CAGE – Grupo de Investigación

SC3 – Super Cómputo y Cálculo Científico – UIS

@sc3uis

Introducción a C y C++

De la primera línea al C++ moderno (C++23)

Módulo 3 – Semillero CAGE

Luis Alejandro Torres Niño | sc3.operaciones@uis.edu.co

Objetivos del módulo

- Entender el recorrido **código fuente** → **ejecutable** y compilar con `gcc`/`g++` en el cluster.
- Dominar lo básico de C/C++: tipos, control de flujo, funciones, archivos y `Makefile`.
- Comprender **memoria**: arreglos, punteros, *stack* vs *heap*, fugas.
- Usar la **STL** (`vector`, `array`, `map`, algoritmos y lambdas).
- Conocer el C++ moderno: `C++11` → `C++23` y qué viene en `C++26`.
- Desarrollar **intuición de rendimiento**: localidad de caché y banderas de optimización (puente hacia OpenMP, MPI y CUDA).

Agenda

1. Del código al ejecutable
2. Fundamentos del lenguaje
3. Funciones, cabeceras y `Makefile`
4. Memoria: arreglos, punteros, *stack* y *heap*
5. La biblioteca estándar (STL)
6. C++ moderno: C++11 a C++23 (y C++26)
7. Pensar en HPC: localidad y optimización
8. Retos individuales y reto grupal

1
Del código al ejecutable

C y C++: ¿en qué se parecen y en qué no?

C

- Lenguaje procedural (1972).
- Sin clases, sin sobrecarga, sin plantillas.
- Cadenas = arreglos de `char`.
- Base de kernels, drivers y bibliotecas numéricas clásicas (BLAS, LAPACK).

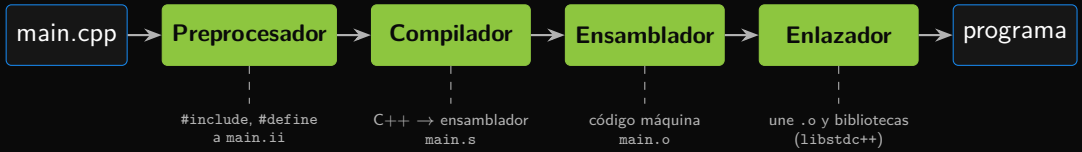
C++

- Multiparadigma (1985): procedural, objetos, genérico, funcional.
- Clases, plantillas, `std::string`, STL.
- **RAII**: los recursos se liberan solos.
- Estándar ISO cada 3 años desde 2011.

★ En HPC

Casi todo el software científico del cluster (GROMACS, LAMMPS, OpenFOAM, CP2K) está escrito en C, C++ o Fortran. OpenMP, MPI y CUDA se usan desde C/C++.

Las cuatro etapas de la compilación




```
g++ -E main.cpp -o main.ii      # solo preprocesar
g++ -S main.cpp                 # hasta ensamblador (main.s)
g++ -c main.cpp                 # hasta objeto (main.o)
g++ main.o -o programa          # enlazar
```

Compilar en el cluster

```
ssh usuario@toctoc.sc3.uis.edu.co      # bastion
ssh guane                              # frontend del cluster

module avail gnu                       # versiones disponibles
module load gnu14                      # GCC 14.2.0  (C++23)
g++ --version
```

Banderas esenciales

- ➔ `-std=c++23` (o `c++17`, `c++20`…)
- ➔ `-Wall -Wextra`: activa advertencias **siempre**
- ➔ `-O2`: optimización razonable
- ➔ `-g`: símbolos de depuración
- ➔ `-o nombre`: nombre del ejecutable

★ Versiones

`gnu14` = GCC 14.2.0, `gnu15` = GCC 15.2.0 (`<mdspan>`, *pack indexing*). C se compila con `gcc`, C++ con `g++`.

