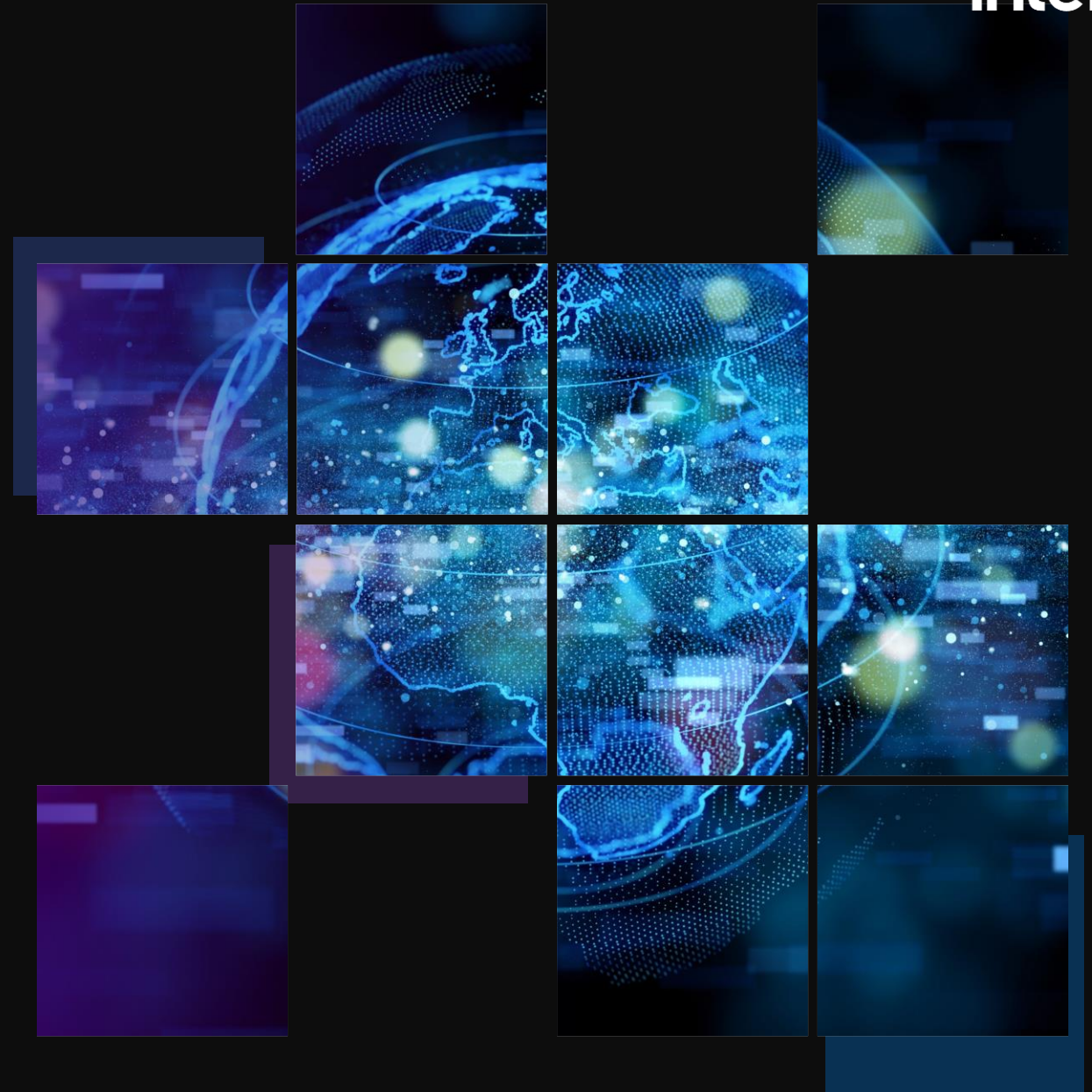


Sesión 2: DPC++ avanzado

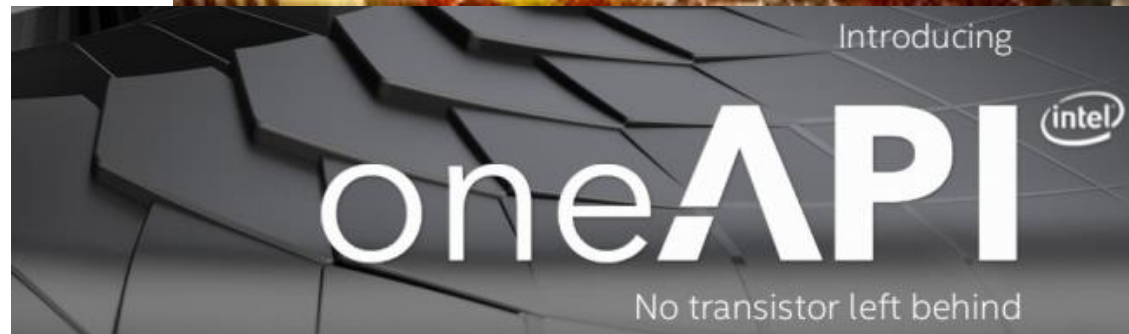
Carlos García Sánchez

-
Conoce de la mano de expertos oneAPI
La solución para programadores de Intel



Agenda

- DPC++ básico
- Selección Device
- Manejo datos: Buffers y USM
- Ejec. y kernel schedule
- Sub-Groups



OBJETIVO

Introducir Data Parallel C++, la estructura de código y conceptos clave para conseguir que escribir código rápidamente!!

RETOS PROGRAMMING EN ARQUITECTURAS PARALELAS

Crecimiento de cargas de trabajo especializadas

Variedad de hardware

No hay lenguaje de programación común o API

No existen herramientas en todas las plataformas

Cada plataforma requiere adaptar el software

Application Workloads Need Diverse Hardware



SCALAR



VECTOR



MATRIX

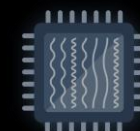


SPATIAL

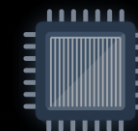
Middleware / Frameworks

Language & Libraries

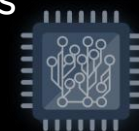
XPU^s



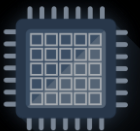
CPU



GPU



FPGA



OTHER ACCEL.

