c, root=0)
```

El interés de este fragmento no está en los detalles de implementación, sino en la secuencia de diseño: repartir filas de `A`, difundir `B`, calcular un bloque local de `C` y reunir luego los resultados parciales. Esa estructura permite ver con claridad dónde aparece el costo de comunicación y dónde se concentra el cómputo local.

Este ejemplo permite ver que una misma tarea puede expresarse con APIs diferentes, pero la estrategia concreta cambia con la arquitectura y con el costo de sincronización o transferencia.

5.9. Cierre del capítulo

Conviene leer estas APIs no solo como herramientas aisladas, sino como respuestas a problemas de arquitectura diferentes. Pthreads enfatiza control fino sobre memoria compartida. OpenMP privilegia productividad en ese mismo entorno. MPI, en cambio, obliga a pensar datos, partición y comunicación como parte explícita del algoritmo.

Vistas en conjunto, estas APIs ayudan a formular mejor una pregunta de diseño: no cuál herramienta es mejor en abstracto, sino cuál se ajusta mejor a la arquitectura disponible, al tipo de problema y al nivel de complejidad aceptable.

En el próximo capítulo el foco estará puesto en las herramientas disponibles en Python para trabajar con estos problemas en un contexto de aprendizaje accesible.

5.10. Ejercicios del capítulo

- Explique qué tipo de arquitectura se asocia más naturalmente con Pthreads y OpenMP.
- Describa por qué MPI resulta importante en sistemas distribuidos.
- Utilice un ejemplo para explicar qué se entiende por un enfoque híbrido en programación paralela.
- Compare el tipo de control que ofrecen Pthreads y OpenMP sobre la ejecución paralela.
- Explique qué función cumplen `scatter`, `gather` y `reduce` en MPI.
- Proponga qué API sería más adecuada para un clúster de computadoras y justifique la respuesta.
- Describa qué dificultades podrían aparecer al paralelizar una multiplicación de matrices.

- Indique en qué situación OpenMP podría ser preferible a Pthreads en memoria compartida.
- Explique cómo podría combinarse MPI con OpenMP en un sistema híbrido.

Parte II

Herramientas y aplicaciones

6 Paralelismo en Python

En este libro, Python funciona como lenguaje de trabajo para experimentar con conceptos de programación paralela. La elección no implica desconocer la importancia de lenguajes como C, que siguen siendo fundamentales cuando se busca control fino sobre memoria, hilos y cercanía con el hardware. Sin embargo, para un recorrido introductorio conviene priorizar una herramienta que permita concentrarse primero en los conceptos centrales del paralelismo sin agregar toda la complejidad sintáctica y operativa de un lenguaje de más bajo nivel.

Esta decisión también responde a una consideración formativa y laboral. Python constituye una herramienta especialmente útil porque aparece con frecuencia en tareas de automatización, análisis de datos, desarrollo de servicios, integración de sistemas y procesamiento de información. Su presencia en contextos académicos, científicos y profesionales vuelve especialmente valioso usarlo como lenguaje de trabajo para introducir estrategias de paralelización sin exigir una especialización inmediata en desarrollo de sistemas de bajo nivel.

Además, Python ocupa hoy un lugar central en inteligencia artificial, aprendizaje automático, ciencia de datos y cómputo científico. Bibliotecas como NumPy, Numba, PyTorch, JAX o TensorFlow forman parte del ecosistema con el que actualmente se construyen desde prototipos experimentales hasta sistemas productivos. Por ese motivo, trabajar con Python en este libro no solo facilita la comprensión de los fundamentos, sino que también conecta esos fundamentos con herramientas y problemas frecuentes en contextos académicos y profesionales reales.

6.1. Objetivos del capítulo

- presentar herramientas de Python útiles para explorar paralelismo en CPU;
- comparar enfoques basados en hilos, procesos y compilación JIT;
- relacionar cada estrategia con el tipo de problema que conviene resolver;

- señalar las limitaciones del lenguaje y sus implicancias prácticas;
- introducir errores frecuentes de concurrencia y criterios básicos de diagnóstico.

6.2. Qué permite comparar Python

En el resto del libro, Python resulta especialmente útil porque permite contrastar estrategias diferentes sobre problemas similares sin cambiar de entorno de trabajo en cada paso. Con relativa facilidad puede pasarse de una versión secuencial a una versión con hilos, otra con procesos y una versión compilada con Numba. Esa continuidad vuelve más visible qué cambia realmente cuando se modifica la forma de expresar el cálculo y, más adelante, también facilita extender la misma lógica hacia otros niveles de abstracción.

Esta posibilidad comparativa tiene un valor didáctico importante. Ayuda a ver que el paralelismo no es una técnica única, sino un conjunto de decisiones sobre partición de datos, coordinación, costo de ejecución y nivel de abstracción. Justamente por eso Python funciona aquí menos como un fin en sí mismo que como una plataforma conveniente para observar, medir y contrastar estrategias distintas sobre una misma familia de problemas.

6.3. Herramientas centrales de este capítulo

Antes de entrar en ejemplos concretos, conviene ordenar qué herramientas ocuparán el centro del capítulo. Aquí el foco estará puesto sobre todo en `threading`, `multiprocessing`, pools o executors y Numba en CPU. Se trata de herramientas distintas, pero todas permiten examinar una misma pregunta: cómo pasar de una versión secuencial en Python a variantes más eficientes cuando el problema exige trabajo intensivo de CPU o una mejor organización del cálculo.

- `threading` y sus variantes permiten trabajar con concurrencia basada en hilos;
- `multiprocessing` crea procesos independientes y evita la restricción principal del GIL para tareas intensivas de CPU;
- Numba compila funciones Python y puede acercar el rendimiento al de implementaciones más cercanas al hardware sin abandonar el lenguaje de trabajo.

Más adelante se retomarán otras estrategias de optimización sobre arreglos y tensores, pero en este punto conviene concentrarse en estas herramientas porque constituyen una base importante para entender el paralelismo explícito en CPU.

6.4. Limitaciones de Python para paralelismo

Uno de los puntos centrales aquí es el Global Interpreter Lock, conocido como GIL. Se trata de una limitación relevante en la implementación tradicional de Python, ya que condiciona la ejecución simultánea de hilos dentro de un mismo intérprete. Por ese motivo, no siempre alcanza con crear múltiples threads para obtener una mejora real en tareas intensivas de CPU.

Conviene agregar una precisión importante. Desde Python 3.13 existe la posibilidad experimental de compilar una variante sin GIL, en el marco de los cambios recientes del intérprete. Sin embargo, esa opción todavía no constituye la modalidad estándar de uso y sigue en una etapa de adopción y maduración. Por ese motivo, en este libro se trabaja con la versión oficial y estable de Python tal como se utiliza de manera habitual, es decir, con GIL habilitado. Más adelante, a medida que estas variantes se consoliden, será razonable revisar este punto en futuras versiones del material.

En términos prácticos, esto significa que los hilos en Python suelen ser adecuados para tareas I/O-bound, como leer archivos, esperar respuestas de red o interactuar con dispositivos. En cambio, para tareas CPU-bound, como sumar grandes arreglos, procesar imágenes o calcular modelos, el GIL limita la ganancia esperable si se usan hilos convencionales.

Comprender esta limitación es fundamental para interpretar correctamente los resultados experimentales. Un programa con muchos threads puede parecer más complejo o más paralelo y, sin embargo, rendir igual o peor que una versión secuencial si el problema está dominado por cálculo puro. En cambio, cuando se trabaja con procesos independientes, cada uno dispone de su propio intérprete y puede ejecutar cálculo en paralelo sin quedar restringido por el mismo GIL, aunque a costa de introducir mayores costos de creación, serialización y comunicación.

6.5. Threads y procesos: qué cambia en Python y qué cambia respecto de C

En Python conviene distinguir con claridad entre threads y procesos porque no representan solo dos maneras de lanzar trabajo, sino también dos modelos distintos de memoria y de costo. Los threads comparten el mismo espacio de memoria del proceso y, dentro de Python, también comparten el mismo intérprete. Eso vuelve más simple el acceso a datos comunes, pero introduce la restricción del GIL en la implementación habitual del lenguaje. Los procesos, en cambio, tienen memoria separada y cada uno cuenta con su propio intérprete, de modo que pueden ejecutar cálculo en paralelo real sobre varios núcleos, aunque compartir datos entre ellos resulte más costoso.

Desde el punto de vista práctico, esta diferencia explica por qué los threads suelen convenir en tareas I/O-bound y los procesos en tareas CPU-bound. Si un thread pasa gran parte del tiempo esperando una respuesta de red, una lectura de disco o la llegada de datos desde otro sistema, el costo del GIL pesa menos y la concurrencia sigue siendo útil. Si el trabajo consiste en cálculo sostenido, la situación cambia: varios threads de Python pueden alternar sobre un mismo intérprete sin traducirse en una aceleración proporcional. Con procesos, en cambio, el paralelismo puede aprovechar mejor los cores disponibles, pero aparece un costo adicional al copiar, serializar o coordinar datos entre espacios de memoria separados.

La comparación con C ayuda a precisar todavía más esta idea. En C, bibliotecas como Pthreads también trabajan con hilos dentro de un mismo proceso y con memoria compartida, pero no existe una restricción equivalente al GIL del intérprete estándar de Python. Por ese motivo, en C los threads suelen ser una herramienta natural tanto para concurrencia como para paralelismo CPU-bound, siempre que la sincronización esté bien resuelta. Python conserva parte de esa lógica de memoria compartida cuando se usan threads, pero agrega una capa de ejecución interpretada que modifica el resultado esperado. Dicho de forma simple, en C la pregunta principal suele ser cómo sincronizar hilos sin errores; en Python también hay que preguntar si conviene usar hilos o si el problema exige pasar a procesos, vectorización o compilación.

Si en el futuro una variante sin GIL se vuelve la modalidad oficial y dominante de Python, esta distinción entre threads y process no perderá sentido, aunque probablemente cambie de énfasis. Los threads podrían ganar relevancia también para tareas CPU-bound, porque dejarían de arrastrar la restricción principal que hoy condiciona su rendimiento. Sin embargo, seguiría siendo importante

diferenciarlos de los procesos: ambos modelos no solo se distinguen por su capacidad de cómputo, sino también por el modo en que comparten memoria, se aíslan entre sí, gestionan fallos y asumen costos de coordinación. Dicho de otro modo, un Python sin GIL volvería menos tajante la oposición actual entre threads para entrada/salida y procesos para cálculo intensivo, pero no eliminaría la necesidad de elegir entre memoria compartida e intérpretes aislados según la estructura real del problema.

6.6. Patrones básicos para paralelizar loops en Python

Uno de los usos más comunes del paralelismo en Python consiste en transformar un loop secuencial en una ejecución repartida entre workers. Conviene, entonces, ordenar primero los patrones más habituales con los que suele plantearse esa transformación.

6.6.1. Creación manual de workers

La forma más explícita consiste en crear un hilo o un proceso por tarea. Este enfoque es pedagógicamente útil porque muestra con claridad qué significa lanzar trabajo independiente, iniciar workers y esperar su finalización.

Un patrón mínimo con hilos podría escribirse así:

```python
from threading import Thread


def task(value):
    print(value * value)


if __name__ == "__main__":
    threads = [Thread(target=task, args=(i,)) for i in range(4)]

    for thread in threads:
        thread.start()
```

```
    for thread in threads:
        thread.join()
```

Y un patrón equivalente con procesos podría expresarse de este modo:

```
from multiprocessing import Process


def task(value):
    print(value * value, flush=True)


if __name__ == "__main__":
    processes = [Process(target=task, args=(i,)) for i in range(4)]

    for process in processes:
        process.start()

    for process in processes:
        process.join()
```

Estas versiones son claras, pero no escalan bien cuando hay muchas tareas pequeñas. Crear un worker por iteración introduce un overhead considerable y, además, dificulta la recuperación ordenada de resultados.

6.6.2. Pools y executors

Cuando la tarea consiste en aplicar la misma función muchas veces con diferentes entradas, conviene reutilizar un conjunto fijo de workers. Ahí aparecen los pools y los executors.

Un ejemplo con `ProcessPoolExecutor` permite ver el patrón moderno más útil para tareas CPU-bound:

```python
from concurrent.futures import ProcessPoolExecutor


def task(value):
    return value * value


if __name__ == "__main__":
    with ProcessPoolExecutor() as executor:
        results = list(executor.map(task, range(8)))

    print(results)
```

Para tareas I/O-bound, una estructura análoga con `ThreadPoolExecutor` suele ser más natural. La diferencia importante no está solo en la sintaxis, sino en el tipo de problema para el cual cada executor resulta apropiado.

En términos prácticos, los executors suelen ser preferibles a la creación manual masiva porque reutilizan workers, simplifican la recolección de resultados y expresan mejor la intención del programa.

6.7. Secuencial, procesos y Numba sobre problemas clásicos

Para comparar estrategias conviene partir de un problema pequeño y homogéneo. Un ejemplo típico es recorrer un vector y acumular resultados. En versión secuencial, el patrón es simple:

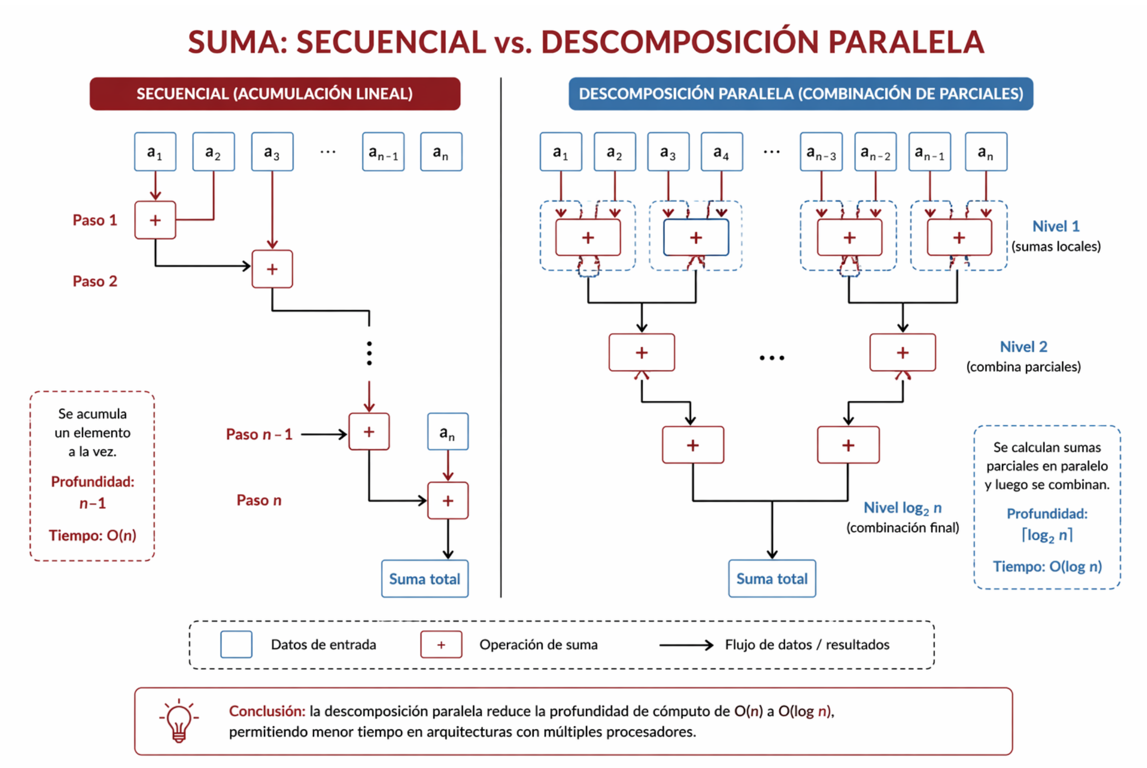

Figura 6.1: Transformación de un recorrido secuencial en una ejecución repartida entre trabajadores con reducción final.

```python
def sum_elements(values):
    total = 0
    for value in values:
        total += value
    return total
```

Si se intenta resolver este mismo problema con `threading`, puede que el rendimiento no mejore en tareas CPU-bound debido al GIL. Con `multiprocessing`, en cambio, es posible repartir bloques entre procesos distintos y combinar subtotales. Una versión conceptual mínima podría verse así:

```python
from multiprocessing import Pool


def partial_sum(values):
    total = 0
```

```
    for value in values:
        total += value
    return total


if __name__ == "__main__":
    chunks = [list(range(0, 250000)), list(range(250000, 500000))]

    with Pool(processes=2) as pool:
        partials = pool.map(partial_sum, chunks)

    print(sum(partials))
```

La idea importante es que la ganancia no proviene solo de usar una biblioteca diferente, sino de haber repartido los datos y de aceptar el costo de serialización y sincronización entre procesos.

Una lógica análoga puede usarse para multiplicar matrices si el trabajo se divide por bloques de filas. En este caso conviene partir primero de una versión secuencial breve, porque eso vuelve más clara la transición hacia las variantes paralelas.