Hola mundo: C y C++

hola.c

```c
#include <stdio.h>

int main(void) {
    printf("Hola, HPC!\n");
    return 0;
}
```

```
gcc -Wall hola.c -o hola_c
```

hola.cpp

```cpp
#include <iostream>

int main() {
    std::cout << "Hola, HPC!\n";
    return 0;
}
```

```
g++ -std=c++23 -Wall hola.cpp -o hola
```

- `main` es el punto de entrada; el valor de retorno es el *código de salida* (`echo $?`).
- `std::cout` pertenece al espacio de nombres `std`; evita `using namespace std;` en cabeceras.

Reto individual 1: tu primer programa

Reto 1 – Calentamiento

1. Conéctate a `guane`, carga `gnu14` y compila `hola.c` y `hola.cpp`.
2. Modifica el programa para que imprima tu nombre y el nombre del nodo (`gethostname` de `<unistd.h>`).
3. Ejecútalo en un nodo de cómputo: `srun -p legacy -n 1 ./hola`. ¿Qué nodo imprime?
4. Genera `hola.ii`, `hola.s` y `hola.o` con `-E`, `-S`, `-c`. ¿Cuántas líneas tiene el `.ii` de `hola.cpp`? ¿Por qué tantas?

2
Fundamentos del lenguaje

Tipos de datos fundamentales

Tipo	Típico en x86-64	Uso
`char`	1 byte	carácter / byte
`int`	4 bytes	entero con signo
`long long`	8 bytes	enteros grandes
`float`	4 bytes	real simple (~7 dígitos)
`double`	8 bytes	real doble (~16 dígitos)
`bool`	1 byte	verdadero / falso
`size_t`	8 bytes	tamaños e índices (sin signo)

★ Tamaño exacto

El estándar no fija el tamaño de `int`. Si importa, usa `<cstdint>`: `int32_t`, `int64_t`, `uint8_t`. Consúltalo con `sizeof(T)` y `std::numeric_limits<T>`.

Trampas numéricas

```cpp
#include <iostream>
#include <limits>

int main() {
    int a = 3, b = 2;
    std::cout << a / b << "\n";                    // 1   (division entera)
    std::cout << static_cast<double>(a) / b << "\n"; // 1.5
    std::cout << static_cast<int>(3.9) << "\n";    // 3   (trunca)

    std::cout << (0.1 + 0.2 == 0.3) << "\n";       // 0   (falso!)
    std::cout << std::numeric_limits<int>::max() << "\n";
}
```

❄ Punto flotante

`0.1 + 0.2 != 0.3`: los reales se representan en binario con error. Compara con tolerancia: `std::abs(x-y) < 1e-9`. Esto importa cuando se paralelizan sumas (el orden cambia el resultado).

Entrada y salida estándar

```cpp
#include <iostream>

int main() {
    double altura_m;
    std::cout << "Altura en metros: ";
    std::cin >> altura_m;                  // lee del teclado

    double pies = altura_m * 3.28084;
    std::cout << altura_m << " m = " << pies << " ft\n";
}
```

- ➔ `std::cout` salida, `std::cin` entrada, `std::cerr` errores (sin *buffer*).
- ➔ En C: `printf`/`scanf` con formatos (`%d`, `%f`, `%s`).
- ➔ En `sbatch` la salida va a `slurm-<jobid>.out` (módulo 1), por lo que la entrada interactiva no aplica: usa **argumentos** (`argc, argv`) o archivos.

Control de flujo

```cpp
if (n % 2 == 0) {
    std::cout << "par\n";
} else if (n > 100) {
    std::cout << "impar grande\n";
} else {
    std::cout << "impar\n";
}

switch (opcion) {
  case 1: /* ... */ break;
  case 2: /* ... */ break;
  default: break;
}
```

```cpp
for (int i = 0; i < 10; ++i) {
    suma += i;
}

int j = 0;
while (j < 10) { j += 2; }

do {
    leer();
} while (!valido);

for (int x : {1, 2, 3}) {   // range-for
    std::cout << x << " ";
}
```

`==` compara, `=` asigna. `break` sale del bucle; `continue` salta a la siguiente iteración.