ONEAPI INICIATIVA INDUSTRIAL

ALTERNATIVA A SOLUCIÓN DE ÚNICO PROVEEDOR

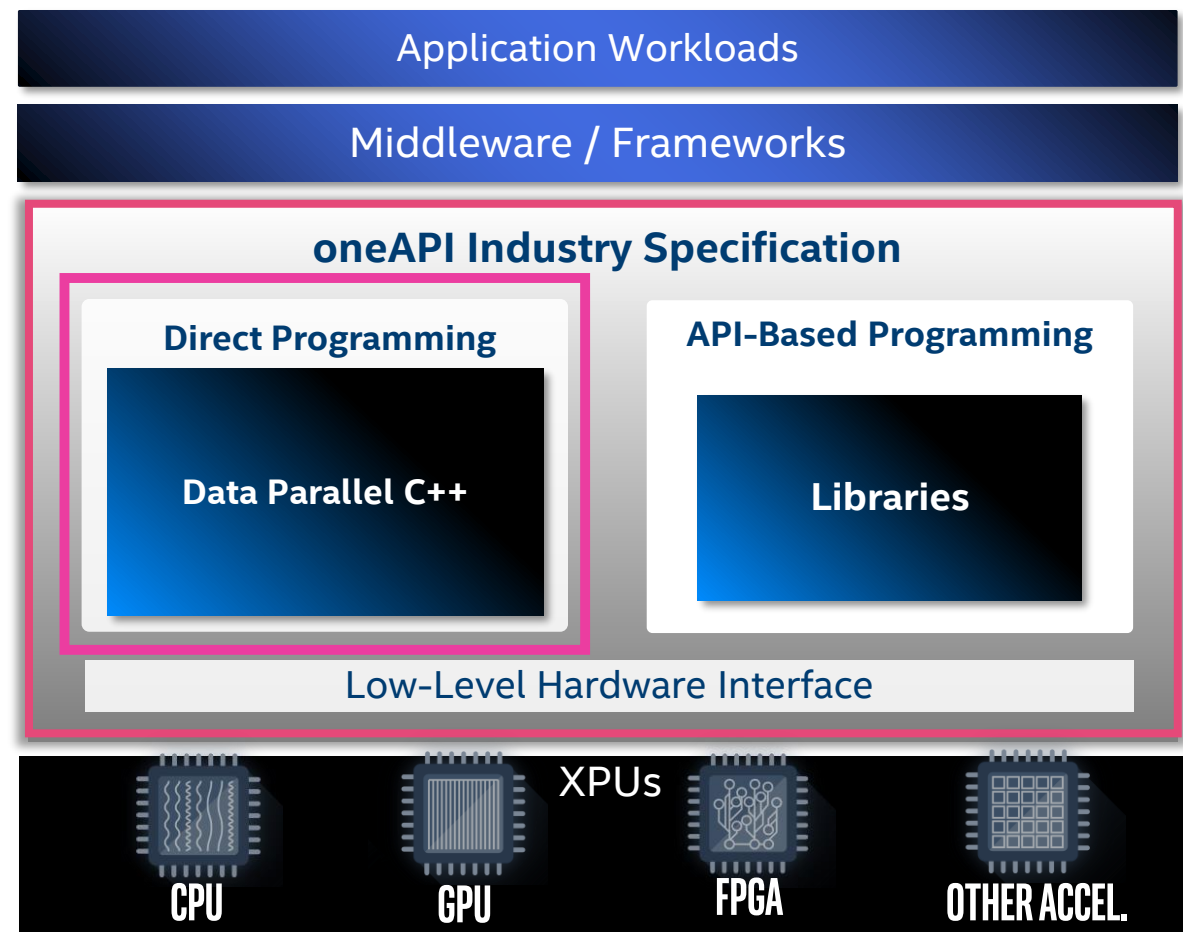
Un lenguaje basado en estándares, DPC++, basado en C++ y SYCL

Potentes API diseñadas para acelerar funciones de dominio específico

Interfaz hardware a bajo nivel para proporcionar una capa de abstracción favorable a su adopción por los fabricantes

Estándar abierto para promover el apoyo de la comunidad y la industria

Permite la reutilización de código en diferentes arquitecturas y proveedores



Visit oneapi.com for more details



■ DPC++ básico

DPC++ BÁSICO

```
#include <CL/sycl.hpp>
```

```
using namespace sycl;
```

```
#define dpc_r access::mode::read
```

```
#define dpc_w access::mode::write
```

```
#define dpc_rw access::mode::read_write
```

DPC++ BÁSICO

```
int main() {
    float A[1024], B[1024], C[1024];
    {
        buffer<float, 1> bufA { A, range<1> {1024} };
        buffer<float, 1> bufB { B, range<1> {1024} };
        buffer<float, 1> bufC { C, range<1> {1024} };

        queue q;
        q.submit([&](handler& h) {
            auto A = bufA.get_access<dpc_r>(h);
            auto B = bufB.get_access<dpc_r>(h);
            auto C = bufC.get_access<dpc_w>(h);

            h.parallel_for(range<1> {1024}, [=](id<1> i) {
                C[i] = A[i] + B[i];
            });
        });
    }
    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

1. Creación buffer y
usando punteros del host

DPC++ BÁSICO

```
int main() {
    float A[1024], B[1024], C[1024];
    {
        buffer<float, 1> bufA { A, range<1> {1024} };
        buffer<float, 1> bufB { B, range<1> {1024} };
        buffer<float, 1> bufC { C, range<1> {1024} };

        queue q;
        q.submit([&](handler& h) {
            auto A = bufA.get_access<dpc_r>(h);
            auto B = bufB.get_access<dpc_r>(h);
            auto C = bufC.get_access<dpc_w>(h);

            h.parallel_for(range<1> {1024}, [=](id<1> i) {
                C[i] = A[i] + B[i];
            });
        });
    }
    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