```
def matmul_sequential(matrix_a, matrix_b):
    result = []
    for row in matrix_a:
        result_row = []
        for j in range(len(matrix_b[0])):
            total = 0
            for k in range(len(matrix_b)):
                total += row[k] * matrix_b[k][j]
            result_row.append(total)
        result.append(result_row)
    return result
```

Si luego se reparte el trabajo por bloques de filas, cada proceso recibe un subconjunto de filas de la matriz `A`, calcula su bloque de salida usando la matriz `B` y luego los bloques parciales se reensamblan en la matriz resultado.

```
from multiprocessing import Pool


def multiply_block(block, matrix_b):
    result = []
    for row in block:
        result_row = []
        for j in range(len(matrix_b[0])):
            total = 0
            for k in range(len(matrix_b)):
                total += row[k] * matrix_b[k][j]
            result_row.append(total)
        result.append(result_row)
    return result


if __name__ == "__main__":
    matrix_a = [
        [0, 1, 2],
        [3, 4, 5],
        [6, 7, 8],
        [9, 10, 11],
    ]
    matrix_b = [
        [0, 1],
        [2, 3],
        [4, 5],
    ]
    row_blocks = [matrix_a[:2], matrix_a[2:]]

    with Pool(processes=2) as pool:
        partial_blocks = pool.starmap(
            multiply_block,
```

```
            [(block, matrix_b) for block in row_blocks],
        )

    matrix_c = []
    for block in partial_blocks:
        matrix_c.extend(block)
```

Aquí reaparece una idea central del capítulo: paralelizar no consiste solo en repartir operaciones, sino también en decidir cómo particionar los datos y cómo recomponer luego el resultado total.

Con Numba, el mismo problema puede atacarse desde otra lógica: en lugar de coordinar múltiples procesos desde Python, se compila el cálculo y se deja que la biblioteca optimice la ejecución.

En los ejemplos iniciales de este capítulo se usan listas de Python para mantener visible la estructura del recorrido y evitar que una biblioteca externa oculte parte del costo. Con Numba, sin embargo, esta decisión tiene límites: para obtener buenos resultados suelen necesitarse datos tipados y estructuras compatibles con compilación eficiente, como arreglos numéricos. Por ese motivo, Numba debe leerse como una transición hacia una ejecución más cercana al código compilado, no simplemente como el mismo loop de Python ejecutado de otra manera.

```
from numba import njit, prange


@njit(parallel=True)
def sum_numba(values):
    total = 0.0
    for index in prange(len(values)):
        total += values[index]
    return total
```

Este enfoque es especialmente importante en el libro porque conecta con OpenMP a nivel conceptual y permite contrastar una estrategia basada en compilación con otras basadas en workers explícitos.

La misma lógica puede extenderse a una multiplicación de matrices escrita de manera explícita:

```
from numba import njit, prange


@njit(parallel=True)
def matmul_numba(matrix_a, matrix_b):
    rows = len(matrix_a)
    inner = len(matrix_b)
    cols = len(matrix_b[0])
    result = [[0.0 for _ in range(cols)] for _ in range(rows)]

    for i in prange(rows):
        for j in range(cols):
            total = 0.0
            for k in range(inner):
                total += matrix_a[i][k] * matrix_b[k][j]
            result[i][j] = total

    return result
```

En este caso, el paralelismo aparece al repartir el cálculo de filas de la matriz resultado. La estructura del algoritmo sigue siendo clara, pero el costo del problema ya no está solo en acumular valores, sino en coordinar muchas operaciones sobre dos estructuras bidimensionales.

Al medir funciones compiladas con Numba conviene considerar una corrida inicial de calentamiento. La primera ejecución puede incluir el costo de compilación just in time, por lo que no siempre representa el tiempo real de las ejecuciones posteriores. Para comparar con más cuidado, suele ejecutarse una vez la función, descartar esa medición y luego repetir varias corridas usando un temporizador como `time.perf_counter()`.

6.8. Un caso práctico transversal: Sobel en CPU

Además de los ejemplos más clásicos, conviene introducir un problema que acompañará el resto del recorrido del libro: la detección de bordes mediante el operador de Sobel.

En términos generales, Sobel aplica máscaras de convolución sobre una imagen en escala de grises para estimar cambios de intensidad en direcciones horizontal y vertical. A partir de esas respuestas parciales puede construirse una imagen de bordes. Desde el punto de vista del paralelismo, se trata de un caso útil porque trabaja sobre datos bidimensionales, exige recorrer vecinos y permite comparar con claridad distintas maneras de expresar un mismo cálculo.

Figura 6.2: Flujo general de aplicación del filtro de Sobel sobre una imagen en CPU.

Una formulación secuencial mínima puede escribirse así:

```
def sobel_sequential(image):
    kernel_x = [
        [-1, 0, 1],
        [-2, 0, 2],
        [-1, 0, 1],
    ]
    kernel_y = [
        [-1, -2, -1],
        [0, 0, 0],
        [1, 2, 1],
    ]
```

```
    height = len(image)
    width = len(image[0])
    result = [[0.0 for _ in range(width)] for _ in range(height)]

    for row in range(1, height - 1):
        for col in range(1, width - 1):
            gx = 0.0
            gy = 0.0
            for kr in range(3):
                for kc in range(3):
                    pixel = image[row + kr - 1][col + kc - 1]
                    gx += pixel * kernel_x[kr][kc]
                    gy += pixel * kernel_y[kr][kc]
            result[row][col] = abs(gx) + abs(gy)

    return result
```

Esta versión deja ver con claridad la lógica básica del problema: para cada posición interior de la imagen, se toman vecinos cercanos, se aplican dos máscaras y luego se combinan los resultados parciales. También permite advertir por qué se trata de un problema intensivo en CPU cuando la imagen crece, ya que el cálculo debe repetirse para una gran cantidad de píxeles.

Una primera aceleración razonable consiste en conservar la misma lógica general, pero compilarla con Numba sobre CPU:

```
from numba import njit, prange


@njit(parallel=True)
def sobel_numba(image):
    kernel_x = (
        (-1, 0, 1),
        (-2, 0, 2),
        (-1, 0, 1),
```

```
    )
    kernel_y = (
        (-1, -2, -1),
        (0, 0, 0),
        (1, 2, 1),
    )

    height = len(image)
    width = len(image[0])
    result = [[0.0 for _ in range(width)] for _ in range(height)]

    for row in prange(1, height - 1):
        for col in range(1, width - 1):
            gx = 0.0
            gy = 0.0
            for kr in range(3):
                for kc in range(3):
                    pixel = image[row + kr - 1][col + kc - 1]
                    gx += pixel * kernel_x[kr][kc]
                    gy += pixel * kernel_y[kr][kc]
            result[row][col] = abs(gx) + abs(gy)

    return result
```

En este punto no hace falta desarrollar una comparación experimental detallada. Sin embargo, sí conviene señalar una expectativa general: cuando el tamaño de la imagen es suficientemente grande, se espera que una versión compilada con Numba CPU reduzca de manera sustancial el tiempo de ejecución respecto de la versión secuencial escrita solo con loops de Python.

6.9. Cómo elegir una estrategia

Antes de escribir código conviene responder una pregunta simple: ¿el problema está limitado por entrada y salida o por cálculo?

- si el problema es I/O-bound, los hilos suelen ser una opción razonable;
- si el problema es CPU-bound, los procesos o herramientas como Numba suelen ser mejores alternativas;
- si el trabajo consiste en aplicar operaciones repetitivas sobre arreglos numéricos, conviene evaluar primero vectorización, que se verá en próximos capítulos;
- si se necesita acelerar código Python sin reescribir todo el algoritmo en otro lenguaje, Numba funciona como un puente especialmente útil.

6.10. Errores frecuentes al paralelizar en Python

Hasta aquí el capítulo se concentró en elegir bibliotecas y patrones. Sin embargo, al paralelizar también aparecen errores específicos que conviene reconocer desde una etapa introductoria.

Uno de los más conocidos es la race condition. Aparece cuando varios hilos o procesos acceden a un mismo dato compartido sin coordinación suficiente y el resultado depende del orden efectivo de ejecución. En esos casos, el programa puede producir salidas distintas entre una ejecución y otra.

En memoria compartida, una actualización aparentemente simple como incrementar un contador puede volverse problemática si no se protege adecuadamente. Conceptualmente, el problema no está en la suma en sí, sino en que leer, modificar y escribir no siempre constituye una operación atómica.

Otro riesgo importante es el deadlock. Ocurre cuando dos o más tareas quedan esperando recursos entre sí y ninguna puede avanzar. En Python, este problema puede aparecer al combinar locks, colas, joins o recursos compartidos en un orden incorrecto.

En un nivel introductorio, conviene quedarse con tres señales de alerta:

- resultados no deterministas cuando se reutilizan datos compartidos;
- programas que terminan más lentamente de lo esperado sin mejorar con más workers;
- ejecuciones que quedan bloqueadas o no finalizan.

Reconocer estos síntomas es importante porque no todo problema de paralelismo es un problema de rendimiento. A veces el código falla primero en corrección y solo después en velocidad.

6.11. Una mirada inicial al debugging y profiling

En este contexto, debugging y profiling nombran dos prácticas complementarias. El debugging busca detectar y explicar errores de corrección, por ejemplo resultados inconsistentes, bloqueos o comportamientos no esperados. El profiling, en cambio, se orienta a observar cómo se consume el tiempo de ejecución y qué partes del programa concentran el costo principal. En problemas paralelos, ambas miradas suelen necesitarse al mismo tiempo, porque una implementación puede fallar por coordinación incorrecta o funcionar bien desde el punto de vista lógico pero rendir mucho peor de lo esperado.

Debuggear programas paralelos exige observar más que el resultado final. En Python conviene empezar por preguntas sencillas:

- ¿los workers están haciendo trabajo real o permanecen ociosos?;
- ¿el tiempo de ejecución se concentra en el cómputo o en crear procesos, mover datos o sincronizar?;
- ¿el uso de CPU se distribuye entre varios cores o sigue concentrado en uno solo?;
- ¿las salidas cambian entre ejecuciones equivalentes?

Una primera aproximación razonable consiste en medir tiempos con cuidado, observar la carga de CPU del sistema y contrastar resultados entre distintas configuraciones. Ese tipo de observación ya alcanza para detectar muchos errores de diseño, por ejemplo tareas demasiado pequeñas para justificar procesos o uso de threads en problemas claramente CPU-bound.

Por ese motivo, profiling y debugging deben leerse aquí como prácticas de observación básica: medir, comparar, repetir y buscar explicaciones consistentes para el comportamiento del programa.

El anexo de trabajos prácticos retoma estas ideas mediante ejercicios de medición progresiva. Allí se propone comparar versiones secuenciales, con threads, con procesos y con Numba, registrando entorno de ejecución, tiempos, speed-up y eficiencia.

6.12. Una tabla de síntesis del capítulo

Estrategia	Conviene usarla cuando	Ventaja principal	Límite principal
Versión secuencial	el problema es pequeño o se necesita una línea de base clara	simplicidad y facilidad de inspección	no aprovecha varios cores
`threading`	el trabajo es I/O-bound o la coordinación entre tareas pesa más que el cálculo	bajo costo de coordinación y modelo directo de concurrencia	el GIL limita su utilidad en tareas CPU-bound
`multiprocessing`	el problema es CPU-bound y puede dividirse en bloques independientes	paralelismo real entre procesos y evita el GIL	serialización, copia de datos y mayor overhead
Numba CPU	se quiere acelerar cálculo numérico sin salir de Python	compilación JIT y mejor aprovechamiento de CPU	exige código compatible y no reemplaza toda estrategia de partición

6.13. Ejercicios del capítulo

- Compare tareas I/O-bound y CPU-bound en relación con `threading` y `multiprocessing`.
- Distinga una race condition de un problema de rendimiento.
- Diseñe una tabla de resultados para comparar una versión secuencial, una versión con threads, una versión con procesos y una versión con Numba de un mismo problema.
- Proponga un problema I/O-bound y otro CPU-bound, e indique qué estrategia usaría en cada caso.
- Describa una situación en la que un programa paralelo en Python podría quedar bloqueado o producir resultados no deterministas.
- Justifique cuál de las siguientes estrategias elegiría para un problema de suma de grandes arreglos numéricos: `threading`, `multiprocessing` o Numba.

- Redacte una reflexión breve sobre por qué una implementación más paralela en apariencia puede no ser la más rápida en Python.
- Explique qué observaría primero para diagnosticar por qué una versión paralela no mejora respecto de la secuencial.

7 Vectorización, broadcasting y optimización sobre arreglos

Después de estudiar hilos, procesos y compilación JIT sobre CPU, conviene introducir otra idea central para el rendimiento: muchas mejoras no provienen de crear workers explícitos, sino de reformular el cálculo. En problemas numéricos regulares, el cambio decisivo suele consistir en dejar de pensar en iteraciones individuales y pasar a operar sobre arreglos o tensores completos.

Este cambio de formulación reduce la sobrecarga del intérprete, aprovecha bibliotecas implementadas en bajo nivel y suele mejorar la relación entre cómputo y acceso a memoria. Por ese motivo, la vectorización y el broadcasting no deben leerse como meras definiciones aisladas, sino como parte de una estrategia de optimización sobre CPU.

7.1. Objetivos del capítulo

- definir vectorización y broadcasting en un nivel introductorio;
- explicar por qué reformular un cálculo puede ser más efectivo que paralelizarlo de manera explícita;
- relacionar estas técnicas con SIMD, con NumPy y con la organización de datos en memoria;
- introducir tensores en PyTorch CPU como continuidad conceptual del trabajo sobre arreglos;
- retomar el caso de Sobel para mostrar cómo un mismo problema puede reescribirse sobre arreglos y tensores completos.

7.2. Planteo del capítulo

En el capítulo anterior se estudiaron estrategias donde el programador decide de forma visible cómo repartir trabajo entre threads, procesos o regiones paralelas compiladas con Numba. Aquí el foco cambia. La pregunta central ya no es cómo crear workers, sino cómo expresar el mismo problema de una forma que el entorno pueda ejecutar de manera más eficiente.

En términos generales, esta diferencia separa dos estrategias complementarias. El paralelismo explícito distribuye tareas. La reformulación sobre arreglos cambia el nivel de abstracción del cálculo. En muchos problemas regulares, esta segunda vía produce mejores resultados sobre CPU precisamente porque evita parte del overhead de coordinación y aprovecha implementaciones optimizadas.

7.3. Qué es la vectorización

La vectorización consiste en aplicar operaciones sobre estructuras completas de datos, como vectores o matrices, en lugar de recorrer elemento por elemento mediante bucles explícitos en Python. Esta estrategia se apoya en implementaciones optimizadas de bajo nivel y en capacidades del hardware asociadas con SIMD, es decir, la ejecución de una misma instrucción sobre múltiples datos.

En términos prácticos, vectorizar significa reemplazar una lógica del tipo “para cada elemento, hacer una operación” por una expresión que opera sobre todo el arreglo de una vez. La operación conceptual puede ser la misma, pero el costo de ejecución cambia porque ya no se depende del intérprete de Python para administrar cada iteración individual.

Una comparación mínima permite verlo con claridad. Si se quiere sumar dos vectores `a` y `b`, una formulación explícita con un bucle en Python podría ser:

```
result = []
for i in range(len(a)):
    result.append(a[i] + b[i])
```

Frente a una formulación vectorizada:

```
import numpy as np

result = np.array(a) + np.array(b)
```

La segunda versión no es solo más breve. También es más eficiente porque delega el cálculo a una implementación optimizada.

En este punto aparece una diferencia importante respecto de los ejemplos iniciales con listas de Python. Un arreglo de NumPy no es solo una lista con otra sintaxis: tiene una forma (`shape`), un tipo de dato (`dtype`) y una organización concreta en memoria. La forma indica sus dimensiones; el tipo define cuánta memoria ocupa cada elemento y qué operaciones son válidas; y la disposición interna condiciona si los datos pueden recorrerse de manera contigua y eficiente.

Algo análogo puede verse con multiplicación de matrices. Una formulación explícita exige recorrer filas, columnas y productos parciales:

```
result = []
for i in range(len(a)):
    row = []
    for j in range(len(b[0])):
        total = 0
        for k in range(len(b)):
            total += a[i][k] * b[k][j]
        row.append(total)
    result.append(row)
```

Frente a una formulación vectorizada:

```
import numpy as np

result = np.array(a) @ np.array(b)
```

Aquí también aparece el cambio de nivel de abstracción: en lugar de describir cada multiplicación y cada suma parcial, se expresa directamente la operación matricial completa y se deja la ejecución en manos de una biblioteca optimizada.

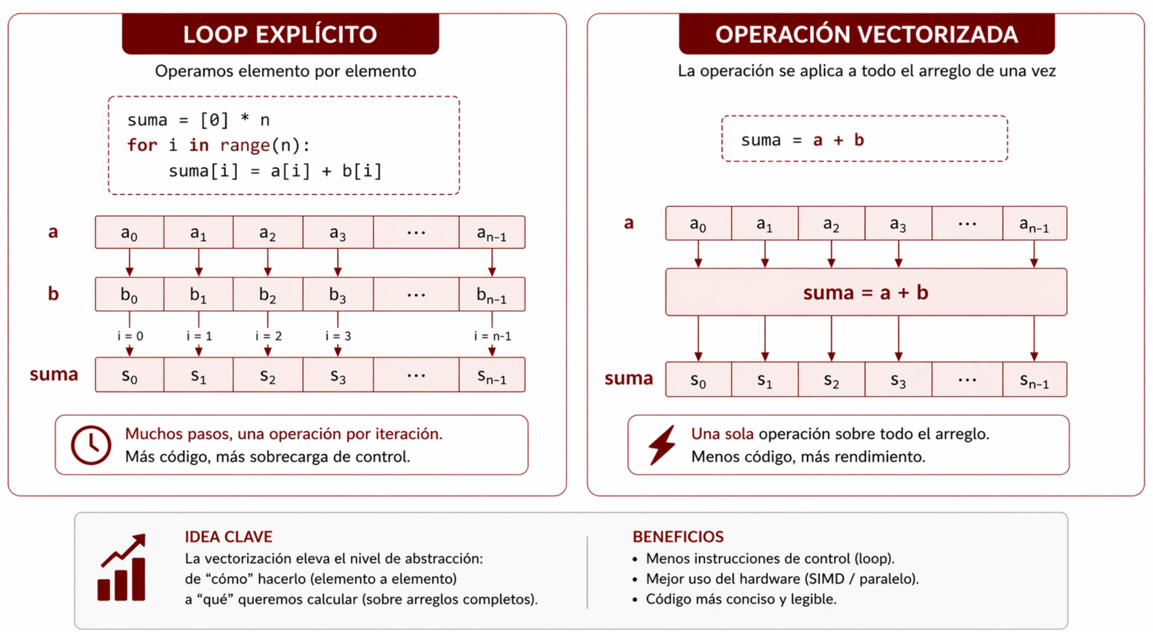

Figura 7.1: Comparación entre un loop explícito y una formulación vectorizada de la misma operación.

7.4. SIMD como fundamento

SIMD, sigla de Single Instruction, Multiple Data, permite que una misma operación actúe simultáneamente sobre varios elementos. Esta idea resulta especialmente adecuada para tareas repetitivas sobre grandes conjuntos de datos, como procesamiento de imágenes, audio, video, simulaciones físicas y entrenamiento de modelos de inteligencia artificial.

Conviene notar que SIMD no es lo mismo que vectorización, aunque ambas ideas estén muy relacionadas. SIMD es una capacidad de hardware o de bajo nivel. La vectorización es una forma de expresar el cálculo de modo tal que ese hardware pueda aprovecharse. Dicho de otro modo, la vectorización suele ser la puerta de entrada de alto nivel a comportamientos cercanos a SIMD.

7.5. Qué es el broadcasting

El broadcasting es un conjunto de reglas que permite combinar arreglos de distinto tamaño cuando sus dimensiones son compatibles según ciertos criterios. En lugar de expandir manualmente los

datos o escribir bucles adicionales, la biblioteca aplica la operación como si ciertos valores se difundieran sobre la estructura mayor.

Un caso elemental es sumar un escalar a todos los elementos de un vector:

```
import numpy as np

values = np.array([1, 2, 3])
result = values + 5
```

Aquí el valor `5` actúa como si se replicara sobre todas las posiciones del arreglo, aunque esa expansión no se escriba manualmente. El interés del broadcasting está justamente en evitar código adicional y permitir expresiones compactas sobre estructuras compatibles.

También puede aparecer en operaciones entre matrices y vectores, por ejemplo cuando se desea sumar un vector fila a todas las filas de una matriz. Este patrón es muy frecuente en procesamiento de imágenes, normalización de datos y cálculo científico.

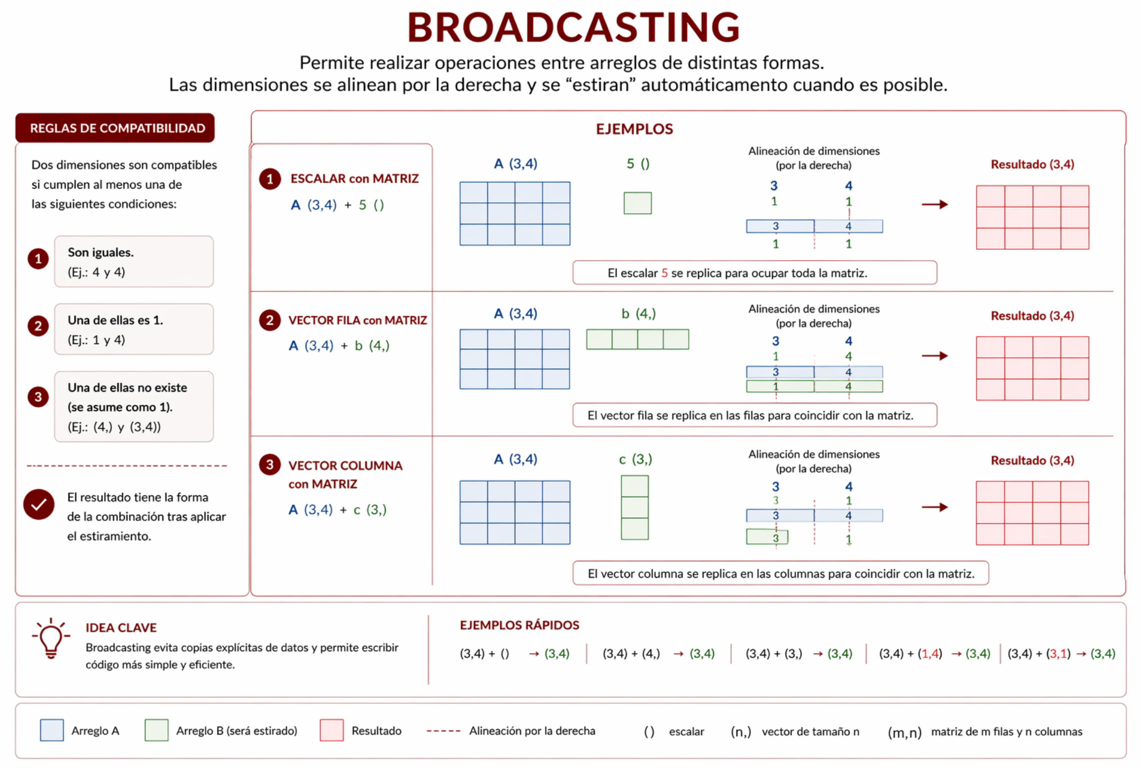

Figura 7.2: Esquema de broadcasting para mostrar cómo se compatibilizan dimensiones en operaciones sobre arreglos.

7.6. Relación entre vectorización y broadcasting

Vectorización y broadcasting suelen aparecer juntas. La primera se refiere a operar sobre arreglos completos; la segunda, a compatibilizar dimensiones para que esa operación sea posible sin trabajo manual adicional. Aunque se las confunda con frecuencia, conviene distinguirlas porque responden a problemas diferentes.

La vectorización responde a la pregunta "cómo evitar iterar elemento por elemento". El broadcasting responde a la pregunta "cómo combinar estructuras de distinta forma sin escribir lógica adicional para expandirlas".

7.7. El lugar de NumPy

Antes de paralelizar bucles en Python, conviene considerar una alternativa muchas veces más efectiva: eliminar el bucle explícito. NumPy permite expresar operaciones sobre arreglos completos mediante vectorización. En problemas numéricos regulares, esta estrategia suele superar a muchas soluciones basadas en threads o procesos, justamente porque reduce la sobrecarga del intérprete y aprovecha implementaciones de bajo nivel optimizadas.

Por ejemplo, si el objetivo es sumar dos vectores, una formulación vectorizada como la siguiente suele ser preferible a un loop Python paralelizado manualmente:

```
import numpy as np

a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
c = a + b
```

Esta observación es central para el recorrido del libro: no todo problema repetitivo debe resolverse con hilos o procesos. En muchos casos, la optimización correcta consiste en cambiar el nivel de abstracción del cálculo.

7.8. Reformular también es optimizar

La mejora de rendimiento asociada con vectorización no depende solo de hacer varias operaciones a la vez. También influye el modo en que los datos se organizan y se recorren en memoria. Cuando un arreglo está dispuesto de forma contigua y la operación recorre los datos con regularidad, el hardware puede aprovechar mejor la jerarquía de caché y reducir accesos costosos a memoria principal.

Aquí reaparece una idea vista al estudiar arquitectura y métricas: muchas tareas con arreglos grandes están limitadas por memoria más que por cálculo puro. En esos casos, escribir el problema de forma vectorizada puede mejorar tanto la expresión del cálculo como el patrón de acceso a memoria.

Esto explica por qué una implementación vectorizada puede superar a otra más “paralela” en apariencia. Si la segunda introduce overhead de coordinación entre hilos o procesos, o recorre peor los datos, la ventaja de repartir trabajo entre varios workers puede evaporarse rápidamente.

También conviene reconocer un límite: vectorizar no significa automáticamente usar menos memoria. Algunas expresiones crean arreglos temporales intermedios, especialmente cuando se encadenan varias operaciones sobre matrices o tensores grandes. En problemas pequeños esto suele ser irrelevante, pero en arreglos de gran tamaño puede aumentar el consumo de memoria y afectar el rendimiento. Por ese motivo, además de mirar si una expresión es breve, conviene preguntarse cuántos datos temporales crea y si el tamaño del problema permite sostenerlos en memoria.

7.9. Un ejemplo de localidad: matrices y transposición

La multiplicación de matrices ofrece un caso útil para observar esta idea. Si una matriz se recorre por filas pero la otra se consulta columna por columna, el patrón de acceso puede volverse menos regular. Una manera de mejorar la localidad consiste en transponer previamente una de las matrices para que ambos recorridos se hagan sobre datos dispuestos de forma más conveniente.

Una formulación explícita básica puede escribirse así:

```python
def matmul_sequential(matrix_a, matrix_b):
    rows = len(matrix_a)
    inner = len(matrix_b)
    cols = len(matrix_b[0])
    result = [[0.0 for _ in range(cols)] for _ in range(rows)]

    for i in range(rows):
        for j in range(cols):
            total = 0.0
            for k in range(inner):
                total += matrix_a[i][k] * matrix_b[k][j]
            result[i][j] = total

    return result
```

Si primero se transpone `matrix_b`, el cálculo interno puede reescribirse para recorrer dos filas y no una fila junto con una columna:

```python
def transpose(matrix):
    return [list(row) for row in zip(*matrix)]



def matmul_with_transposed_b(matrix_a, matrix_b):
    matrix_b_t = transpose(matrix_b)
    rows = len(matrix_a)
    cols = len(matrix_b_t)
    result = [[0.0 for _ in range(cols)] for _ in range(rows)]

    for i in range(rows):
        row_a = matrix_a[i]
        for j in range(cols):
            row_b_t = matrix_b_t[j]
            total = 0.0
            for k in range(len(row_a)):
```

```
                total += row_a[k] * row_b_t[k]
            result[i][j] = total

    return result
```

El algoritmo sigue siendo secuencial y mantiene la misma complejidad asintótica. Lo que cambia es la forma de recorrer los datos. Este ejemplo es importante porque muestra que optimizar no siempre significa agregar paralelismo explícito. A veces significa reorganizar los datos para que el acceso a memoria sea más regular.

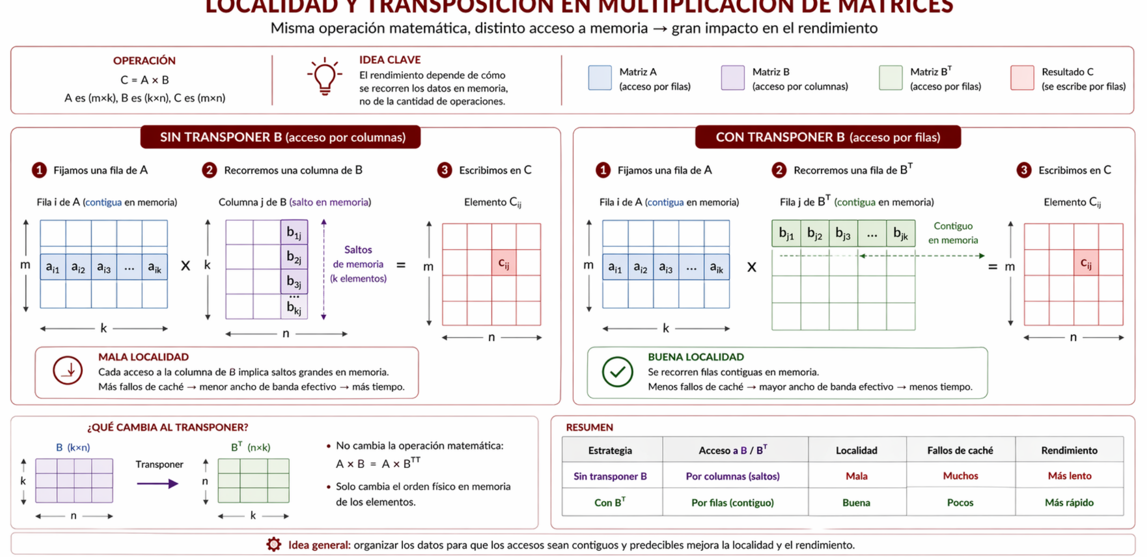

Figura 7.3: Efecto de la localidad de memoria y de la transposición sobre el acceso a datos en multiplicación de matrices.

Con NumPy, esa misma reformulación puede expresarse de modo más directo:

```
import numpy as np

matrix_a = np.array(matrix_a, dtype=np.float64)
matrix_b = np.array(matrix_b, dtype=np.float64)
matrix_b_t = matrix_b.T

matrix_c = matrix_a @ matrix_b
```

```
matrix_c_alt = np.empty((matrix_a.shape[0], matrix_b_t.shape[0]), dtype=np.float64)

for i in range(matrix_a.shape[0]):
    for j in range(matrix_b_t.shape[0]):
        matrix_c_alt[i, j] = np.dot(matrix_a[i], matrix_b_t[j])
```

En este caso, el cálculo sigue apoyándose en una biblioteca optimizada, pero la reformulación vuelve a hacerse visible: al trabajar con la matriz ya transpuesta, cada producto interno se expresa entre filas. Lo importante aquí no es la sintaxis puntual, sino la idea de que la organización de datos y la forma de expresar el cálculo condicionan el rendimiento.

7.10. Broadcasting como técnica puntual de reformulación

Un ejemplo cercano a usos reales aparece al normalizar columnas de una matriz. Si se quiere restar a cada columna su media, una primera formulación secuencial con bucles explícitos podría escribirse así:

```
matrix = [
    [1.0, 10.0, 100.0],
    [2.0, 20.0, 200.0],
    [3.0, 30.0, 300.0],
]

column_means = [
    sum(row[column] for row in matrix) / len(matrix)
    for column in range(len(matrix[0]))
]

centered = []
for row in matrix:
    centered_row = []
    for column in range(len(row)):
```

```
        centered_row.append(row[column] - column_means[column])
    centered.append(centered_row)
```

En cambio, una formulación vectorizada con broadcasting puede escribirse así:

```
import numpy as np

matrix = np.array([
    [1.0, 10.0, 100.0],
    [2.0, 20.0, 200.0],
    [3.0, 30.0, 300.0],
])

# axis=0 indica que la media se calcula columna por columna
column_means = matrix.mean(axis=0)
centered = matrix - column_means
```

Aquí no hace falta expandir manualmente `column_means` para cada fila. El broadcasting aplica la resta como si ese vector se replicara sobre toda la matriz. Comparado con la versión secuencial, cambia el nivel de abstracción y desaparece la necesidad de administrar el recorrido en Python.

7.11. Tensores en CPU: continuidad con PyTorch

Hasta aquí el eje estuvo puesto en NumPy, que sigue siendo la herramienta principal de este capítulo. Sin embargo, conviene introducir una continuidad conceptual importante: muchos de estos mismos problemas también pueden expresarse sobre tensores en PyTorch, aun cuando la ejecución siga ocurriendo en CPU.

Desde un punto de vista introductorio, un tensor puede pensarse como una generalización de arreglos multidimensionales. En CPU, PyTorch permite trabajar con tensores y operaciones sobre bloques completos de datos de una forma cercana a NumPy. La diferencia relevante para el recorrido del libro es que esta formulación servirá luego como puente natural hacia GPU.

Una suma simple de tensores en CPU puede verse así:

```python
import torch

a = torch.tensor([1.0, 2.0, 3.0], device="cpu")
b = torch.tensor([4.0, 5.0, 6.0], device="cpu")
c = a + b
```

También puede reaparecer el broadcasting:

```python
import torch

matrix = torch.tensor([
    [1.0, 10.0, 100.0],
    [2.0, 20.0, 200.0],
    [3.0, 30.0, 300.0],
], device="cpu")

column_means = matrix.mean(dim=0)
centered = matrix - column_means
```

La lógica conceptual es la misma que en NumPy: operar sobre estructuras completas y dejar el detalle de muchas optimizaciones en manos de la biblioteca.

7.12. Un caso práctico transversal: Sobel con arreglos y tensores

En el capítulo anterior se presentó Sobel en una versión secuencial y luego en una variante acelerada con Numba CPU. Ahora conviene retomar el mismo problema desde otra pregunta: cómo reescribir ese cálculo para operar sobre arreglos o tensores completos, en lugar de administrar manualmente cada píxel desde Python.

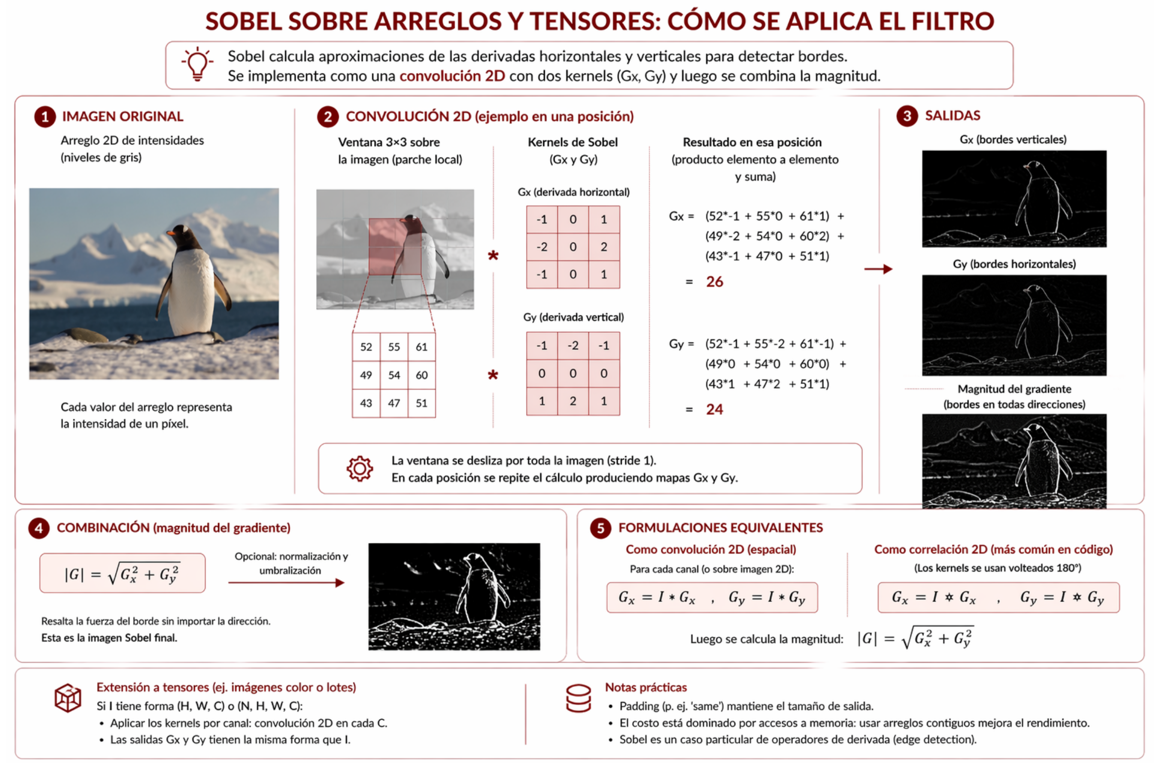

Figura 7.4: Representación del filtro de Sobel como operación de convolución sobre arreglos y tensores.

Si se parte de una imagen en escala de grises almacenada como arreglo bidimensional, una formulación con NumPy puede construirse a partir de cortes que representan las nueve posiciones de cada vecindad `3 x 3`:

```
import numpy as np


def sobel_numpy(image):
    image = np.asarray(image, dtype=np.float32)
    result = np.zeros_like(image)

    top_left = image[:-2, :-2]
    top = image[:-2, 1:-1]
    top_right = image[:-2, 2:]
    left = image[1:-1, :-2]
```

```
    right = image[1:-1, 2:]
    bottom_left = image[2:, :-2]
    bottom = image[2:, 1:-1]
    bottom_right = image[2:, 2:]

    gx = (
        -top_left + top_right
        - 2.0 * left + 2.0 * right
        - bottom_left + bottom_right
    )
    gy = (
        -top_left - 2.0 * top - top_right
        + bottom_left + 2.0 * bottom + bottom_right
    )

    result[1:-1, 1:-1] = np.abs(gx) + np.abs(gy)
    return result
```

La idea importante aquí no es memorizar la fórmula, sino observar la reformulación. El problema deja de expresarse como una doble iteración sobre píxeles y pasa a escribirse como una combinación de subarreglos que representan vecinos alineados.

Esa misma lógica puede trasladarse a tensores en PyTorch CPU sin recurrir todavía a primitivas de convolución ya resueltas. Una versión con tensores y operaciones explícitas puede escribirse así:

```
import torch


def sobel_torch_cpu(image):
    image = torch.as_tensor(image, dtype=torch.float32, device="cpu")
    result = torch.zeros_like(image)

    top_left = image[:-2, :-2]
    top = image[:-2, 1:-1]
```

```
    top_right = image[:-2, 2:]
    left = image[1:-1, :-2]
    right = image[1:-1, 2:]
    bottom_left = image[2:, :-2]
    bottom = image[2:, 1:-1]
    bottom_right = image[2:, 2:]

    gx = (
        -top_left + top_right
        - 2.0 * left + 2.0 * right
        - bottom_left + bottom_right
    )
    gy = (
        -top_left - 2.0 * top - top_right
        + bottom_left + 2.0 * bottom + bottom_right
    )

    result[1:-1, 1:-1] = torch.abs(gx) + torch.abs(gy)
    return result
```

En esta variante, el cálculo sigue ocurriendo sobre CPU, pero ya se lo formula sobre tensores. Esa continuidad es importante porque prepara el cambio de dispositivo que se estudiará en el próximo capítulo. Lo nuevo allí no será la idea de tensor en sí misma, sino su ejecución sobre GPU y las consecuencias que eso trae en términos de transferencias, kernels y jerarquía de memoria del acelerador.

7.13. Criterios para el análisis práctico

Para estudiar estas técnicas con mayor profundidad conviene comparar implementaciones con bucles explícitos frente a versiones vectorizadas, observar cómo cambian las dimensiones de los arreglos en operaciones con broadcasting y relacionar esas transformaciones con los tiempos de ejecución obtenidos.

En particular, conviene revisar al menos estas preguntas:

- ¿la operación puede escribirse sobre arreglos completos o depende de lógica muy irregular?;
- ¿los datos tienen una forma compatible con broadcasting o requieren reestructuración previa?;
- ¿la ganancia observada proviene de menos iteraciones en Python, de mejor acceso a memoria o de ambas cosas?;
- ¿la organización de datos favorece accesos regulares a memoria o introduce recorridos poco locales?;
- ¿el problema conviene expresarlo con arreglos de NumPy o con tensores que luego puedan continuar en GPU?

Estas preguntas permiten leer los resultados con más cuidado y evitar la idea simplista de que cualquier paralelización explícita será superior a una formulación vectorizada.

7.14. Una tabla de síntesis del capítulo

Estrategia	Conviene usarla cuando	Ventaja principal	Límite principal
Loop explícito en Python	el problema es pequeño, didáctico o muy irregular	control detallado del algoritmo	alto costo por iteración
Vectorización con NumPy	la operación es regular sobre arreglos	reduce overhead y aprovecha rutinas optimizadas	menos natural para lógica irregular
Broadcasting	hay que combinar estructuras compatibles de distinta forma	evita expansión manual y simplifica expresiones	requiere comprender bien las dimensiones

Estrategia	Conviene usarla cuando	Ventaja principal	Límite principal
Tensores en PyTorch CPU	conviene trabajar con operaciones sobre tensores y preparar continuidad hacia GPU	misma idea de reformulación sobre CPU con puente natural a aceleradores	agrega otra biblioteca y no reemplaza por sí sola la necesidad de analizar memoria y acceso a datos

Con este marco, el siguiente paso será estudiar qué ocurre cuando estas ideas se trasladan a GPU. La continuidad conceptual ya está instalada: el capítulo siguiente no introducirá desde cero el trabajo con tensores, sino el cambio de dispositivo y de modelo de ejecución.

El anexo retoma esta continuidad en el trabajo práctico sobre Sobel: primero se compara una versión secuencial con variantes en CPU, luego se reformula el cálculo con arreglos y tensores, y finalmente se analiza qué cambia al moverlo a GPU.

7.15. Ejercicios del capítulo

- Defina vectorización con sus palabras.
- Explique qué problema resuelve el broadcasting.
- Distinga entre vectorización, broadcasting y paralelismo explícito.
- Justifique por qué SIMD resulta relevante para estas técnicas.
- Compare conceptualmente la multiplicación de matrices secuencial con la variante que transpone una de las matrices antes del cálculo.
- Describa por qué la normalización de columnas puede expresarse de manera natural con broadcasting.
- Explique cómo se reformula Sobel en la versión con NumPy a partir de cortes sobre el arreglo original.
- Analice un problema numérico sencillo e indique si conviene resolverlo con loop explícito, vectorización con NumPy, broadcasting o tensores en PyTorch CPU. Justifique la decisión.
- Compare el papel que cumple Sobel en el capítulo 05 con el que cumple en este capítulo. Indique qué cambia en la estrategia de optimización.

- Explique por qué PyTorch CPU aparece aquí como continuidad de NumPy y no todavía como un tema centrado en GPU.

8 Computación con GPU

En las últimas décadas, las unidades de procesamiento gráfico (GPU) dejaron de ser dispositivos dedicados exclusivamente al renderizado visual y pasaron a ocupar un lugar relevante en la computación paralela. Este cambio no responde solo a una evolución del hardware gráfico, sino también a una transformación más amplia en la naturaleza de los problemas computacionales contemporáneos. Cada vez con mayor frecuencia, resulta necesario aplicar la misma operación sobre grandes volúmenes de datos, ya sea en simulaciones numéricas, procesamiento de imágenes, análisis de señales o aprendizaje automático. En ese contexto, las GPU se volvieron una alternativa frecuente para este tipo de tareas.

Su importancia proviene de una arquitectura pensada para sostener una gran cantidad de operaciones similares de manera concurrente. A diferencia de una CPU, que suele privilegiar la ejecución rápida y flexible de un número más reducido de hilos complejos, una GPU organiza sus recursos para maximizar el throughput sobre tareas altamente regulares. Por ese motivo, no se trata simplemente de un procesador "más rápido", sino de un dispositivo con una lógica de ejecución distinta, cuya ventaja aparece cuando el problema contiene suficiente paralelismo de datos.

Comprender este cambio de perspectiva resulta importante dentro del recorrido del libro. Hasta aquí se estudiaron hilos, procesos, vectorización y estrategias de paralelización en CPU. El paso hacia GPU no reemplaza esas ideas, sino que las continúa en una escala distinta. Muchas de las preguntas que ya se formularon sobre partición del trabajo, acceso a memoria y medición de rendimiento siguen siendo relevantes, aunque ahora en una arquitectura con reglas propias de organización, memoria y ejecución.

8.1. Objetivos del capítulo

- introducir el papel de las GPU en la computación paralela contemporánea;

- presentar de manera general el modelo de ejecución de CUDA;
- explicar los principales factores que influyen en el rendimiento de un kernel;
- introducir tipos de memoria y conceptos como coalescing, occupancy y warp divergence;
- incorporar criterios básicos de debugging y profiling para kernels GPU;
- presentar PyTorch como vía de acceso de alto nivel al trabajo paralelo con tensores sobre CPU y GPU;
- reconocer el papel de las transferencias, la sincronización y las operaciones por lotes en el rendimiento observado.

8.2. El lugar de las GPU en este recorrido

Las GPU se volvieron relevantes porque permiten ejecutar una gran cantidad de operaciones similares sobre grandes volúmenes de datos. Esta característica las hace adecuadas para problemas donde el paralelismo de datos es dominante, por ejemplo procesamiento de imágenes, simulaciones numéricas y entrenamiento de redes neuronales.

Conviene notar que esta ventaja no proviene solo de tener muchos núcleos. También depende de una arquitectura orientada a ocultar latencias, sostener alto ancho de banda de memoria y ejecutar miles de hilos ligeros sobre un mismo dispositivo. Esa lógica es diferente de la de una CPU generalista, que suele dedicar más recursos a la ejecución rápida de unos pocos hilos complejos.

8.3. Qué caracteriza a una GPU

Una GPU dispone de una gran cantidad de núcleos especializados y de un ancho de banda de memoria muy elevado en comparación con la memoria RAM convencional. Estas características la hacen apta para tareas masivas y repetitivas sobre grandes volúmenes de datos.

En este capítulo conviene distinguir entre dos familias de GPU que responden a contextos de uso diferentes. Por un lado, una GPU de consumo como la NVIDIA GeForce RTX 4090 permite pensar el paralelismo masivo en una estación de trabajo personal. Se trata de una GPU con 16384 núcleos CUDA, 24 GB de memoria GDDR6X y un diseño orientado tanto a gráficos avanzados como a creación de contenido, desarrollo y experimentación con modelos de inteligencia artificial en escala personal.

Por otro lado, una GPU de centro de datos como la NVIDIA H100 representa otra escala de diseño y de uso. Basada en la arquitectura Hopper, incorpora Tensor Cores de cuarta generación y está pensada para entrenamiento e inferencia de modelos grandes, computación científica y análisis de datos en servidores. En sus configuraciones más difundidas ofrece 80 GB de memoria HBM3, un ancho de banda de memoria del orden de 3.35 TB/s y enlaces NVLink de hasta 900 GB/s entre GPU, rasgos que la vuelven adecuada para clústeres y sistemas multi-GPU.

Más allá del modelo puntual, lo importante es comprender el principio general: la GPU ofrece paralelismo masivo, aunque bajo reglas de programación distintas de las de una CPU tradicional.

En términos generales, una CPU optimiza latencia y control. Una GPU optimiza throughput (rendimiento), es decir, la cantidad total de trabajo completado sobre un gran conjunto de datos. Por ese motivo, una GPU no siempre es la mejor elección para cualquier algoritmo. Su ventaja aparece cuando el problema contiene muchas operaciones similares e independientes.

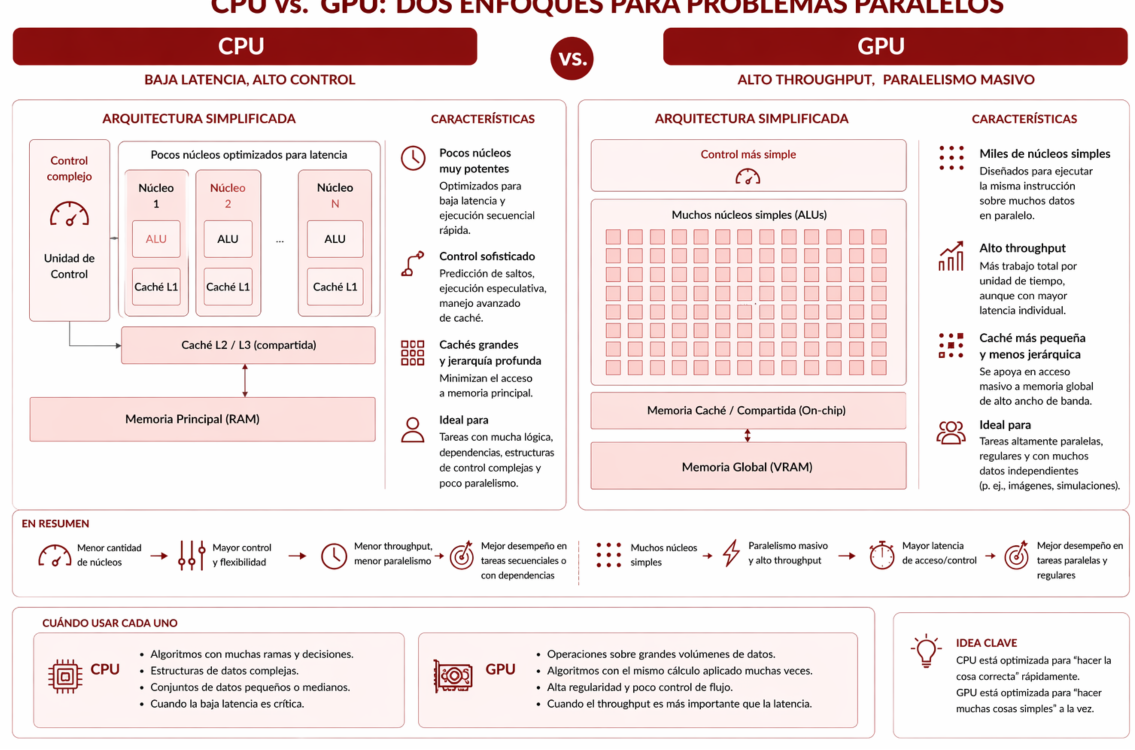

Figura 8.1: Comparación esquemática entre la organización interna de una CPU y una GPU.

Antes de pasar a los modelos de programación, conviene interpretar qué enseñan realmente estas especificaciones. La cantidad de núcleos es un dato relevante, pero no basta por sí sola para

comprender el comportamiento de una GPU. También importan la memoria disponible, el ancho de banda, la posibilidad de comunicar varias GPU entre sí y, sobre todo, el tipo de carga que se quiere ejecutar. Dicho de forma simple, una GPU no se elige solo por tener "más cores", sino por la relación entre arquitectura, memoria y clase de problema. Con este marco, resulta más claro por qué el estudio de CUDA, de la jerarquía de memoria y de la organización en threads, blocks y grids es necesario para entender cómo se aprovecha realmente este hardware.

8.4. CUDA como modelo de programación

CUDA, sigla de Compute Unified Device Architecture, es el modelo desarrollado por NVIDIA para utilizar GPU en cálculos de propósito general. En este capítulo, la ejecución se organiza mediante tres conceptos centrales: grid, block y thread.

Los hilos se agrupan en bloques y los bloques en grillas. Esta organización permite distribuir el trabajo sobre el dispositivo y adaptar la ejecución al problema planteado. La elección de dimensiones adecuadas para bloques y grillas constituye una parte importante del diseño y del análisis de rendimiento.

Un modo simple de leer esta estructura es el siguiente:

- cada `thread` ejecuta el mismo kernel sobre una parte del problema;
- varios `threads` se agrupan en un `block`, que comparte ciertos recursos del hardware;
- el conjunto de bloques forma el `grid`, que representa la ejecución total del kernel.

En este contexto, conviene introducir una noción básica: un kernel es la función que se ejecuta sobre la GPU. A diferencia de una función secuencial tradicional, no se invoca para producir una sola trayectoria de ejecución, sino para que una gran cantidad de hilos ejecuten el mismo código sobre datos distintos. Dicho de forma simple, el kernel contiene la lógica de la operación que se quiere aplicar masivamente, mientras que la grilla y los bloques determinan cuántas veces y cómo se distribuye esa ejecución sobre el hardware.

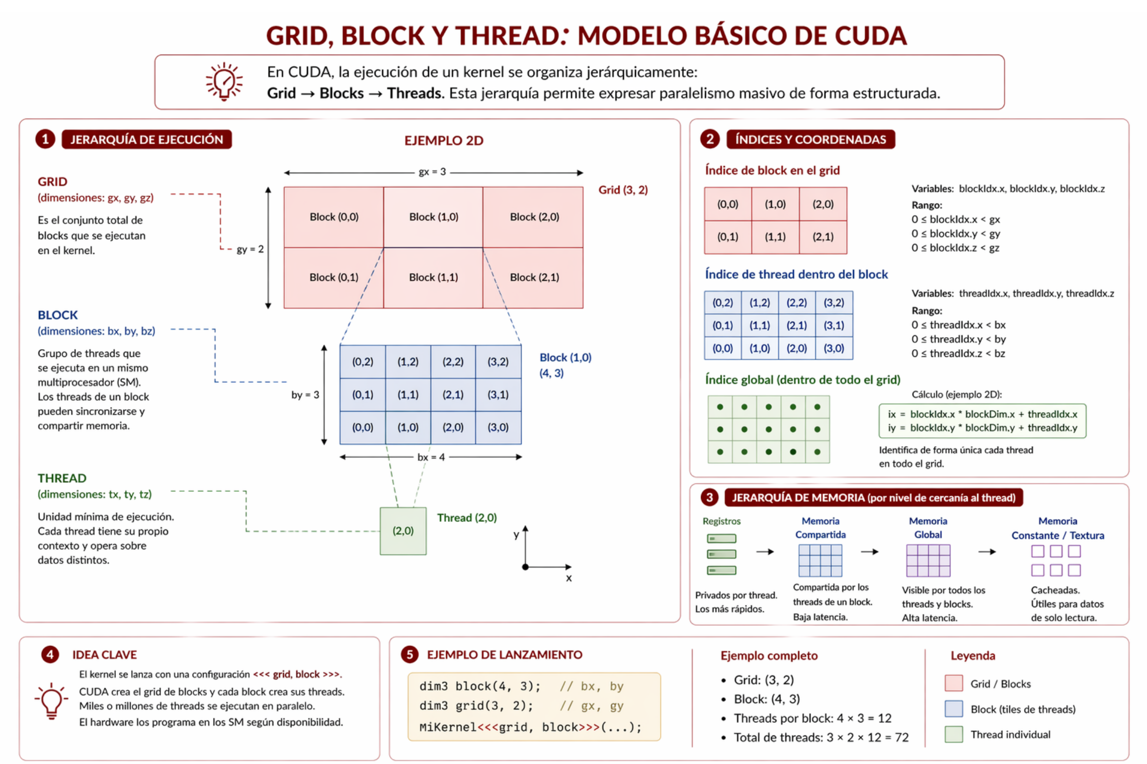

Figura 8.2: Organización jerárquica de la ejecución en CUDA: grid, bloques y threads.

8.5. Alternativas a CUDA

Aunque CUDA ocupa un lugar importante en la introducción a la programación sobre GPU, no constituye la única alternativa disponible. Su ventaja pedagógica radica en que permite presentar con claridad ideas como `thread`, `block`, `grid`, jerarquía de memoria y configuración de ejecución dentro de un ecosistema muy consolidado. Sin embargo, existen otros caminos para trabajar con aceleradores, especialmente cuando se busca portabilidad entre fabricantes o un nivel de abstracción diferente.

Una familia importante de alternativas está formada por modelos más portables, como OpenCL o SYCL, que procuran ofrecer una vía de programación menos atada a un único proveedor de hardware. También existen bibliotecas y frameworks de más alto nivel, como PyTorch o JAX, que permiten expresar operaciones sobre tensores sin escribir kernels de bajo nivel en cada caso. En esos entornos, buena parte del trabajo se delega a bibliotecas optimizadas que luego se ejecutan sobre GPU cuando el hardware y el backend lo permiten.

Por ese motivo, conviene entender la elección de CUDA en este capítulo como una decisión didáctica y no como una afirmación de exclusividad. Aquí se lo utiliza porque ayuda a hacer visibles los conceptos fundamentales de la computación GPU. Más adelante, cuando aparezcan herramientas de más alto nivel, resultará más claro que muchas de ellas reutilizan estas mismas ideas, aunque las presenten con otra interfaz o con un mayor grado de abstracción.

8.6. Un kernel mínimo con Numba

En este capítulo se introduce el uso de GPU desde Python mediante Numba. Esta estrategia resulta adecuada para el enfoque didáctico del libro, ya que permite acercarse a conceptos de GPU sin abandonar el lenguaje principal de trabajo. Al mismo tiempo, Numba conserva de manera bastante directa el modelo de ejecución asociado a CUDA: kernels, grids, blocks, threads y cálculo explícito de índices. Por ese motivo, ofrece una aproximación útil para estudiar cómo se organiza el trabajo en una GPU sin pasar de inmediato a un entorno de más bajo nivel como CUDA C o C++, ni tampoco a un entorno de más alto nivel como PyTorch donde muchos de esos detalles quedan ocultos.

Dicho de otro modo, ofrece una abstracción intermedia: reduce parte de la complejidad sintáctica, pero no oculta los conceptos fundamentales que conviene comprender en una introducción a la programación sobre GPU.

Un ejemplo mínimo, más cercano a operaciones ya vistas en capítulos anteriores, consiste en sumar dos vectores elemento a elemento:

```
from numba import cuda


@cuda.jit
def add_vectors(a, b, result):
    # Calcula el índice global del hilo en una grilla unidimensional.
    index = cuda.grid(1)
    if index < len(result):
        result[index] = a[index] + b[index]
```

En este kernel, cada hilo calcula su índice global dentro del grid y se ocupa de una sola posición de los vectores. Si el índice es válido, lee un elemento de `a`, lee el correspondiente de `b` y escribe la suma en `result`. El interés del ejemplo no está solo en la operación elegida, sino en mostrar el patrón más básico de programación en GPU: un gran conjunto de hilos ejecuta el mismo código sobre datos distintos.

Cuando se lanza el kernel, también debe decidirse la configuración de ejecución:

```
# Cantidad de hilos que tendrá cada bloque.
threads_per_block = 256

# Cantidad de bloques necesaria para cubrir los n elementos.
blocks_per_grid = (n + threads_per_block - 1) // threads_per_block

# Lanza el kernel sobre la GPU con esa configuración.
add_vectors[blocks_per_grid, threads_per_block](a, b, result)
```

En un programa ejecutable, esos arreglos deben existir en el dispositivo o copiarse hacia él antes de lanzar el kernel. Una versión mínima completa puede escribirse así:

```
import numpy as np
from numba import cuda



n = 1024
a_host = np.arange(n, dtype=np.float32)
b_host = np.ones(n, dtype=np.float32)
result_host = np.empty_like(a_host)

a_device = cuda.to_device(a_host)
b_device = cuda.to_device(b_host)
result_device = cuda.device_array_like(result_host)

threads_per_block = 256
blocks_per_grid = (n + threads_per_block - 1) // threads_per_block
```

```
add_vectors[blocks_per_grid, threads_per_block](a_device, b_device, result_device)
cuda.synchronize()

result_device.copy_to_host(result_host)
```

Este ejemplo permite distinguir tres momentos: preparar datos en CPU, copiarlos al dispositivo y ejecutar el kernel, y recuperar el resultado. Esa separación será importante más adelante al medir transferencias y cómputo.

Aquí conviene detenerse porque esta parte expresa una de las decisiones centrales de la programación GPU. `threads_per_block` indica cuántos hilos se agrupan dentro de cada bloque. En este ejemplo se elige 256, un valor frecuente en introducciones porque permite pensar una cantidad suficientemente grande de hilos por bloque sin entrar todavía en ajustes más finos de arquitectura. `blocks_per_grid`, en cambio, indica cuántos bloques harán falta para cubrir el total del problema. Si `n` representa la cantidad de elementos del vector, entonces el cálculo `(n + threads_per_block - 1) // threads_per_block` realiza, en términos prácticos, una división entera con redondeo hacia arriba.

La razón de esa fórmula es sencilla. Si hubiera exactamente 1024 elementos y cada bloque tuviera 256 hilos, alcanzarían 4 bloques. Pero si hubiera 1000 elementos, la división exacta no daría un número entero de bloques. En ese caso, también conviene reservar 4 bloques para asegurar que existan suficientes hilos como para cubrir todas las posiciones del vector. Eso significa que algunos hilos adicionales pueden quedar sin trabajo útil al final. Por ese motivo, dentro del kernel aparece la condición `if index < len(result)`: sirve para evitar que esos hilos sobrantes intenten acceder a posiciones inexistentes.

Esta lógica muestra una diferencia importante respecto de una formulación secuencial. En una CPU secuencial suele pensarse primero en el bucle y luego en el índice. En GPU, además de la operación que realiza cada hilo, hay que decidir cómo se distribuye globalmente el trabajo sobre bloques y grillas. Esa decisión influye directamente en el rendimiento porque condiciona cuántos hilos estarán disponibles, cómo se ocuparán los multiprocesadores y qué tan bien se aprovecharán los recursos del dispositivo. Elegirla bien, por lo tanto, es parte del problema de programación y no un detalle accesorio.

Conviene detenerse un momento en la anotación `@cuda.jit`. La sigla JIT significa just in time, es decir, compilación en el momento de la ejecución. En lugar de interpretar cada instrucción de Python una por una durante el cálculo, Numba traduce la función anotada a una forma ejecutable más cercana al hardware cuando esa función se utiliza. En términos generales, esta estrategia permite conservar una sintaxis accesible y, al mismo tiempo, obtener una ejecución mucho más eficiente que la de un código Python puramente interpretado.

También conviene distinguir esta anotación de otras que aparecen en Numba. `@jit` es el decorador general para compilación just in time, mientras que `@njit` equivale, en términos prácticos, a pedir compilación en modo nativo sin apoyarse en objetos de Python durante la ejecución. `@cuda.jit`, en cambio, no se usa para acelerar una función sobre CPU, sino para definir kernels y funciones asociadas al modelo de ejecución de CUDA sobre GPU. Por ese motivo, aunque los nombres se parezcan, no cumplen exactamente el mismo papel dentro del ecosistema de Numba.

8.7. Un ejemplo bidimensional: multiplicación de matrices

La suma de vectores permite introducir el caso más simple, donde cada hilo se asocia con una única posición de un arreglo unidimensional. Un paso natural consiste en extender esa idea a un problema bidimensional, como la multiplicación de matrices. Este ejemplo resulta útil porque retoma una operación ya trabajada en capítulos anteriores y, al mismo tiempo, anticipa por qué en GPU la organización de accesos a memoria pasa a ser tan importante.

Una versión introductoria, sin optimizaciones adicionales, puede escribirse así:

```
from numba import cuda


@cuda.jit
def matmul(a, b, result):
    row, col = cuda.grid(2)
    if row < result.shape[0] and col < result.shape[1]:
        total = 0.0
        for k in range(a.shape[1]):
```