Archivos: escribir y leer

```cpp
#include <fstream>
#include <iostream>
#include <string>

int main() {
    {   // escribir
        std::ofstream out("datos.txt");          // use std::ios::app para anexar
        for (int i = 1; i <= 5; ++i) out << i << " " << i * i << "\n";
    }                                            // se cierra solo (RAII)

    std::ifstream in("datos.txt");               // leer
    int n, cuadrado;
    while (in >> n >> cuadrado)
        std::cout << n << "^2 = " << cuadrado << "\n";
}
```

- Verifica siempre la apertura: `if (!in) { std::cerr << "no abre\n"; return 1; }`.
- En el cluster: escribe resultados en tu `$HOME` o en el *scratch*, no en `/tmp`.

Reto individual 2: manejo de datos

■ Reto 2 – Estadística de un archivo

Escribe un programa `stats.cpp` que:

1. Reciba el nombre de un archivo por línea de comandos (`argv[1]`).
2. Lea todos los números reales del archivo.
3. Imprima **cantidad, mínimo, máximo, media y desviación estándar**.
4. Termine con código `1` y mensaje en `std::cerr` si el archivo no existe o está vacío.

Extra: genera un archivo de prueba con 1 000 000 de números y mide cuánto tarda en leerse.

3
Funciones, cabeceras y Makefile

Funciones

```cpp
#include <iostream>

double potencia(double base, int exp) {      // por valor
    double r = 1.0;
    for (int i = 0; i < exp; ++i) r *= base;
    return r;
}

void escalar(double& x, double k) { x *= k; }          // por referencia
double suma(const double* v, int n);                   // prototipo (declaracion)

int main() {
    double y = 2.0;
    escalar(y, 3.0);                                   // y vale 6
    std::cout << potencia(y, 2) << "\n";               // 36
}
```

➔ **Por valor**: copia. **Por referencia** (&) o **puntero** (*): modifica el original.

➔ `const` protege datos de entrada: úsalo siempre que no vayas a modificar.

Cabeceras y archivos separados

mate.hpp

```cpp
#pragma once

double cuadrado(double x)
    ;
double cubo(double x);
```

mate.cpp

```cpp
#include "mate.hpp"

double cuadrado(double x)
    {
  return x * x;
}
double cubo(double x) {
  return x * x * x;
}
```

main_mate.cpp

```cpp
#include <iostream>
#include "mate.hpp"

int main() {
  std::cout << cuadrado
       (3.0)
            << " "
            << cubo(3.0)
            << "\n";
}
```

```
g++ -c mate.cpp
g++ -c main_mate.cpp

g++ mate.o main_mate.o -o app
```

★ Regla de oro

Declara en el `.hpp`, **define** en el `.cpp`. `#pragma once` evita incluir una cabecera dos veces. `"mi.hpp"` busca en tu proyecto; `<iostream>` en el sistema.

Make: compilar solo lo que cambió

```
CXX      = g++
CXXFLAGS = -std=c++23 -O2 -Wall -Wextra
OBJS     = main_mate.o mate.o
TARGET   = app

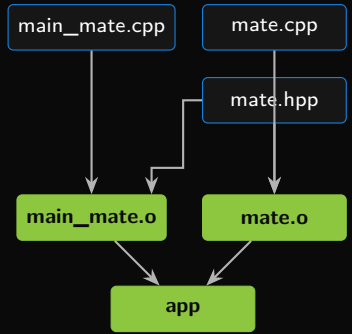
$(TARGET): $(OBJS)
^^I$(CXX) $(OBJS) -o $@

%.o: %.cpp mate.hpp
^^I$(CXX) $(CXXFLAGS) -c $< -o $@

clean:
^^Irm -f $(OBJS) $(TARGET)

.PHONY: clean
```

La sangría de las recetas es un **TAB**, no espacios.



Si solo editas `mate.cpp`, `make` recompila **solo** `mate.o` y vuelve a enlazar.