DPC++ BÁSICO

```
int main() {
    float A[1024], B[1024], C[1024];
    {
        buffer<float, 1> bufA { A, range<1> {1024} };
        buffer<float, 1> bufB { B, range<1> {1024} };
        buffer<float, 1> bufC { C, range<1> {1024} };

        queue q;
        q.submit([&](handler& h) {
            auto A = bufA.get_access<dpc_r>(h);
            auto B = bufB.get_access<dpc_r>(h);
            auto C = bufC.get_access<dpc_w>(h);

            h.parallel_for(range<1> {1024}, [=](id<1> i) {
                C[i] = A[i] + B[i];
            });
        });
    }
    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

2. Creación cola y
emission del trabajo a
la cola del dispositivo

DPC++ BÁSICO

```
int main() {
    float A[1024], B[1024], C[1024];
    {
        buffer<float, 1> bufA { A, range<1> {1024} };
        buffer<float, 1> bufB { B, range<1> {1024} };
        buffer<float, 1> bufC { C, range<1> {1024} };

        queue q;
        q.submit([&](handler& h) {
            auto A = bufA.get_access<dpc_r>(h);
            auto B = bufB.get_access<dpc_r>(h);
            auto C = bufC.get_access<dpc_w>(h);

            h.parallel_for(range<1> {1024}, [=](id<1> i) {
                C[i] = A[i] + B[i];
            });
        });
    }
    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

3. Creación de **accessors**
read/write: permite
identificar dependencias
de datos entre kernels
consecutivos o acceso a
los buffers desde el host

DPC++ BÁSICO

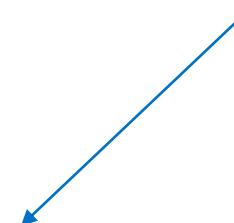
```
int main() {
    float A[1024], B[1024], C[1024];
    {
        buffer<float, 1> bufA { A, range<1> {1024} };
        buffer<float, 1> bufB { B, range<1> {1024} };
        buffer<float, 1> bufC { C, range<1> {1024} };

        queue q;
        q.submit([&](handler& h) {
            auto A = bufA.get_access<dpc_r>(h);
            auto B = bufB.get_access<dpc_r>(h);
            auto C = bufC.get_access<dpc_w>(h);

            h.parallel_for(range<1> {1024}, [=](id<1> i) {
                C[i] = A[i] + B[i];
            });
        });
    }
    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

4. Suma de vectores
mediante el kernel
con la tarea
encolada
parallel_for.

Esquema **lambda**
para expresar la
ejecución paralela
en los work-items



DPC++ BÁSICO

```
int main() {
    float A[1024], B[1024], C[1024];
    {
        buffer<float, 1> bufA { A, range<1> {1024} };
        buffer<float, 1> bufB { B, range<1> {1024} };
        buffer<float, 1> bufC { C, range<1> {1024} };

        queue q;
        q.submit([&](handler& h) {
            auto A = bufA.get_access<dpc_r>(h);
            auto B = bufB.get_access<dpc_r>(h);
            auto C = bufC.get_access<dpc_w>(h);

            h.parallel_for<class vector_add>(range<1> {1024}, [=](id<1> i) {
                C[i] = A[i] + B[i];
            });
        });
    }
    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

En SYCL 1.2.1 se requiere incluir el nombre de la función lambda

Con el flag -fsycl-unnamed-lambda se puede eliminar el requisito

DPC++ BÁSICO

```
int main() {
    float A[1024], B[1024], C[1024];
    {
        buffer<float, 1> bufA { A, range<1> {1024} };
        buffer<float, 1> bufB { B, range<1> {1024} };
        buffer<float, 1> bufC { C, range<1> {1024} };

        queue q;
        q.submit([&](handler& h) {
            auto A = bufA.get_access<dpc_r>(h);
            auto B = bufB.get_access<dpc_r>(h);
            auto C = bufC.get_access<dpc_w>(h);

            h.parallel_for(range<1> {1024}, [=](id<1> i) {
                C[i] = A[i] + B[i];
            });
        });
    }
    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

El **range** define
el espacio de
iteraciones


DPC++ BÁSICO