```
            total += a[row, k] * b[k, col]
        result[row, col] = total
```

Aquí la idea general sigue siendo la misma, pero ahora cada hilo se asocia con una posición `(row, col)` de la matriz resultado. Para obtener ese par de coordenadas se usa `cuda.grid(2)`, que calcula el índice global del hilo en una grilla bidimensional. Si la posición calculada cae dentro de los límites de la matriz, el hilo recorre la dimensión interna `k`, acumula los productos parciales y finalmente escribe un único elemento del resultado.

Este ejemplo permite ver con más claridad cómo cambia la complejidad del problema. En la suma de vectores, cada hilo solo necesita leer dos valores y escribir uno. En la multiplicación de matrices, en cambio, cada hilo debe leer una fila de `a`, una columna de `b` y realizar varias operaciones antes de producir su resultado. Por ese motivo, este tipo de kernel ayuda a entender por qué más adelante será necesario prestar atención no solo a la partición del trabajo, sino también a la jerarquía de memoria, al patrón de accesos y al costo de reutilizar datos.

8.8. Tipos de memoria en GPU

Uno de los motivos por los que dos kernels correctos pueden rendir de forma muy distinta es la jerarquía de memoria del dispositivo. En un nivel introductorio, conviene distinguir al menos estas regiones:

- memoria global: es la memoria principal de la GPU. Tiene gran capacidad, pero también una latencia relativamente alta;
- memoria compartida: es una memoria pequeña y rápida, visible para los hilos de un mismo bloque;
- memoria local: corresponde a datos privados de cada hilo, aunque puede terminar respaldada en memoria global según el uso de registros;
- memoria constante: está pensada para datos de solo lectura que se reutilizan de manera uniforme por muchos hilos.

En términos generales, una estrategia eficiente intenta minimizar accesos costosos a memoria global y reutilizar datos cuando sea posible mediante registros o memoria compartida. Esta obser-

vación conecta directamente con la jerarquía de memoria estudiada en CPU, aunque aquí adquiere todavía mayor relevancia.

8.9. Coalescing

El coalescing describe una situación en la que hilos cercanos acceden a posiciones de memoria global cercanas entre sí. Cuando esto ocurre, el hardware puede agrupar esos accesos y utilizarlos de manera más eficiente.

Si, en cambio, cada hilo accede a posiciones dispersas o poco regulares, la GPU necesita más transacciones de memoria y el kernel pierde rendimiento. Por eso, no basta con que el trabajo esté paralelizado: también importa cómo se recorren y se distribuyen los datos.

8.10. Warp divergence

En muchas GPU, los hilos se ejecutan internamente en grupos llamados warps. En términos simples, un warp es un conjunto de hilos que avanza de manera coordinada. Si todos siguen el mismo camino de ejecución, el hardware trabaja de forma eficiente. Si distintas ramas condicionales hacen que algunos hilos tomen un camino y otros otro, aparece la llamada warp divergence.

La divergencia no significa que el kernel esté mal, pero sí que parte del paralelismo efectivo se pierde porque el warp debe resolver caminos distintos en momentos diferentes. Por eso, en GPU conviene prestar atención a condiciones muy irregulares o a kernels donde los hilos de un mismo grupo no realizan operaciones similares.

8.11. Occupancy

La occupancy puede entenderse, en este nivel, como la relación entre la cantidad de hilos activos en un multiprocesador de la GPU y la cantidad máxima que ese multiprocesador podría sostener. Una occupancy alta suele ayudar a ocultar latencias de memoria, porque mientras algunos hilos esperan datos otros pueden seguir ejecutándose.

Sin embargo, una occupancy alta no garantiza por sí sola un mejor rendimiento. Si el kernel usa demasiados registros, demasiada memoria compartida o presenta malos accesos a memoria, el desempeño puede seguir siendo pobre. Por ese motivo, la occupancy debe leerse como un indicador útil, pero no como un objetivo absoluto.

8.12. Configuración del kernel y rendimiento observado

Uno de los problemas prácticos más importantes al trabajar con GPU consiste en elegir la configuración de `threads_per_block` y `blocks_per_grid`. Si el bloque es demasiado pequeño, puede desaprovechar recursos. Si es demasiado grande, puede limitar la cantidad de bloques activos o generar presión excesiva sobre registros y memoria compartida.

Por ese motivo, dos decisiones de configuración pueden producir tiempos muy distintos aun cuando el kernel sea exactamente el mismo. Esto explica por qué el ajuste de parámetros forma parte del análisis de performance y no debe tratarse como una elección arbitraria.

En términos introductorios, conviene quedarse con esta idea: el rendimiento en GPU depende de la interacción entre cómputo, memoria y configuración de ejecución. No alcanza con que el código "corra en la GPU".

8.13. PyTorch sobre GPU

Después de la aproximación con Numba, conviene pasar a una vía de más alto nivel. En este punto ya no hace falta presentar desde cero la idea de tensor, porque ese trabajo se introdujo en el capítulo anterior. Lo importante es ubicar a PyTorch como una abstracción de cómputo paralelo sobre tensores: el programa expresa operaciones sobre arreglos multidimensionales y la biblioteca decide cómo ejecutarlas de manera eficiente sobre CPU o GPU.

Lo nuevo aquí es el dispositivo: cómo se trasladan tensores a un acelerador, cómo se ejecutan allí las operaciones y por qué el costo total de una solución GPU depende no solo del cómputo, sino también de las transferencias. Esta mirada evita reducir PyTorch a un framework de aprendizaje profundo. Aunque ese sea uno de sus usos más conocidos, para este capítulo interesa sobre todo como herramienta para formular cálculo paralelo de alto nivel.

Desde el punto de vista pedagógico, este pasaje muestra otra escala de abstracción. Con Numba, el foco está en kernels, índices y configuración de ejecución. Con PyTorch GPU, en cambio, el foco se desplaza hacia el trabajo sobre tensores y hacia la decisión de dónde se ejecuta el cálculo. El cambio no invalida lo anterior: permite contrastar dos niveles de trabajo sobre el mismo tipo de hardware.

En entornos reales, esta segunda vía tiene un papel importante en aprendizaje automático, procesamiento de imágenes y cálculo numérico sobre tensores. Muchas aplicaciones no escriben kernels a mano, sino que delegan gran parte de la ejecución a bibliotecas optimizadas que internamente aprovechan la GPU.

8.14. Paralelismo implícito en CPU

Antes de pasar al acelerador, conviene señalar que PyTorch también puede aprovechar paralelismo en CPU. Muchas operaciones sobre tensores no se ejecutan como bucles Python elemento por elemento, sino mediante bibliotecas optimizadas de álgebra lineal y procesamiento numérico. Según la instalación y la plataforma, pueden intervenir implementaciones basadas en OpenMP, BLAS, MKL, oneDNN u otras bibliotecas equivalentes.

Por ese motivo, un programa escrito con una apariencia secuencial puede activar trabajo paralelo por debajo de la interfaz de Python:

```
import torch

torch.set_num_threads(1)
a = torch.rand(5000, 5000)
b = torch.rand(5000, 5000)
c = a @ b
```

El mismo cálculo puede repetirse modificando la cantidad de hilos disponibles:

```
torch.set_num_threads(8)
c = a @ b
```

La comparación no busca fijar un número universal de hilos, sino mostrar una idea conceptual: el usuario expresa una multiplicación de matrices, pero la implementación puede repartir internamente el trabajo entre varios núcleos de CPU. Esto conecta a PyTorch con lo estudiado previamente sobre NumPy, vectorización y bibliotecas numéricas optimizadas.

8.15. Dispositivo, transferencias y costo total

Un punto central al trabajar con PyTorch consiste en elegir el dispositivo de ejecución. En una introducción conviene concentrarse en los casos más habituales:

- `cpu`, para ejecución sobre procesador generalista;
- `cuda`, para GPU NVIDIA;
- `mps`, para GPU de Apple en entornos compatibles.

Estas opciones no son equivalentes en todos los detalles. `cuda` suele ofrecer el soporte más completo para GPU NVIDIA, mientras que `mps` permite aprovechar GPU de Apple pero puede tener diferencias de compatibilidad o rendimiento según la operación y la versión de PyTorch. Por ese motivo, conviene registrar siempre el dispositivo utilizado y no asumir que todos los backends se comportan igual.

Habitualmente se centraliza la elección del dispositivo en un solo lugar:

```
import torch

if torch.cuda.is_available():
    device = "cuda"
elif torch.backends.mps.is_available():
    device = "mps"
else:
    device = "cpu"
```

Esto permite conservar la misma estructura general del código y cambiar solo el lugar donde viven los tensores. Una operación simple sobre GPU puede escribirse así:

```
import torch

values = torch.tensor([1.0, 2.0, 3.0, 4.0], device=device)
result = values * 2 + 1
```

La operación es casi idéntica a la vista en CPU. Lo que cambia es el dispositivo donde se ejecuta. Esta continuidad constituye una de las principales ventajas de las bibliotecas de alto nivel: permiten mantener una interfaz estable mientras el backend decide cómo aprovechar el acelerador.

También puede hacerse explícito el movimiento entre host y dispositivo:

```
import torch

values = torch.tensor([1.0, 2.0, 3.0, 4.0])

if device != "cpu":
    gpu_values = values.to(device)
    gpu_result = gpu_values * 2 + 1
    result_back_on_cpu = gpu_result.to("cpu")
```

Aquí aparecen con claridad tres momentos distintos: datos en CPU, cómputo en GPU y devolución del resultado al host. Esta secuencia es importante porque una implementación acelerada no debe evaluarse solo por el tiempo del cálculo puro. Si la transferencia domina el tiempo total, la mejora esperada puede reducirse mucho o incluso desaparecer.

Conviene formular una regla práctica: los datos deben estar en el mismo dispositivo donde se ejecutará la operación. Si un tensor se creó en CPU y el cálculo se realizará en GPU, será necesario moverlo al dispositivo acelerador. Si varios cálculos se encadenan sobre GPU, en cambio, conviene mantener allí los tensores intermedios y evitar copias de ida y vuelta en cada paso. La transferencia de regreso a CPU solo es necesaria cuando el resultado debe usarse en un contexto que vive en el host: por ejemplo, convertirlo a NumPy, guardarlo con una biblioteca que espera datos en CPU, graficarlo, imprimir o inspeccionar valores concretos, o entregarlo a otra parte del programa que no trabaja sobre GPU.

Esta distinción también evita errores frecuentes. PyTorch no permite operar directamente, en una misma expresión, con tensores ubicados en dispositivos distintos. Si `a` está en CPU y `b` está en

GPU, antes de calcular con ambos habrá que mover uno de los dos para que compartan dispositivo. Por eso, además de preguntar si una operación puede acelerarse, debe preguntarse dónde viven sus entradas, dónde conviene que quede el resultado y qué se hará con ese resultado después.

La misma lógica puede verse en una multiplicación de matrices:

```
import torch

m1 = torch.tensor([[1.0, 2.0], [3.0, 4.0]], device=device)
m2 = torch.tensor([[5.0, 6.0], [7.0, 8.0]], device=device)
matrix_result = m1 @ m2
```

La notación sigue siendo la misma que en CPU, pero ahora el cálculo puede delegarse al acelerador. Este tipo de continuidad explica por qué PyTorch GPU ocupa un lugar tan importante en flujos contemporáneos de aprendizaje profundo y procesamiento numérico intensivo.

8.16. Ejecución asíncrona y sincronización

En GPU, muchas operaciones de PyTorch se lanzan de manera asíncrona respecto del programa Python. Esto significa que una instrucción puede solicitar una operación en el dispositivo y devolver el control al host antes de que el cálculo haya terminado efectivamente. La idea es similar a la ya vista al medir kernels CUDA: para interpretar correctamente los tiempos, no basta con registrar el reloj alrededor de la línea que lanza el trabajo.

Una medición básica en CUDA con PyTorch debe sincronizar antes de detener el temporizador:

```
from time import perf_counter
import torch

a = torch.rand(5000, 5000, device="cuda")
b = torch.rand(5000, 5000, device="cuda")

start = perf_counter()
c = a @ b
```

```
torch.cuda.synchronize()
elapsed = perf_counter() - start
```

Sin esa sincronización, el tiempo medido puede representar principalmente el costo de encolar la operación y no su ejecución completa. Esta observación es importante porque refuerza una idea central del capítulo: en GPU hay que distinguir entre lanzar trabajo, ejecutar trabajo y transferir resultados.

8.17. Operaciones por lotes y paralelismo de datos

Otra forma importante de aprovechar PyTorch consiste en formular operaciones por lotes. En lugar de resolver muchos casos pequeños con un bucle Python, suele ser preferible organizar los datos en un tensor con una dimensión adicional y aplicar una operación vectorizada o por lotes.

Por ejemplo, si se dispone de muchas multiplicaciones de matrices independientes, puede usarse `torch.bmm` para expresar el lote completo:

```
import torch

batch_size = 1024
a = torch.rand(batch_size, 32, 32, device=device)
b = torch.rand(batch_size, 32, 32, device=device)

result = torch.bmm(a, b)
```

En este caso, cada elemento del lote representa una multiplicación independiente. PyTorch puede delegar esa operación a rutinas optimizadas que aprovechan mejor el throughput del dispositivo que una secuencia de llamadas pequeñas desde Python. El punto conceptual es el mismo que atraviesa todo el capítulo: para obtener rendimiento no alcanza con tener hardware paralelo, también hay que expresar el problema de una forma que exponga suficiente paralelismo.

8.18. Experimentos prácticos con PyTorch

Para que estas ideas no queden solo en el plano conceptual, conviene plantear algunos experimentos breves. No se trata de obtener una medición universal, porque los resultados dependen del hardware, de la instalación de PyTorch y del sistema operativo. El objetivo es observar tendencias y relacionarlas con los conceptos del capítulo.

Un primer experimento consiste en comparar una operación tensorial con una operación escrita como bucle Python. La diferencia no está únicamente en la cantidad de operaciones aritméticas, sino en el lugar donde se ejecuta el trabajo principal:

```
from time import perf_counter
import torch

n = 100_000
a = torch.rand(n)
b = torch.rand(n)
c = torch.empty(n)

start = perf_counter()
for index in range(n):
    c[index] = a[index] + b[index]
elapsed_loop = perf_counter() - start

start = perf_counter()
c = a + b
elapsed_tensor = perf_counter() - start
```

La segunda forma expresa la suma como una operación completa sobre tensores. Eso permite que PyTorch delegue la ejecución a código optimizado, mientras que el bucle Python introduce una gran cantidad de pasos interpretados. Este ejemplo recupera una idea ya trabajada con NumPy: para aprovechar bibliotecas numéricas, conviene formular operaciones sobre colecciones completas de datos siempre que el problema lo permita.

Un segundo experimento permite observar el paralelismo implícito en CPU. La misma multiplicación de matrices puede medirse cambiando la cantidad de hilos disponibles:

```
from time import perf_counter
import torch

a = torch.rand(4000, 4000)
b = torch.rand(4000, 4000)

torch.set_num_threads(1)
start = perf_counter()
c = a @ b
elapsed_one_thread = perf_counter() - start

torch.set_num_threads(8)
start = perf_counter()
c = a @ b
elapsed_many_threads = perf_counter() - start
```

La comparación ayuda a discutir una situación frecuente: el código Python parece secuencial, pero la operación invocada puede estar implementada mediante rutinas paralelas de bajo nivel. El número más adecuado de hilos no es fijo; depende del procesador, de la carga del sistema y de la biblioteca usada internamente.

Un tercer experimento separa transferencia y cómputo en GPU. Esta distinción es importante porque una versión acelerada puede resultar poco conveniente si los datos se copian hacia la GPU solo para realizar una operación pequeña y volver inmediatamente a CPU:

```
from time import perf_counter
import torch

if torch.cuda.is_available():
    device = "cuda"
    a = torch.rand(5000, 5000)
    b = torch.rand(5000, 5000)
```

```
start = perf_counter()
a_gpu = a.to(device)
b_gpu = b.to(device)
torch.cuda.synchronize()
transfer_to_gpu = perf_counter() - start

start = perf_counter()
c_gpu = a_gpu @ b_gpu
torch.cuda.synchronize()
compute_on_gpu = perf_counter() - start

start = perf_counter()
c = c_gpu.to("cpu")
torch.cuda.synchronize()
transfer_to_cpu = perf_counter() - start
```

Este tipo de medición permite analizar el costo total de una estrategia GPU. Si el cómputo es grande y los datos permanecen en el acelerador para varias operaciones encadenadas, la transferencia inicial puede amortizarse. Si, en cambio, cada operación obliga a copiar datos de ida y vuelta, el costo de comunicación puede dominar.

Estos experimentos también ayudan a interpretar por qué PyTorch no debe pensarse solo como una biblioteca de redes neuronales. Desde el punto de vista de la programación paralela, ofrece una forma de expresar operaciones masivas sobre tensores, delegar ejecución a implementaciones optimizadas y razonar sobre el lugar donde viven los datos.

8.19. Numba CUDA y PyTorch GPU

Conviene comparar brevemente ambos enfoques para no confundir sus papeles dentro del libro.

Herramienta	Nivel de abstracción	Conviene usarla cuando	Lo que deja ver con más claridad
Numba CUDA	más cercano al kernel y a CUDA	interesa entender cómo se distribuye trabajo en threads, blocks y grids	organización interna de la ejecución GPU
PyTorch GPU	más cercano a tensores y operaciones de alto nivel	el problema ya se expresa como cálculo sobre tensores y se quiere usar un acelerador sin escribir kernels manuales	continuidad entre tensores y aceleración sobre GPU

Esta comparación ayuda a ubicar mejor el sentido pedagógico del capítulo. Numba CUDA permite entender cómo funciona la GPU de forma más cercana al hardware. PyTorch GPU muestra cómo muchas aplicaciones reales acceden a ese mismo hardware desde un nivel de abstracción mayor.

8.20. Continuidad del caso práctico transversal: Sobel con PyTorch GPU

Después de haber trabajado Sobel en CPU con Numba y luego sobre arreglos y tensores en CPU, conviene retomar ahora el mismo problema desde la lógica del acelerador. En este punto, el interés principal no está en redefinir el operador desde cero, sino en observar qué cambia cuando la misma formulación sobre tensores se ejecuta en GPU.

Figura 8.3: Aplicación conceptual del filtro de Sobel en GPU a partir de una formulación basada en convolución.

Una versión con PyTorch GPU puede escribirse así:

```
import torch


def sobel_torch_gpu(image, device):
    image = torch.as_tensor(image, dtype=torch.float32, device=device)
    result = torch.zeros_like(image)

    top_left = image[:-2, :-2]
    top = image[:-2, 1:-1]
    top_right = image[:-2, 2:]
    left = image[1:-1, :-2]
    right = image[1:-1, 2:]
    bottom_left = image[2:, :-2]
```