Reto individual 3: tu propia biblioteca

■ Reto 3 – Biblioteca de geometría

1. Crea `geo.hpp` y `geo.cpp` con funciones `area_circulo`, `area_triangulo` y `perimetro_rectangulo`.
2. Escribe `main.cpp` que lea los parámetros por `argv` y llame a las funciones.
3. Crea un `Makefile` con las reglas `all`, `clean` y una regla de patrón `%.o: %.cpp`.
4. Edita solo `main.cpp` y ejecuta `make`: comprueba que **no** se recompila `geo.cpp`.

4

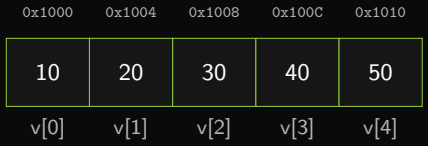
Memoria: arreglos, punteros, stack y heap

Arreglos: bloques contiguos

```cpp
int v[5] = {10, 20, 30, 40, 50};

for (int i = 0; i < 5; ++i)
    std::cout << v[i] << " ";

// matriz 2x3 (en memoria: 6 enteros)
int m[2][3] = {{1,2,3},{4,5,6}};
```

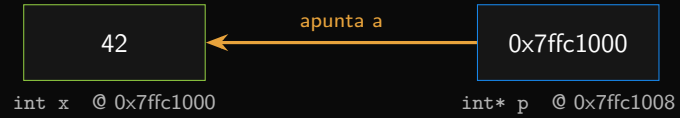


Cada `int` ocupa 4 bytes: direcciones consecutivas. Por eso recorrer un arreglo **en orden** es rápido (caché).

❄ Sin comprobación

`v[5]` compila pero es **comportamiento indefinido**: C/C++ no revisa límites.

Punteros: una variable que guarda una dirección




```cpp
int x = 42;
int* p = &x;          // &x : direccion de x
std::cout << *p;      // *p : valor al que apunta  -> 42
*p = 7;               // modifica x a traves del puntero
int* q = nullptr;     // puntero "a nada": nunca desreferenciar
```

Punteros en funciones: swap

```
void swap_ptr(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}

void swap_ref(int& a, int& b) {
    int t = a;
    a = b;
    b = t;
}

int x = 1, y = 2;
swap_ptr(&x, &y);    // x=2, y=1
swap_ref(x, y);      // x=1, y=2
```

- En C solo existen punteros; en C++ puedes usar **referencias**, más seguras (nunca nulas).
- Un puntero a `const` (`const int*`) permite leer pero no modificar.
- Aritmética: `p + 1` avanza `sizeof(*p)` bytes.

★ Importa en HPC

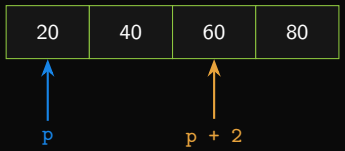
MPI y CUDA reciben **punteros** a buffers (`MPI_Send(buf,…)`, `cudaMemcpy(dst,src,…)`). Dominar punteros es requisito.

Punteros y arreglos

```cpp
int v[4] = {10, 20, 30, 40};
int* p = v;                        // el nombre del arreglo decae a &v[0]

std::cout << *p        << "\n"; // 10
std::cout << *(p + 2) << "\n"; // 30   (igual que v[2])
std::cout << p[3]      << "\n"; // 40

for (int* it = v; it != v + 4; ++it)
    *it *= 2;                      // v = {20, 40, 60, 80}
```



Stack y heap



Heap

- Tamaño elegido en ejecución
- Tú reservas y liberas (`new`/`delete`)
- Más lento, mucha capacidad

Stack

- Automático: se libera al salir de la función
- Muy rápido, tamaño limitado (~8 MB)
- `double a[10000000]` local ⇒ *stack overflow*

Memoria dinámica: new y delete

```cpp
#include <iostream>

int main() {
    int n = 1000;
    double* datos = new double[n];        // reserva en el heap

    for (int i = 0; i < n; ++i) datos[i] = i * 0.5;
    std::cout << datos[n - 1] << "\n";    // 499.5

    delete[] datos;                        // libera: new[] -> delete[]
    datos = nullptr;
}
```

- ➔ `new`↔`delete`, `new[]`↔`delete[]`. En C: `malloc`↔`free`.
- ➔ Cada reserva sin su liberación = **fuga de memoria** (*memory leak*).
- ➔ **C++ moderno**: prefiere `std::vector` o `std::unique_ptr`; casi nunca `new` a mano.