```
int main() {
    float A[1024], B[1024], C[1024];
    {
        buffer<float, 1> bufA { A, range<1> {1024} };
        buffer<float, 1> bufB { B, range<1> {1024} };
        buffer<float, 1> bufC { C, range<1> {1024} };

        queue q;
        q.submit([&](handler& h) {
            auto A = bufA.get_access<dpc_r>(h);
            auto B = bufB.get_access<dpc_r>(h);
            auto C = bufC.get_access<dpc_w>(h);

            h.parallel_for(range<1> {1024}, [=](id<1> i) {
                C[i] = A[i] + B[i];
            });
        });
    }
    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

Cada iteración
(work-item)
tiene
identificadores
id (i) distintos



DPC++

```
h.single_task(
    [=] () {
        // kernel function is executed EXACTLY once on a SINGLE work-item
    });

h.parallel_for(
    range<3>(1024,1024,1024), // using 3D in this example
    [=] (id<3> myID) {
        // kernel function is executed on an n-dimensional range (NDrange)
    });

h.parallel_for(
    nd_range<3>({1024,1024,1024},{16,16,16}), // using 3D in this example
    [=] (nd_item<3> myID) {
        // kernel function is executed on an n-dimensional range (NDrange)
    });

h.parallel_for_work_group(
    range<2>(1024,1024), // using 2D in this example
    [=] (group<2> grp) {
        // kernel function is executed once per work-group
    });

grp.parallel_for_work_item(
    range<1>(1024), // using 1D in this example
    [=] (h_item<1> myItem) {
        // kernel function is executed once per work-item
    });
```

Basic data parallel

Explicit ND-Range

Hierarchical parallelism



■ Selección del Device

SELECCIÓN DE UN CUSTOM DEVICE

- El código siguiente muestra la heurística para un `device_selector` específico. El dispositivo seleccionado prioriza un dispositivo GPU porque la clasificación de enteros devuelta es mayor que para la CPU u otro acelerador

```
#include <CL/sycl.hpp>
using namespace cl::sycl;

class my_device_selector : public cl::sycl::device_selector {
public:
    int operator()(const cl::sycl::device &Device) const override {
        int rating;
        if (dev.is_gpu() & (dev.get_info<info::device::name>().find("Intel") != std::string::npos))
            rating=3;
        else if (dev.is_gpu()) rating=2;
        else if (dev.is_cpu()) rating=1;
        return rating;
    }
};

int main() {
    my_device_selector selector;
    queue q(selector);
    std::cout << "Device: " << q.get_device().get_info<info::device::name>() << std::endl;
}
```

SELECCIÓN DE UN CUSTOM DEVICE

- El código siguiente muestra la heurística para un `device_selector` específico. El dispositivo seleccionado prioriza un dispositivo GPU porque la clasificación de enteros devuelta es mayor que para la CPU u otro acelerador

```
#include <CL/sycl.hpp>
using namespace cl::sycl;

class my_device_selector public cl::sycl::device_selector {
public:
    int operator()(const cl::sycl::device &Device) const override {
        int rating;
        if (dev.is_gpu() & (dev.get_info<info::device::name>().find("Intel") != std::string::npos))
            rating=3;
        else if (dev.is_gpu()) rating=2;
        else if (dev.is_cpu()) rating=1;
        return rating;
    }
};

int main() {
    my_device_selector selector;
    queue q(selector);
    std::cout << "Device: " << q.get_device().get_info<info::device::name>() << std::endl;
}
```

SELECCIÓN DE UN CUSTOM DEVICE

- El código siguiente muestra la heurística para un `device_selector` específico. El dispositivo seleccionado prioriza un dispositivo GPU porque la clasificación de enteros devuelta es mayor que para la CPU u otro acelerador

```
#include <CL/sycl.hpp>
using namespace cl::sycl;

class my_device_selector public cl::sycl::device_selector {
public:
    int operator()(const cl::sycl::device &Device) const override {
        int rating;
        if (dev.is_gpu() & (dev.get_info<info::device::name>().find("Intel") != std::string::npos))
            rating=3;
        else if (dev.is_gpu()) rating=2;
        else if (dev.is_cpu()) rating=1;
        return rating;
    }
};

int main() {
    my_device_selector selector;
    queue q(selector);
    std::cout << "Device: " << q.get_device().get_info<info::device::name>() << std::endl;
}
```

