Programación Paralela

Apéndice: C++

Segundo Semestre, 2025

---

1 Introducción al C y C++

Aunque muchos de los ejemplos de CUDA estarán en C, podremos utilizar C++ para simplificar los ejemplos del host. ya que C++ es un lenguaje que tiene una biblioteca estándar muy amplia y que es muy eficiente en términos de tiempo de ejecución y para algunas cosas es más sencillo.

Se asume que tenés algún conocimiento básico de programación en lenguaje C (u otro lenguaje de alto nivel), con lo cual no se explicarán conceptos como `if` o `int a = 0`, pero sí se explicarán conceptos que en C no se ven como `vector` o `sort`.

```cpp
// Este header puede no estar disponible en tu sistema
// en ese caso no lo incluyas, e inclui cada header a mano (como siempre)
#include <bits/stdc++.h>

using namespace std;

int main() {
  // solucion

  return 0;
}
```

El código que se ejecuta comienza en la sección main, aunque se pueden definir funciones adicionales si es necesario. Para compilarlo tendremos que utilizar el compilador de NVIDIA *nvcc* con los siguientes parámetros:

```
nvcc codigo.cc -o binario
```

Donde `codigo.cc` es el nombre del archivo que contiene el código fuente y `binario` es el nombre del ejecutable que se generará. Si querés utilizar C, sólo hay que cambiar la extensión del código fuente a `codigo.c`.

1.1 Entrada y salida

Si bien en C se aprendieron a utilizar las funciones `scanf` y `printf`, en C++ se utilizan los objetos `cin` y `cout` para leer y escribir datos al flujo de entrada y salida (STDIN y STDOUT, respectivamente), que son más fáciles de utilizar, ya que ignora los espacios y puede utilizarse como un *parser* ejemplo:

1.1.1 cin

```cpp
string pre, post;
int n;

cin >> pre >> n >> post;
```

Este código por ejemplo podría leer una patente argentina de las nuevas, independientemente de la cantidad de espacios que haya entre las letras y los números (incluso puede funcionar si se ingresan *new lines*).

```
  AA 234 BB
```

```
  AA
345
CD
```

1.1.2 cout

Lo mismo se puede hacer con la salida, utilizando `cout`:

```
string pre = "AA", post = "BB";
int n = 234;
cout << pre << " " << n << " " << post << endl;
```

Que imprimiría la patente `AA 234 BB`.

1.2 while(cin)

Para leer una cantidad indefinida de datos, se puede utilizar el bucle `while(cin)`, de la siguiente forma:

```
int n;
while(cin >> n) {
  cout << n << endl;
}
```

Este código leerá números enteros hasta que se llege al `EOF` (End Of File), que en la terminal.

1.3 for

El lenguaje C++ ha mejorado la forma de hacer bucles `for`, permitiendo hacer bucles más sencillos utilizando la palabra reservada `auto` para inferir el tipo de dato de la variable que se está iterando. Por ejemplo:

```
  vector<int> v = {1, 2, 3, 4, 5};
  for(auto i : v) {
    cout << i << endl;
  }
```

Que facilita muchísimo la escritura del código.

1.4 getline

Puede ser necesario tener que leer una línea completa y no ignorar los espacios como cuando utilizamos `cin`, en este caso se puede utilizar `getline` de la siguiente manera:

```
string s;
getline(cin, s);
```

2 int, long long, string

En C++ hay varios tipos de datos básicos (o no tanto) que hay que saber utilizar. Los más comunes son `int`, `long long` y `string`. El primero es un entero de 32 bits, el segundo es un entero de 64 bits y el tercero es una cadena de caracteres (ambos pueden ser signed or `unsigned`).

Si bien se suelen utilizar `int` en la mayoría de los problemas, hay que tener en cuenta que puede haber problemas en los cuales elegir signo o tamaño incorrecto puede llevar a errores en la solución. Por ejmeplo, un error común suele ser hacer esto:

```
1 int a = 123456789;
2 long long b = a * a; // b = -1757895751
```

Esto se da porque `a * a` se calcula como un `int` y luego se convierte a `long long`, lo que hace que por un momento `a * a` sea un *overflow* de `int`, y por ende el resultado sea un número negativo.