Detectar fugas: ASan y valgrind

```cpp
// fuga.cpp
int main() {
    double* p = new double[1000];   // 8000 bytes
    p[0] = 1.0;
    return 0;                       // sin delete[]
}
```

```
g++ -g -fsanitize=address fuga.cpp -o fuga && ./fuga
#  ==ERROR: LeakSanitizer: detected memory leaks
#  Direct leak of 8000 byte(s) in 1 object(s)

valgrind --leak-check=full ./fuga
#  definitely lost: 8,000 bytes in 1 blocks
```

★ Hábito

Durante el desarrollo compila con `-g -fsanitize=address,undefined`. Para producción quita los *sanitizers* y usa `-O2`.

Reto individual 4: memoria dinámica

■ Reto 4 – Vector a mano

1. Reserva con `new[]` un arreglo de `n double` (`n` por `argv`).
2. Escribe funciones `llenar`, `suma`, `maximo` que reciban `(const double*, int)`.
3. Introduce **a propósito** una fuga y un acceso fuera de rango; detéctalos con `-fsanitize=address` y con `valgrind`.
4. Corrige y comprueba que ambos reportes quedan limpios.
5. *Extra:* reescribe con `std::vector<double>` y compara el código.

5
La biblioteca estándar (STL)

Contenedores: vector, array y map

```cpp
#include <array>
#include <map>
#include <string>
#include <vector>

std::vector<int> v = {5, 2, 9, 1, 7};       // tamano dinamico, memoria contigua
v.push_back(3);                              // crece solo
std::cout << v.size() << " " << v[0] << "\n";

std::array<double, 3> a = {1.0, 2.0, 3.0}; // tamano fijo, en el stack

std::map<std::string, int> edad = {{"ana", 30}, {"luis", 25}};
for (const auto& [nombre, e] : edad)         // structured bindings (C++17)
    std::cout << nombre << ": " << e << "\n";
```

➔ `std::vector` libera su memoria solo (RAII) y es **contiguo**: su `data()` se pasa a MPI/CUDA.

Algoritmos y lambdas

```cpp
#include <algorithm>
#include <numeric>
#include <vector>

std::vector<int> v = {5, 2, 9, 1, 7};

std::sort(v.begin(), v.end());                          // 1 2 5 7 9
int suma = std::accumulate(v.begin(), v.end(), 0);      // 24
auto it  = std::find(v.begin(), v.end(), 7);            // iterador

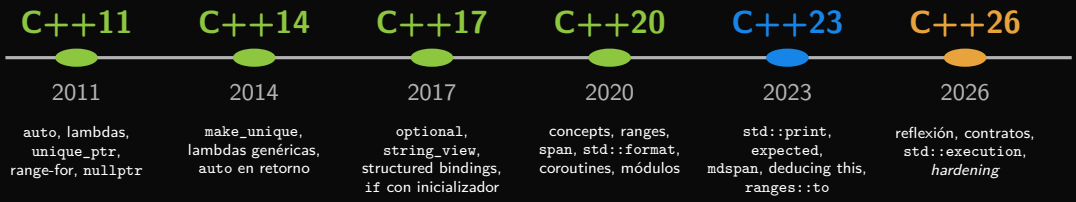
// lambda: funcion anonima
std::sort(v.begin(), v.end(), [](int a, int b) { return a > b; });  //
    descendente

int k = 3;
int cuantos = std::count_if(v.begin(), v.end(),
                            [k](int x) { return x > k; });          // captura k
```

Idea clave: los algoritmos trabajan sobre **rangos de iteradores**; el mismo `sort` sirve para cualquier contenedor.

6
C++ moderno: C++11 a C++23

Línea de tiempo del estándar



C++23 es el último estándar **publicado**; C++26 está técnicamente completo (marzo 2026) y en proceso de publicación. Fuente: H. Sutter, *Trip report: March 2026 ISO C++ standards meeting*; cppreference.com.