EJECUCIÓN ASÍNCRONA

- Una aplicación SYCL contiene dos partes:
 - Código Host
 - Un grafo de ejecuciones de los kernel
- La **ejecución independiente** se produce excepto en aquellos puntos de sincronización necesario
 - El código de host envía trabajo, se crea grafo de dependencias
 - El grafo de ejecución y los movimientos de datos necesarios se efectúa **de forma asíncrona desde el host**, manejado por el runtime SYCL

EJECUCIÓN ASÍNCRONA

Host

Ejecución
host

Encola
kernel en el
grafo y
continúa

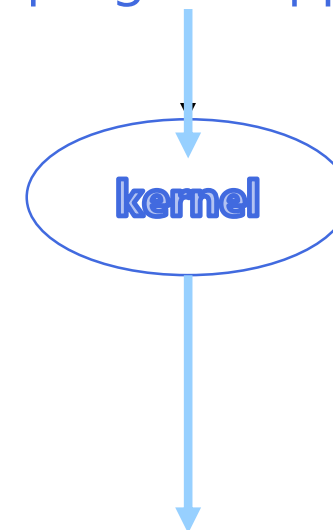
main.cpp:

```
#include <iostream>

int main() {
    const size_t array_size = 16;
    int data[array_size];
    {
        buffer<int, 1> resultBuf{ data, range<1>{array_size} };
        queue q;
        q.submit([&](handler& h) {
            auto resultAcc = resultBuf.get_access<access::mode::write>(h);
            h.parallel_for(range<1>{array_size}, [=](id<1> i) {
                resultAcc[i] = static_cast<int>(i.get(0));
            });
        });
        for( int i = 0; i < array_size; i++ ) {
            std::cout << "data[" << i << "] = " << data[i] << std::endl;
        }
        return 0;
    }
}
```

Graph

Grafo se ejecuta
de forma
asíncrona por el
programa ppal



EJECUCIÓN ASÍNCRONA

```
int main() {
    auto R = range<1>{ num };
    buffer<int> A{ R }, B{ R }; //Create Buffers A and B
    queue Q; //Create a device queue

    Q.submit([&](handler& h) { //Submit Kernel 1
        auto out = A.get_access<access::mode::write>(h); //Accessor for buffer A
        h.parallel_for(R, [=](id<1> idx) {
            out[idx] = idx[0]; }); });

    Q.submit([&](handler& h) { //This task will wait till the first queue is complete
        auto out = A.get_access<access::mode::write>(h); //Accessor for buffer A
        h.parallel_for(R, [=](id<1> idx) {
            out[idx] = idx[0]; }); });

    Q.submit([&](handler& h)
        auto out = B.get_access<access::mode::write>(h); //Accessor for buffer B
        h.parallel_for(R, [=](id<1> idx) {
            out[idx] = idx[0]; }); });

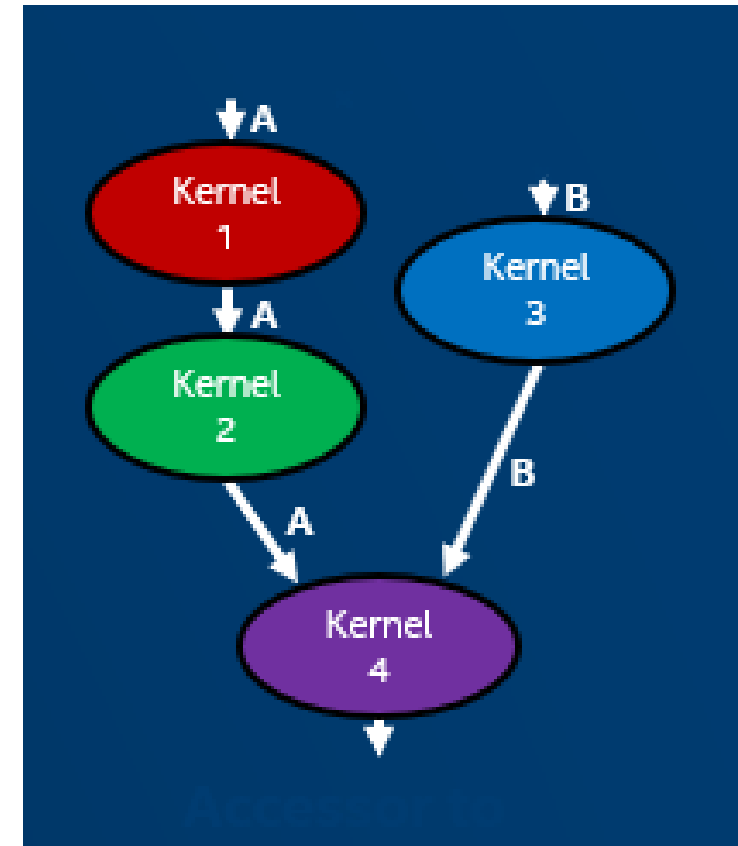
    Q.submit([&](handler& h) //This task will wait till kernel 2 and 3 are complete
        auto in = A.get_access<access::mode::read>(h);
        auto inout = B.get_access<access::mode::read_write>(h);
        h.parallel_for(R, [=](id<1> idx)
            inout[idx] *= in[idx]; });
        // And the following is back to device code
        auto result = B.get_access<access::mode::read>();
        for (int i=0; i<num; ++i) std::cout << result[i] << "\n";
        return 0;
    }
```