3 Vectores (arrays dinámicos)

Los vectores en C++ son una estructura de datos similar a los arrays de C pero con algunas diferencias. La principal diferencia es que los vectores pueden cambiar de tamaño dinámicamente, es decir, que se pueden agregar o quitar elementos en tiempo de ejecución. Es importante señalar que los vectores en C++ comienzan en la posición 0, al igual que los arrays de C, y que se accede a los elementos de la misma forma que en C. Por otra parte la complejidad de las operaciones de acceso son $O(1)$ al igual que en C y agregar o quitar elementos al final del vector es $O(1)$ en promedio (ya que en el peor caso puede llegar a ser $O(n)$). Sin embargo agregar o quitar elementos en cualquier otra posición es $O(n)$.

La forma de declarar un vector en C++ es la siguiente:

```
1 vector<T> v;
```

Donde `T` es el tipo de dato que se almacenará en el vector. Por ejemplo, si se quiere un vector de enteros, se puede hacer lo siguiente:

```
1 vector<int> v;
```

Mientras que se puede inicializar un vector con un tamaño específico de la siguiente forma:

```
1 vector<int> v(10);
```

En este caso el vector `v` tendrá un tamaño de 10 elementos (normalmente inicializados en 0). También se puede inicializar un vector con un tamaño específico y un valor específico de la siguiente forma:

```
1 vector<int> v(10, 5);
```

En este caso el vector `v` tendrá un tamaño de 10 elementos, todos inicializados en 5. Para acceder a los elementos de un vector se puede hacer de la misma forma que en un array de C, por ejemplo:

```
1 vector<int> v(10, 5);
2 v[0] += 1;
3 cout << v[0] << endl; // Imprime 6
```

Para agregar un elemento al final de un vector se puede hacer de la siguiente forma:

```
vector<int> v;
v.push_back(5);
```

En este caso el vector `v` tendrá un tamaño de 1 elemento, y el elemento en la posición 0 será 5. Para quitar un elemento al final de un vector se puede hacer de la siguiente forma:

```
vector<int> v;
v.push_back(5);
v.pop_back();
```

En este caso el vector `v` tendrá un tamaño de 0 elementos. Para recorrer un vector se puede hacer de la siguiente forma:

```
vector<int> v(10, 5);
for(int i = 0; i < v.size(); i++) {
  cout << v[i] << endl;
}
```

En este caso se recorre el vector `v` e imprime cada uno de los elementos.

3.1 Iteradores

Para recorrer un vector de forma más sencilla se puede utilizar un iterador, los iteradores son similares a los punteros en C y tienen la ventaja de que el iterador se puede mover tanto hacia adelante como hacia atrás con el operador `++` y `--`, respectivamente. Por ejemplo:

```
vector<int> v(10, 5);
for(auto it = v.begin(); it != v.end(); it++) {
  cout << *it << endl;
}
```

3.2 algorithms

3.2.1 sort

Además de los iteradores, los vectores tienen una serie de funciones que permiten realizar operaciones sobre ellos. Por ejemplo, se puede ordenar un vector de la siguiente forma:

```
#include <algorithm>

vector<int> v = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};
sort(v.begin(), v.end());
```

En este caso el vector `v` quedará ordenado de menor a mayor entre los índices `v.begin()` y `v.end()`, ya que `v.begin()` es un iterador que apunta al primer elemento del vector y `v.end()` es un iterador que apunta al final del vector (NO apunta al último elemento del vector, ya que `v.begin()` sería la posición 0 y `v.end()` sería la posición n).

Cabe señalar que la función `sort` es una función de la librería `algorithm` y que se puede utilizar para ordenar cualquier tipo de contenedor que tenga iteradores, no sólo vectores, y además ordena entre cualquier rango de iteradores, no sólo entre `begin()` y `end()`. Por ejemplo podríamos ordenar los elementos entre los índices 2 y 5 de un vector de la siguiente forma:

```
sort(v.begin() + 2, v.begin() + 5);
```

Nótese que `v.begin() + 2` es un iterador que apunta al tercer elemento del vector, ya que el operador `+` retorna otro iterador que apunta 2 posiciones más adelante que el iterador original. Lo mismo sucede con `-` que retorna un iterador que apunta a una posición anterior, por ejemplo `v.end() - 1` es un iterador que apunta al último elemento del vector, ya que `v.end()` apunta a una posición después del último elemento del vector.

3.2.2 reverse sort

En C++ ordenar un vector de mayor a menor es un poco más complicado que ordenarlo de menor a mayor, ya que la función `sort` no tiene un parámetro que permita ordenar de mayor a menor, para ello se utilizan iteradores reversos que se mueven de atrás hacia adelante. Por ejemplo:

```
1 sort(v.rbegin(), v.rend());
```

En este caso el vector `v` quedará ordenado de mayor a menor entre los índices `v.rbegin()` y `v.rend()`.