C++11 / C++14: el cambio de era

```cpp
#include <iostream>
#include <memory>
#include <vector>

int main() {
    auto x = 3.5;                                   // tipo deducido (C++11)
    std::vector<int> v = {1, 2, 3, 4};
    for (auto e : v) std::cout << e << " ";          // range-for (C++11)

    auto cuadrado = [](auto a) { return a * a; };    // lambda generica (C++14)
    std::cout << cuadrado(4) << " " << cuadrado(1.5) << "\n";

    auto p = std::make_unique<int[]>(5);             // make_unique (C++14)
    p[0] = 42;                                       // se libera solo al salir
}
```

RAII: `unique_ptr` (C++11) libera el recurso en su destructor: adiós `delete`. Compila con `-std=c++14` o superior.

C++17: expresividad

```cpp
#include <iostream>
#include <map>
#include <optional>
#include <string>

using Mapa = std::map<std::string, int>;

std::optional<int> buscar(const Mapa& m, const std::string& k) {
    if (auto it = m.find(k); it != m.end())   // if con inicializador
        return it->second;
    return std::nullopt;                      // "sin valor"
}

int main() {
    Mapa m = {{"a", 1}, {"b", 2}};
    if (auto r = buscar(m, "b")) std::cout << *r << "\n";
    for (const auto& [clave, valor] : m)      // structured bindings
        std::cout << clave << "=" << valor << " ";
}
```

C++20: concepts, ranges, span, format

```cpp
#include <concepts>
#include <format>
#include <iostream>
#include <ranges>
#include <vector>

template <std::integral T>                    // concept: solo enteros
T doble(T x) { return 2 * x; }

int main() {
    std::vector<int> v = {1, 2, 3, 4, 5, 6};
    auto pares = v | std::views::filter([](int x) { return x % 2 == 0; })
                   | std::views::transform([](int x) { return x * x; });
    for (int x : pares) std::cout << x << " ";            // 4 16 36

    std::cout << std::format("\ndoble(21) = {}\n", doble(21));
}
```

También: `std::span<T>` (vista sin copia de un arreglo), *coroutines* y módulos.

C++23: print, expected y ranges::to

```cpp
#include <expected>
#include <print>
#include <ranges>
#include <string>
#include <vector>
std::expected<double, std::string> dividir(double a, double b) {
    if (b == 0.0) return std::unexpected("division por cero");
    return a / b;
}

int main() {
    std::println("Hola {} desde C++{}", "HPC", 23);      // reemplaza cout/printf
    if (auto r = dividir(1, 4)) std::println("1/4 = {}", *r);
    auto e = dividir(1, 0);
    if (!e) std::println("error: {}", e.error());         // sin excepciones
    auto cuadrados = std::views::iota(1, 6)
                   | std::views::transform([](int x) { return x * x; })
                   | std::ranges::to<std::vector>();      // a vector
    for (int c : cuadrados) std::print("{} ", c);          // 1 4 9 16 25
}
```

C++23: mdspan y deducing this

std::mdspan

```cpp
#include <mdspan>
#include <vector>
#include <print>

int main() {
  std::vector<double> d = {1,2,3,
                           4,5,6};
  std::mdspan A(d.data(), 2, 3);   // 2x3
  for (std::size_t i = 0; i < A.extent(0)
      ; ++i)
    for (std::size_t j = 0; j < A.extent
        (1); ++j)
      std::print("{} ", A[i, j]);
}
```

Necesita `module load gnu15`.

Deducing this

```cpp
struct Vec {
  std::vector<double> d;

  auto&& operator[](this auto&& self
      ,
                    std::size_t i) {
    return self.d[i];
  }
};
```

Un método sirve para `const` y no `const`.

Para HPC: `mdspan` separa el **dato** de su **disposición** (filas/columnas); es la base de matrices en bibliotecas como *Kokkos*.