kernel1

kernel2

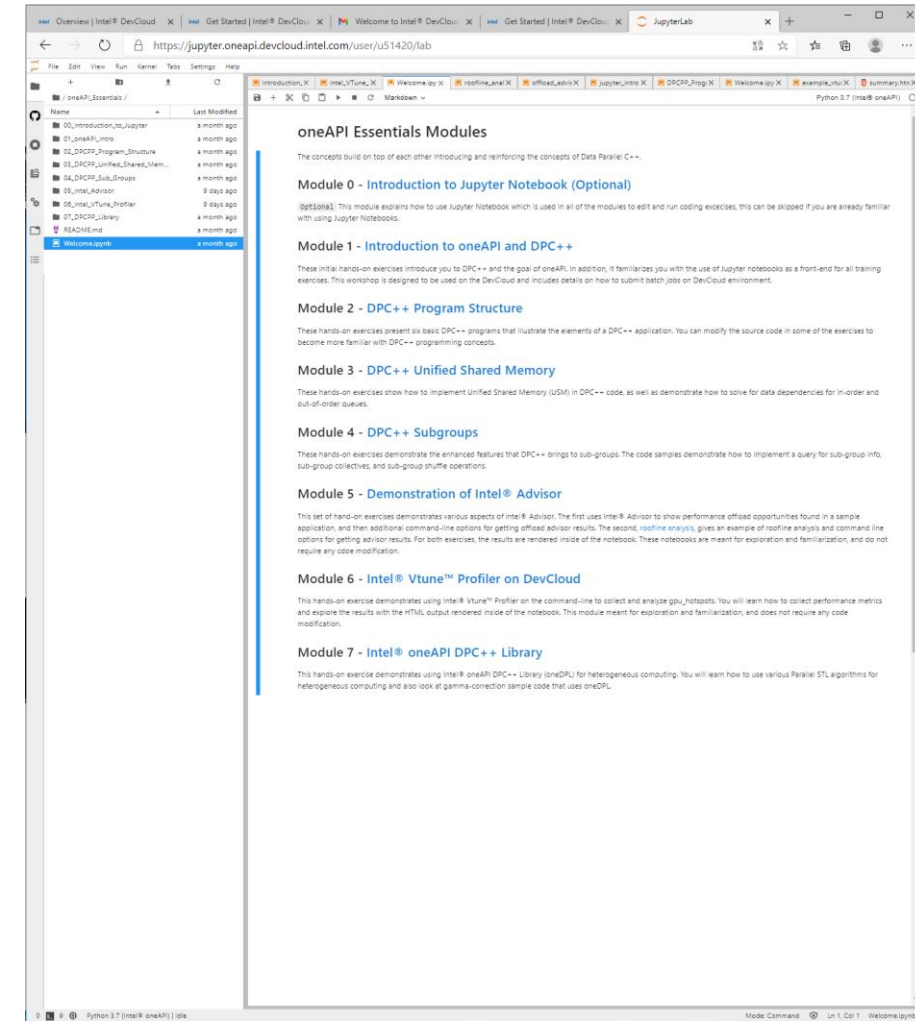