```
    bottom = image[2:, 1:-1]
    bottom_right = image[2:, 2:]

    gx = (
        -top_left + top_right
        - 2.0 * left + 2.0 * right
        - bottom_left + bottom_right
    )
    gy = (
        -top_left - 2.0 * top - top_right
        + bottom_left + 2.0 * bottom + bottom_right
    )

    result[1:-1, 1:-1] = torch.abs(gx) + torch.abs(gy)
    return result
```

La estructura del cálculo resulta familiar porque sigue la misma reformulación sobre tensores ya presentada en el capítulo anterior. Lo nuevo es que ahora esos tensores viven en el dispositivo acelerador. Esto permite ver con claridad el sentido de la progresión del libro: primero se estudió el problema de forma secuencial y compilada sobre CPU, luego se lo reformuló sobre arreglos y tensores, y ahora esa misma formulación se ejecuta sobre GPU.

También conviene señalar un límite importante. El hecho de que una versión GPU exista no implica automáticamente que siempre sea preferible. Si la imagen es pequeña o si el costo de mover datos domina el tiempo total, la mejora puede no justificar el cambio de dispositivo. En cambio, cuando el volumen de datos crece o el cálculo se encadena con otras operaciones que ya permanecen en GPU, la aceleración puede resultar mucho más significativa.

Este mismo criterio permite proyectar el ejemplo hacia una continuidad natural: si en lugar de una sola imagen se trabaja con secuencias de cuadros, el problema se acerca al procesamiento de video. Esa proyección no necesita desarrollarse aquí por completo, pero sí ayuda a ver cómo el caso práctico transversal puede escalar hacia escenarios de mayor volumen de datos.

8.21. Qué conviene observar al analizar una implementación GPU

Al estudiar o medir un kernel, conviene observar al menos estas cuestiones:

- si el problema tiene suficiente paralelismo de datos como para justificar el uso de la GPU;
- si los accesos a memoria global son regulares o dispersos;
- si los hilos de un mismo warp siguen trayectorias de ejecución similares;
- si la configuración de bloques e hilos parece razonable para el tamaño del problema;
- si el costo de copiar datos entre CPU y GPU no anula la mejora obtenida en el cómputo;
- si el cálculo permanece en GPU lo suficiente como para amortizar las transferencias.

Estas preguntas ayudan a interpretar resultados experimentales y a evitar una expectativa ingenua según la cual cualquier cálculo será automáticamente más rápido por ejecutarse en GPU.

8.22. Debugging y profiling en GPU

En GPU, una parte importante del trabajo consiste en distinguir entre un kernel incorrecto y un kernel correcto pero ineficiente. Ese diagnóstico requiere una forma de observación algo diferente de la que se usa en CPU, porque aquí intervienen también copias de datos entre host y dispositivo, configuración de ejecución y patrones de acceso a memoria.

Una primera recomendación consiste en separar conceptualmente tres tiempos distintos:

- el tiempo de preparación y copia de datos hacia la GPU;
- el tiempo de ejecución del kernel;
- el tiempo de devolución de resultados al host.

Si no se distinguen estas etapas, una medición puede llevar a conclusiones engañosas. Un kernel muy rápido puede parecer lento si el volumen de transferencia domina el tiempo total.

En una medición básica con Numba, además, conviene sincronizar explícitamente antes de detener el reloj, ya que la ejecución puede ser asincrónica:

```python
from time import perf_counter
from numba import cuda


start = perf_counter()
kernel[blocks_per_grid, threads_per_block](values)
cuda.synchronize()
elapsed = perf_counter() - start
```

Sin esa sincronización, el tiempo medido puede no reflejar la ejecución real del kernel.

Desde el punto de vista del debugging, conviene observar al menos estos síntomas:

- resultados incorrectos por índices mal calculados o límites mal controlados;
- kernels que funcionan pero rinden poco por accesos no coalesced;
- configuraciones de bloques que reducen occupancy o desaprovechan el dispositivo;
- tiempos dominados por copia de datos en lugar de por cómputo.

En un nivel introductorio, profiling y debugging en GPU no requieren todavía herramientas sofisticadas. Lo importante es aprender a formular buenas preguntas: si el cuello de botella está en memoria, en configuración, en divergencia o en transferencia, la estrategia de mejora será distinta.

8.23. Cierre de la unidad

Este capítulo permitió introducir la GPU como una plataforma de paralelismo masivo orientada a throughput, con una organización de hilos y una jerarquía de memoria propias. También mostró que el rendimiento no depende solo de ejecutar el mismo cálculo en otro hardware, sino de comprender cómo interactúan configuración, memoria, transferencia y volumen de trabajo.

Con este marco, el recorrido del libro queda ya completo en su progresión principal: paralelismo explícito en CPU, reformulación eficiente del cálculo sobre arreglos y tensores, y aceleración sobre GPU. El capítulo final reunirá esa secuencia en una síntesis general para ayudar a decidir cuándo conviene usar cada familia de herramientas.

8.24. Ejercicios del capítulo

- Explique por qué una GPU resulta adecuada para tareas altamente paralelizables.
- Describa la función de los conceptos grid, block y thread en CUDA.
- Distinga memoria global y memoria compartida en una GPU.
- Explique con sus palabras qué significan coalescing, occupancy y warp divergence.
- Justifique por qué conviene separar tiempo de transferencia y tiempo de kernel al medir rendimiento.
- Proponga una tarea que podría beneficiarse del uso de GPU y explique brevemente por qué.
- Describa un caso en el que una mala elección de `threads_per_block` podría degradar el rendimiento.
- Explique qué error puede cometerse al medir un kernel si no se sincroniza la GPU antes de registrar el tiempo final.
- Explique por qué dos kernels correctos pueden rendir de forma muy distinta en una misma GPU.
- Describa por qué una operación de PyTorch sobre CPU puede aprovechar paralelismo aunque el código Python parezca secuencial.
- Explique por qué una operación por lotes como `torch.bmm` puede resultar más adecuada que un bucle Python con muchas multiplicaciones pequeñas.
- Diseñe una medición simple en PyTorch que permita separar tiempo de transferencia y tiempo de cómputo en GPU.
- Describa qué observaría primero para diagnosticar una implementación GPU correcta que no muestra mejora de rendimiento.
- Explique qué cambia entre la versión de Sobel sobre tensores en CPU y su ejecución sobre GPU con PyTorch.
- Justifique en qué tipo de situación Sobel sobre GPU tendría más sentido que Sobel sobre CPU.

Parte III

Cierre y proyección

9 Cierre y conclusión

Al llegar a este punto, conviene recuperar la idea central que recorrió todo el libro: paralelizar no consiste solamente en usar más hardware, sino en aprender a descomponer problemas, elegir una estrategia de ejecución adecuada, medir resultados e interpretar límites reales de escalabilidad. Esa perspectiva atraviesa desde los conceptos más generales hasta los casos de vectorización y GPU.

El recorrido desarrollado buscó justamente construir esa mirada. Primero se distinguieron los conceptos fundamentales del paralelismo y su diferencia respecto de la concurrencia y la computación distribuida. Luego se introdujeron arquitecturas, métricas y límites teóricos. Sobre esa base se estudiaron modelos de programación, APIs clásicas y herramientas accesibles en Python. En la parte final del libro, esa progresión se organizó en tres movimientos complementarios: paralelismo explícito en CPU, reformulación eficiente del cálculo sobre arreglos y tensores en CPU, y aceleración sobre GPU.

En términos generales, el libro deja cuatro aprendizajes principales.

El primero es conceptual. Resulta importante distinguir con precisión entre ejecución secuencial, concurrencia, paralelismo y distribución. Esa diferenciación inicial evita muchas confusiones posteriores y permite leer con mayor claridad las decisiones de diseño.

El segundo es arquitectónico. El rendimiento de una implementación no depende solo del algoritmo, sino también del tipo de memoria, de la jerarquía de caché, del ancho de banda disponible y de los costos de sincronización. Por ese motivo, una solución correcta puede exhibir resultados muy distintos según la plataforma en la que se ejecute.

El tercero es metodológico. Medir tiempos no alcanza por sí solo. Conviene interpretar speed-up, eficiencia, fracción secuencial, acceso a memoria y sobrecarga de coordinación. Sin ese paso analítico, la comparación entre implementaciones queda incompleta.

El cuarto es práctico. En muchos problemas no existe una única forma correcta de mejorar rendimiento. A veces conviene repartir tareas de manera explícita en CPU; en otros casos, reformular el cálculo mediante vectorización o trabajo con tensores; en otros, aprovechar GPU; y en otros, aceptar que el costo de paralelizar o acelerar no justifica el beneficio esperado.

9.1. Criterios que conviene conservar

Más allá de las bibliotecas o arquitecturas puntuales, este libro intentó sostener algunos criterios generales que conviene conservar al seguir estudiando el tema.

Uno de ellos es que la elección de una estrategia debe responder a la estructura del problema. No se elige primero una herramienta y luego se busca dónde usarla. Conviene analizar si el trabajo es regular o irregular, si predomina el cómputo o el acceso a memoria, si el problema admite división natural en subtareas y qué costo introduce la coordinación entre ellas.

Otro criterio importante es que el paralelismo siempre debe leerse junto con sus límites. Las leyes de Amdahl y Gustafson, la jerarquía de memoria, la localidad de caché, el false sharing, NUMA o los costos de transferencia en GPU muestran que escalar no es simplemente agregar recursos. También exige comprender dónde aparece el cuello de botella.

Un tercer criterio tiene que ver con la observación. A lo largo del recorrido aparecieron señales de debugging y profiling en CPU y GPU. Ese punto conviene subrayarlo: en sistemas paralelos, muchos errores no son evidentes a primera vista. Un resultado no determinista, una mejora menor a la esperada o un kernel correcto pero ineficiente exigen mirar más allá del código fuente y prestar atención al comportamiento real del sistema.

Como cierre sintético del recorrido, conviene recuperar una tabla de decisión general sobre algunas de las herramientas discutidas en el libro. No debe leerse como una receta cerrada, sino como un recordatorio de que la elección siempre depende de la estructura del problema, de la arquitectura disponible y de la medición posterior.

Estrategia	Conviene usarla cuando	Ventaja principal	Límite principal
`threading`	la tarea espera entrada/salida	bajo costo y buena respuesta concurrente	no acelera bien tareas CPU-bound por el GIL

Estrategia	Conviene usarla cuando	Ventaja principal	Límite principal
`multiprocessing`	la tarea es CPU-bound y puede repartirse en bloques	evita el GIL	serialización y comunicación entre procesos
NumPy	el problema es numérico y regular sobre arreglos	vectorización eficiente con una interfaz simple	menor flexibilidad para lógica irregular
Numba CPU	se quiere acelerar código numérico en Python sin abandonar loops y estructuras explícitas	compilación JIT y paralelismo cercano al hardware sobre CPU	requiere código compatible y cierto cuidado con tipos y estructuras
Numba GPU	interesa controlar kernels, índices y configuración de ejecución sobre GPU	hace visible el modelo de ejecución del acelerador	exige mayor atención a memoria, configuración y detalles del hardware
PyTorch CPU	el problema ya se expresa de manera natural sobre tensores en CPU	continuidad clara con operaciones sobre tensores y puente natural hacia GPU	agrega una biblioteca más pesada si solo se necesita cálculo numérico simple
PyTorch GPU	el cálculo ya se formula sobre tensores y puede beneficiarse del acelerador	permite mantener una interfaz de alto nivel sobre GPU	el costo de transferencia puede anular la mejora si el volumen de trabajo es bajo

Esta síntesis no reemplaza la medición, pero ayuda a recuperar de un vistazo parte de los criterios prácticos construidos a lo largo del libro. También permite leer con más claridad la progresión final del recorrido: de repartir trabajo en CPU, a reformular el cálculo sobre estructuras de datos completas, y de allí a mover ese cálculo hacia un acelerador cuando el problema lo justifica.

En ese sentido, el caso de Sobel utilizado en la parte final del libro funciona como un buen recordatorio metodológico. El mismo problema pudo leerse como loop secuencial, como cálculo

compilado sobre CPU, como reformulación sobre arreglos y tensores, y finalmente como ejecución sobre GPU. Esa continuidad no convierte al caso en una receta universal, pero sí ayuda a ver que lo decisivo no es la herramienta aislada, sino la relación entre estructura del problema, forma de expresarlo y arquitectura elegida.

El anexo que sigue propone una forma de llevar esa continuidad al trabajo práctico. Sus actividades no agregan un nuevo marco teórico, sino que ordenan ejercicios de medición y comparación sobre problemas ya vinculados con el recorrido: suma de vectores, multiplicación de matrices, Sobel y procesamiento de video. En particular, el trabajo final con video 4K permite integrar procesamiento por frames, tensores, CPU, GPU, medición de tiempos y cuidado de memoria en una aplicación visible.

9.2. Del fundamento a la práctica contemporánea

Otro aspecto que conviene retener es que los fundamentos clásicos siguen siendo relevantes incluso cuando se trabaja con herramientas modernas. Los modelos de programación paralela, las APIs históricas y la distinción entre memoria compartida y distribuida, siguen apareciendo en bibliotecas de alto nivel, frameworks de datos y entornos de aceleración contemporáneos.

En ese sentido, aprender paralelismo no equivale a memorizar tecnologías pasajeras. Supone construir una base conceptual que permita entender por qué una herramienta funciona, en qué contexto resulta adecuada y cuáles son sus límites. Esa base es la que hace posible adaptarse a nuevas bibliotecas y nuevas plataformas sin empezar siempre desde cero.

9.3. Continuidad del estudio

Este libro presentó conceptos y herramientas contemporáneas para introducir el estudio del paralelismo desde una perspectiva a la vez conceptual y práctica. A lo largo del recorrido se abordaron modelos, métricas, arquitecturas y bibliotecas que permiten comprender cómo se descompone un problema, cómo se distribuye el trabajo y bajo qué criterios conviene evaluar el rendimiento obtenido.

Si bien aquí fue necesario separar algunos conceptos para volverlos más claros en un trayecto introductorio, conviene subrayar que, en aplicaciones reales, paralelismo, concurrencia y sistemas distribuidos suelen convivir. Un mismo sistema puede coordinar múltiples tareas concurrentes, paralelizar partes de su cómputo y, al mismo tiempo, repartir trabajo o datos entre varios nodos. Por ese motivo, el cierre de este libro no debe leerse como un punto final, sino como una base desde la cual pueden continuarse varias líneas de profundización.

Una de esas líneas de continuidad aparece en el estudio más profundo de modelos y herramientas ya introducidos de manera inicial en el recorrido. En ese marco, una posibilidad clara es avanzar hacia MPI en un nivel más desarrollado. El uso introductorio de operaciones colectivas permite comprender el modelo, pero un estudio posterior podría incorporar topologías, patrones de comunicación más complejos y escenarios más cercanos a clústeres reales.

También existe una continuidad natural hacia frameworks y modelos más recientes, como Dask, Ray o enfoques de dataflow. Estos temas no fueron desarrollados en el cuerpo principal del libro porque exceden su carácter introductorio, pero constituyen una expansión razonable para quien quiera conectar fundamentos de paralelismo con ecosistemas contemporáneos de datos y sistemas distribuidos.

Por último, puede profundizarse el estudio de GPU desde una perspectiva más cercana a la optimización fina, incorporando herramientas de profiling específicas, análisis detallado de kernels y estrategias avanzadas de memoria.

10 Anexo: trabajos prácticos integradores

Este anexo reúne propuestas de trabajo práctico para acompañar el recorrido conceptual del libro. Su propósito es convertir las ideas de descomposición, medición, speed-up, eficiencia, vectorización y uso de GPU en experiencias reproducibles de programación y análisis.

Las actividades no buscan reemplazar el desarrollo teórico de los capítulos, sino ofrecer situaciones concretas en las que sea necesario decidir una estrategia, implementar variantes comparables, medir con cuidado y explicar los resultados obtenidos. Por ese motivo, cada trabajo incluye objetivos, consignas, criterios de medición, estructura de informe y condiciones de entrega.

10.1. Criterios generales de medición

En todos los trabajos conviene definir una línea de base clara. Habitualmente será la versión secuencial, aunque en algunos casos puede utilizarse otra implementación de referencia si se justifica. El speed-up debe calcularse a partir de tiempos promedio:

$$S = \frac{T_{base}}{T_{metodo}}$$

La eficiencia o performance porcentual puede expresarse como:

$$E = \frac{S}{p} \times 100$$

cuando existe una cantidad de procesos, workers o hilos comparable. Si se informa performance en otro sentido, debe aclararse la definición utilizada.

Para medir tiempos en Python se recomienda usar `time.perf_counter()`, repetir cada ejecución varias veces y reportar promedios. Cuando se utilice Numba, debe considerarse una corrida de calentamiento para no mezclar el tiempo de compilación just in time con el tiempo de ejecución. Cuando se utilice GPU, debe sincronizarse el dispositivo antes de detener el temporizador de la etapa medida.

10.2. Trabajo práctico 1: suma paralela de un vector

10.2.1. Objetivo

El objetivo de este trabajo es comparar una versión secuencial de suma de un vector con variantes basadas en threads y procesos. El ejercicio permite observar el costo de repartir trabajo, el efecto del Global Interpreter Lock (GIL) en tareas CPU-bound y la diferencia entre concurrencia y paralelismo efectivo en Python.

10.2.2. Punto de partida

La versión secuencial puede tomar como referencia el siguiente programa. Se usa una lista de Python para mantener explícito el costo del recorrido y evitar que una biblioteca numérica realice optimizaciones que no forman parte del objetivo inicial.