¿Qué viene? C++26

- ➔ **Reflexión estática**: el programa inspecciona sus tipos en compilación.
- ➔ **Contratos**: precondiciones y postcondiciones (`pre`, `post`, `contract_assert`).
- ➔ ***Hardening***: acceso fuera de rango detectado en `vector`, `span`.
- ➔ **`std::execution`**: modelo unificado de tareas asíncronas (CPU y GPU).
- ➔ *Pack indexing*: `Ts...[0]`.

★ Soporte de compiladores

`-std=c++26`: GCC 14 (parcial). *Pack indexing*: GCC 15. Reflexión (`-freflection`) y contratos: GCC 16. En el cluster hoy: `gnu14`, `gnu15`.

❄ Prudencia

En producción apunta a **C++20/23** y verifica el soporte de tu compilador.

Fuentes: H. Sutter, *Trip report: March 2026 ISO C++ standards meeting*; gcc.gnu.org/projects/cxx-status.html.

Reto individual 5: moderniza tu código

■ Reto 5 – De C a C++ moderno

Parte del programa `stats.cpp` del Reto 2 (o de uno en C con `malloc` y `printf`):

1. Cambia los arreglos manuales por `std::vector<double>` y `printf` por `std::println`.
2. Usa `std::accumulate` y `std::ranges::minmax` en lugar de bucles a mano.
3. Devuelve un `std::expected<Resultado, std::string>` en vez de terminar con `exit(1)`.
4. Verifica que compila sin advertencias con `-std=c++23 -Wall -Wextra`.

Pregunta: ¿cuántas líneas ganaste o perdiste? ¿Qué es más fácil de leer?

7

Pensar en HPC: localidad y optimización

La jerarquía de memoria



	Tamaño típico	Latencia
Registros	$\sim$ KB	< 1 ns
Caché L1	32–64 KB	$\sim$1 ns
Caché L2	0.25–2 MB	$\sim$4 ns
Caché L3	decenas de MB	$\sim$15 ns
Memoria RAM	GB–TB	$\sim$80 ns

Órdenes de magnitud típicos; varían según la CPU.

Un acceso a RAM puede costar **decenas de veces** lo de uno a L1. La caché trae **líneas de 64 bytes**: acceder a datos contiguos aprovecha esa línea.

Ejemplo: multiplicar matrices $C = A \cdot B$

```cpp
using Matriz = std::vector<double>;        // n*n almacenada por filas

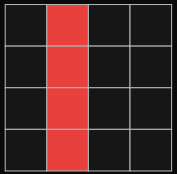
void matmul_ijk(const Matriz& A, const Matriz& B, Matriz& C, int n) {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j) {
            double s = 0.0;
            for (int k = 0; k < n; ++k)
                s += A[i*n + k] * B[k*n + j];  // B: salta n (malo)
            C[i*n + j] = s;
        }
}

void matmul_ikj(const Matriz& A, const Matriz& B, Matriz& C, int n) {
    for (int i = 0; i < n; ++i)
        for (int k = 0; k < n; ++k) {
            double a = A[i*n + k];
            for (int j = 0; j < n; ++j)
                C[i*n + j] += a * B[k*n + j];  // consecutivo (bueno)
        }
}
```

Mismas $2n^3$ operaciones; cambia solo el **orden de los bucles**.

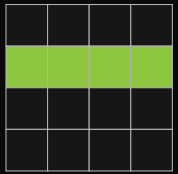
¿Por qué cambia el rendimiento?

ijk: B recorrida por columnas



cada acceso cae en
una línea de caché distinta

ikj: B recorrida por filas



los accesos son contiguos:
se aprovecha la línea de caché

C/C++ guardan las matrices **por filas** (*row-major*); Fortran, por columnas.

Resultados medidos

Compilación (n = 512)	ijk (GFLOP/s)	ikj (GFLOP/s)
-O0	0.27	0.47
-O2	0.87	3.65
-O3	0.89	6.45

- ➔ Con `-O3`, **ikj** es unas 7 veces más rápido que **ijk** sin cambiar el algoritmo.
- ➔ Medido con `g++ 14`, `std::chrono::steady_clock`, en una máquina virtual pequeña; **tus números en `guane` serán distintos**: mídelos.

Banderas útiles: `-O2`/`-O3`; `-march=native` (instrucciones vectoriales del nodo donde compilas: ejecuta en el **mismo tipo de nodo**, o puede dar *illegal instruction*); `-ffast-math` (reordena operaciones: con cuidado).

Medir tiempo en C++

```cpp
#include <chrono>

template <class F>
double medir(F&& f) {                          // devuelve segundos
    auto t0 = std::chrono::steady_clock::now();
    f();
    return std::chrono::duration<double>(
               std::chrono::steady_clock::now() - t0).count();
}

// uso
double t = medir([&] { matmul_ikj(A, B, C, n); });
double gflops = 2.0 * n * n * n / 1e9 / t;
std::cout << "tiempo " << t << " s  ->  " << gflops << " GFLOP/s\n";
```

- ➔ Usa `steady_clock`; repite y reporta mínimo o mediana. **GFLOP/s** es la métrica estándar.
- ➔ Mide siempre con `-O2` o `-O3`: medir sin optimizar no refleja el rendimiento real.

Reto individual 6: orden de bucles

■ Reto 6 – Caza el mejor orden

1. Implementa las 6 permutaciones de bucles (ijk, ikj, jik, jki, kij, kji) de la multiplicación de matrices.
2. Mide GFLOP/s de cada una para $n = 256, 512, 1024$ con `-O2`.
3. Presenta una tabla y explica con un dibujo cuáles recorren memoria contigua.
4. Prueba `-O3 -march=native` (compila y ejecuta en el mismo tipo de nodo) y reporta el cambio.
5. Ejecuta en un nodo de cómputo: `srun -p legacy -n 1 ./matmul 1024`.

8
Reto grupal

Reto grupal: mini biblioteca de álgebra lineal

★ Reto grupal – Equipos de 3 o 4 personas

Construyan la biblioteca `minila` en C++23:

1. Clase `Matriz` (RAII con `std::vector<double>`), acceso `A(i,j)` y dimensiones.
2. Suma, producto por escalar, transpuesta y **producto de matrices** (versión `ikj`).
3. `Makefile` con `all`, `test`, `bench`, `clean`; código en `.hpp`/`.cpp`.
4. Pruebas con `assert` ($A \cdot I = A$, $(A^T)^T = A$).
5. `bench`: GFLOP/s para varios n, ejecutado con `srun -p legacy`.
6. Limpio bajo `-fsanitize=address,undefined`.

Entregables y criterios

Entregables

- Repositorio con el código y el `Makefile`.
- `README` con instrucciones de compilación (`module load gnu14`).
- Tabla de GFLOP/s por tamaño.
- Exposición corta por equipo.

Criterios

- Correctitud (las pruebas pasan).
- Memoria limpia (ASan/valgrind).
- Claridad y uso de C++ moderno.
- Análisis del rendimiento con dibujos o gráficas.

★ Se reutiliza

Esta misma `minila` será la base de los retos de los módulos de **OpenMP** y **MPI**: la paralelizaremos.

Referencias

- Imperial College London RCDS, *Introduction to C/C++* (material base del semillero).
- cppreference.com: referencia de C y C++ y tabla de soporte de compiladores.
- GCC, *C++ Standards Support in GCC*: gcc.gnu.org/projects/cxx-status.html.
- H. Sutter, *Trip report: March 2026 ISO C++ standards meeting*: herbsutter.com.
- B. Stroustrup, *A Tour of C++*, 3.ª ed., Addison-Wesley, 2022.
- U. Drepper, *What Every Programmer Should Know About Memory*, 2007.
- C++ Core Guidelines: isocpp.github.io/CppCoreGuidelines.

Cierre

★ Lo esencial de C/C++ para HPC

Compila con *warnings* y *sanitizers*, **mide antes de optimizar** y cuida la localidad de memoria: es la base de todo lo que se paraleliza después.

¡Gracias!

sc3.operaciones@uis.edu.co