```python
from time import perf_counter


def sum_elements(values):
    total = 0.0
    for value in values:
        total += value
    return total


if __name__ == "__main__":
    complexity = 1_000_000
```

```
    values = [float(index) for index in range(complexity)]

    start = perf_counter()
    result = sum_elements(values)
    elapsed = perf_counter() - start

    print("Result:", result)
    print("Time:", elapsed)
    print("Complexity:", complexity)
```

10.2.3. Consignas

Implementar y comparar las siguientes variantes:

- versión secuencial;
- versión con threads;
- versión con procesos.

Ejecutar las variantes con `p = 1, 2, 4, 8, 16`, siempre que el equipo utilizado permita esos valores de manera razonable. Utilizar al menos tres tamaños de problema: uno pequeño, uno intermedio y uno grande. Por ejemplo, pueden emplearse `c = 12`, `c = 1_000_000` y `c = 100_000_000`, ajustando el tamaño si la memoria disponible no permite completar la ejecución.

Para cada combinación se deben realizar varias corridas y reportar el tiempo promedio. La medición debe concentrarse en el cómputo y en los costos propios de la estrategia elegida, aclarando si la creación de datos queda incluida o excluida.

10.2.4. Resultados esperados

El informe deberá incluir una tabla con las siguientes columnas:

- algoritmo;
- complejidad;
- procesos, workers o threads utilizados;

- tiempo promedio;
- speed-up;
- eficiencia o performance porcentual;
- equipo utilizado.

También debe identificarse el mayor speed-up y la mayor eficiencia para cada tamaño de problema.

10.2.5. Informe

El informe deberá incluir:

- objetivo del trabajo;
- descripción breve de las variantes implementadas;
- entorno de ejecución: CPU, cantidad de núcleos físicos y lógicos, RAM, sistema operativo y versión de Python;
- metodología de medición: valores de `p`, valores de `c`, cantidad de corridas y forma de calcular los tiempos reportados;
- tabla de resultados;
- análisis de los resultados;
- conclusiones sobre los límites observados al aumentar la cantidad de workers o el tamaño del problema;
- código utilizado.

10.2.6. Entrega

La entrega deberá incluir un informe en PDF o Markdown y el código utilizado para generar los resultados.

10.3. Trabajo práctico 2: multiplicación de matrices

10.3.1. Objetivo

El objetivo de este trabajo es comparar distintas estrategias para multiplicar matrices cuadradas en Python. La multiplicación de matrices permite observar el costo computacional de un algoritmo regular, la importancia de la localidad de memoria, el efecto de la transposición, el límite de `threading` en cargas CPU-bound y la diferencia entre implementar paralelismo manual o delegar trabajo a bibliotecas optimizadas.

10.3.2. Implementaciones requeridas

Implementar y comparar las siguientes versiones:

- secuencial tradicional;
- secuencial con matriz transpuesta;
- `threading`;
- `multiprocessing`;
- NumPy;
- Numba CPU;
- PyTorch CPU;
- PyTorch GPU, si el hardware disponible lo permite.

Todas las versiones deben calcular el mismo resultado para una misma entrada. Para verificarlo, debe reportarse un checksum o una métrica equivalente. Si los checksums difieren entre métodos para la misma complejidad, se debe revisar la implementación antes de interpretar tiempos.

10.3.3. Consignas

Ejecutar el benchmark con al menos las siguientes combinaciones:

Workers	Complejidad
1	512
4	512

	Workers	Complejidad
	4	1024
Núcleos físicos disponibles		1024

Usar una semilla fija para generar datos reproducibles. Registrar tiempos, checksums, speed-ups y eficiencias. En PyTorch GPU, sincronizar el dispositivo antes de detener el temporizador.

10.3.4. Resultados esperados

Armar una tabla de resultados para cada combinación ejecutada. La tabla deberá incluir:

- método;
- tiempo promedio en segundos;
- checksum;
- speed-up;
- performance o eficiencia porcentual.

La versión secuencial tradicional debe utilizarse como línea de base, salvo que el informe justifique explícitamente otra referencia.

10.3.5. Preguntas de análisis

Responder brevemente:

- ¿Por qué `threading` no necesariamente mejora en esta carga de trabajo?
- ¿Qué explica la mejora de las versiones con matriz transpuesta?
- ¿El speed-up de `multiprocessing` es lineal con la cantidad de workers? ¿Por qué?
- ¿Qué aporta `numba.njit` incluso cuando no se activa paralelismo explícito?
- ¿Puede aparecer una eficiencia superior al 100 %? ¿Cómo debe interpretarse?
- ¿Qué diferencias aparecen entre NumPy, Numba y PyTorch para este problema?

10.3.6. Informe

El informe deberá incluir:

- objetivo del trabajo;
- descripción breve del problema y de las variantes implementadas;
- entorno de ejecución: CPU, núcleos físicos y lógicos, RAM, sistema operativo, versión de Python, versiones de bibliotecas relevantes y GPU si corresponde;
- metodología de medición: combinaciones ejecutadas, semilla utilizada, cantidad de corridas y forma de calcular los tiempos reportados;
- tablas de resultados;
- verificación de checksum;
- análisis a partir de las preguntas propuestas;
- conclusiones sobre qué estrategia resultó más adecuada y bajo qué condiciones;
- código utilizado.

10.3.7. Entrega

La entrega deberá incluir un informe en PDF o Markdown y el código utilizado para generar los resultados.

10.4. Trabajo práctico 3: filtro de Sobel para detección de bordes

10.4.1. Objetivo

El objetivo de este trabajo es implementar y comparar distintas versiones del filtro de Sobel. El caso permite estudiar un patrón de cómputo regular sobre imágenes, basado en vecindades locales de `3 x 3`, y analizar cómo cambian las decisiones de implementación al pasar de loops explícitos a NumPy, Numba, PyTorch y GPU.

El filtro de Sobel estima cambios de intensidad en una imagen en escala de grises mediante dos máscaras de convolución. Una máscara aproxima variaciones horizontales y la otra variaciones verticales. Para cada píxel interior se toma una vecindad `3 x 3`, se aplican ambas máscaras

mediante productos elemento a elemento y sumas, y luego se combinan los resultados parciales para obtener una magnitud de borde.

En términos de cálculo, si la intensidad se representa como `I(x, y)`, el gradiente puede escribirse como:

$$\nabla I = \left(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right)$$

En imágenes digitales, esas derivadas se aproximan con diferencias finitas y máscaras discretas. Sobel combina derivación y suavizado local mediante pesos como `1-2-1`, lo que permite detectar bordes de manera simple y suficientemente estable para un trabajo introductorio.

10.4.2. Conversión a escala de grises

El trabajo debe aplicarse sobre imágenes en escala de grises. Si la imagen original está en RGB, debe convertirse a una única intensidad por píxel. Un criterio habitual de luminancia es:

$$I = 0.299R + 0.587G + 0.114B$$

El criterio matemático de conversión debe mantenerse equivalente entre implementaciones para que los tiempos y la métrica de salida sean comparables.

Una versión secuencial pura puede escribirse así:

```
# rgb: lista 2D de pixeles, cada pixel como tupla o lista (R, G, B)
gray = []
for row in rgb:
    gray_row = []
    for red, green, blue in row:
        intensity = int(0.299 * red + 0.587 * green + 0.114 * blue)
        intensity = 0 if intensity < 0 else 255 if intensity > 255 else intensity
        gray_row.append(intensity)
    gray.append(gray_row)
```

Una versión vectorizada equivalente con NumPy puede escribirse así:

```
import numpy as np

# rgb: arreglo HxWx3 con canales R, G, B
rgb = np.asarray(img_rgb, dtype=np.float32)
gray = 0.299 * rgb[:, :, 0] + 0.587 * rgb[:, :, 1] + 0.114 * rgb[:, :, 2]
gray_u8 = np.clip(gray, 0, 255).astype(np.uint8)
```

10.4.3. Ejemplo mínimo de Sobel

```
import numpy as np

image = np.array([
    [10, 10, 10, 10],
    [10, 80, 80, 10],
    [10, 80, 80, 10],
    [10, 10, 10, 10],
], dtype=np.float32)

kernel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], dtype=np.float32)
kernel_y = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]], dtype=np.float32)

patch = image[0:3, 0:3]
gx = np.sum(patch * kernel_x)
gy = np.sum(patch * kernel_y)
gradient = np.sqrt(gx * gx + gy * gy)

print(f"gx={gx:.2f}, gy={gy:.2f}, |grad|={gradient:.2f}")
```

10.4.4. Etapa 1: CPU secuencial, Numba CPU y NumPy

Implementar y comparar:

- versión secuencial;
- versión Numba paralela sobre CPU;
- versión NumPy.

Ejecutar las tres versiones sobre imágenes de al menos estos tamaños:

- `750 x 750`;
- `1500 x 1500`;
- `3000 x 3000`;
- `6000 x 6000`, si la memoria disponible lo permite.

Medir por separado el tiempo de conversión RGB a gris y el tiempo de aplicación de Sobel. También debe reportarse el tiempo total como suma de ambos. La carga y guardado en disco no deben incluirse en esos tiempos.

10.4.5. Etapa 2: Numba GPU

Implementar una versión Numba GPU del filtro de Sobel. La medición debe separar, cuando corresponda:

- tiempo de transferencia CPU a GPU;
- tiempo de conversión RGB a gris, si se realiza en GPU;
- tiempo de aplicación de Sobel;
- tiempo de transferencia GPU a CPU;
- tiempo total considerado para la comparación.

Antes de detener el temporizador de una operación GPU debe sincronizarse el dispositivo. El informe debe aclarar qué tiempos se incluyen en el speed-up reportado.

10.4.6. Etapa 3: PyTorch CPU y PyTorch GPU

Implementar y comparar:

- versión PyTorch CPU;
- versión PyTorch GPU, si el hardware disponible lo permite.

La versión PyTorch debe mantener el mismo criterio de conversión a escala de grises y una formulación equivalente del filtro. Si se utiliza convolución mediante tensores, el informe debe explicar brevemente cómo se representan las máscaras de Sobel.

10.4.7. Métrica de salida

Como métrica simple de salida se utilizará el porcentaje de píxeles blancos de la imagen Sobel:

$$\%\, blancos = \frac{\text{cantidad de pixeles con valor } 255}{\text{cantidad total de pixeles}} \times 100$$

Esta métrica no reemplaza la inspección visual, pero permite comparar rápidamente si las variantes producen resultados compatibles.

10.4.8. Resultados esperados

Para cada tamaño de imagen, armar una tabla con las siguientes columnas:

- método;
- tiempo RGB a gris en segundos;
- tiempo Sobel en segundos;
- tiempo total en segundos;
- porcentaje de píxeles blancos;
- speed-up;
- performance o eficiencia porcentual.

Los tiempos, speed-ups y porcentajes deben calcularse a partir de promedios de varias corridas.

10.4.9. Preguntas de análisis

Responder brevemente:

- ¿Qué diferencias de tiempo se observan entre secuencial, NumPy y Numba CPU?
- ¿Cómo evoluciona el speed-up al aumentar el tamaño de imagen?

- ¿Las salidas son equivalentes en términos visuales y en porcentaje de píxeles blancos?
- ¿En qué tamaños de imagen se amortiza el costo de transferir datos entre CPU y GPU?
- ¿Cómo se comparan Numba GPU y PyTorch GPU para este problema?
- ¿Qué factores podrían explicar diferencias entre métodos aun cuando implementan el mismo filtro?

10.4.10. Informe

El informe deberá incluir:

- objetivo del trabajo;
- descripción breve del filtro de Sobel y de las variantes implementadas;
- entorno de ejecución: CPU, núcleos físicos y lógicos, RAM, sistema operativo, versión de Python, versiones de bibliotecas relevantes y GPU si corresponde;
- metodología de medición: tamaños de imagen, cantidad de corridas, forma de calcular promedios, exclusión de I/O y sincronización en GPU;
- tablas de resultados por tamaño de imagen;
- comparación visual breve de las salidas;
- análisis a partir de las preguntas propuestas;
- conclusiones sobre qué estrategia resultó más adecuada y bajo qué condiciones;
- código utilizado.

10.4.11. Entrega

La entrega deberá incluir un informe en PDF o Markdown, el código utilizado y una muestra representativa de las imágenes de salida.

10.5. Trabajo práctico 4: procesamiento de video 4K con PyTorch

10.5.1. Objetivo

El objetivo de este trabajo final es integrar el recorrido práctico en una aplicación visible: procesar un video 4K de aproximadamente 30 segundos mediante un filtro elegido por cada estudiante

o grupo. El trabajo combina procesamiento de imágenes, tensores, CPU, GPU, benchmarking y manejo cuidadoso de memoria.

El video debe procesarse frame por frame o en lotes pequeños. Cada frame se trata como una imagen estática. Una vez procesados los frames, debe reconstruirse el video final y reincorporarse el audio original si el material de entrada lo contiene.

10.5.2. Filtros posibles

El filtro elegido debe poder implementarse tanto en versión secuencial como en PyTorch. Algunas opciones posibles son:

- escala de grises o luminancia;
- desenfoque gaussiano;
- realce de nitidez;
- emboss o efecto relieve;
- detección de bordes con Laplaciano;
- Prewitt o Scharr;
- sepia;
- inversión de colores;
- umbralización binaria;
- posterización;
- ajuste de brillo y contraste;
- viñeta simple;
- pixelado por bloques;
- detección de movimiento simple mediante diferencia entre frames consecutivos.

El filtro elegido debe mantenerse equivalente entre implementaciones para que la comparación sea válida.

10.5.3. Implementaciones requeridas

Desarrollar al menos estas versiones:

- versión secuencial en Python, utilizada como línea de base;

- versión PyTorch sobre CPU;
- versión PyTorch sobre GPU, si el hardware disponible lo permite.

Si no se dispone de GPU compatible, el informe debe indicarlo explícitamente y comparar la versión secuencial con PyTorch CPU.

10.5.4. Manejo de memoria

No debe cargarse el video completo en memoria. Un frame 4K de `3840 x 2160` píxeles con tres canales en `uint8` ocupa cerca de 25 MB sin compresión. Si se convierte a `float32`, puede superar los 95 MB por frame. A 30 cuadros por segundo, un video de 30 segundos contiene alrededor de 900 frames. Por ese motivo, almacenar todos los frames y sus tensores intermedios puede requerir decenas de GB de memoria.

La estrategia recomendada es procesar el video como flujo:

- leer un frame;
- convertirlo al formato requerido por la implementación;
- aplicar el filtro;
- escribir el frame procesado;
- liberar referencias a arreglos o tensores intermedios;
- continuar con el siguiente frame.

También puede trabajarse con lotes pequeños si la memoria disponible lo permite. En ese caso, el informe debe justificar el tamaño de lote utilizado. En GPU, debe transferirse al dispositivo solo el frame o lote actual y evitar mantener el video completo en memoria del acelerador.

Para reconstruir el video, conviene separar el filtrado de la reincorporación del audio. Una estrategia habitual es generar primero el video procesado y luego combinarlo con la pista de audio original mediante una herramienta externa como `ffmpeg`. Ese paso debe documentarse, pero no debe mezclarse con el tiempo de filtrado si el objetivo del benchmark es comparar implementaciones de cómputo.

10.5.5. Benchmarking

El informe deberá medir y reportar:

- tiempo de lectura o decodificación de frames;
- tiempo de filtrado;
- tiempo de escritura o codificación;
- tiempo total del pipeline;
- cantidad de frames procesados;
- resolución del video;
- cuadros por segundo del video original;
- frames por segundo efectivos del procesamiento;
- herramienta o codec utilizado para reconstruir el video;
- uso aproximado de memoria, si puede registrarse;
- speed-up de PyTorch CPU y PyTorch GPU respecto de la versión secuencial.

Cuando se use GPU, debe sincronizarse antes de detener el temporizador de la etapa de filtrado. Si se mide el pipeline completo, el informe debe aclarar que el tiempo incluye costos de lectura, escritura y reconstrucción del video.

10.5.6. Informe

El informe deberá incluir:

- descripción del video utilizado: resolución, duración, frames por segundo y cantidad total de frames;
- descripción del filtro elegido y justificación breve;
- entorno de ejecución: CPU, núcleos físicos y lógicos, RAM, sistema operativo, versión de Python y GPU si corresponde;
- explicación de las implementaciones desarrolladas;
- metodología de medición;
- tabla de resultados;
- análisis de speed-up y límites observados;
- explicación del manejo de memoria;
- descripción del proceso de reconstrucción del video y reincorporación de audio;

- conclusiones sobre la estrategia más adecuada para el caso estudiado;
- código utilizado.

10.5.7. Entrega

La entrega deberá incluir un informe en PDF o Markdown, el código utilizado y el video procesado resultante.

Version 1.3

30 de mayo de 2026