kernel3

kernel4



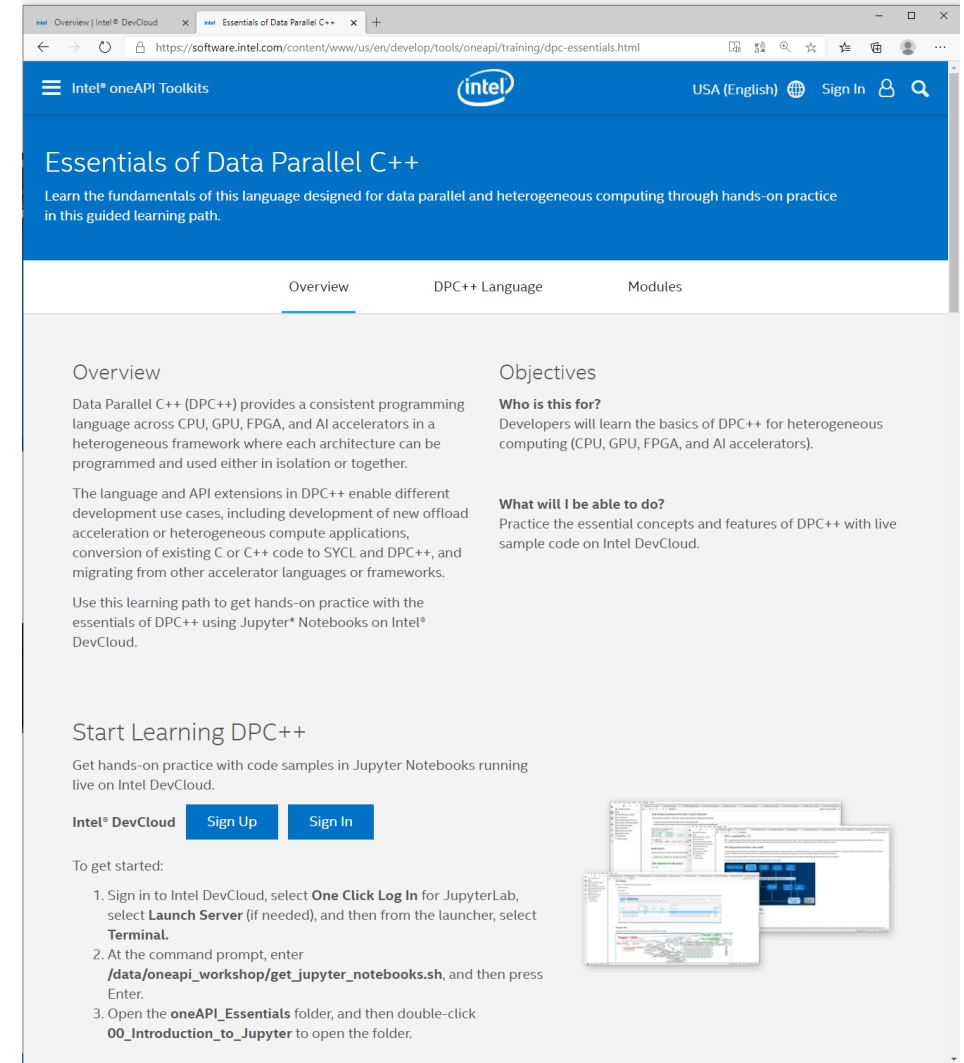
HANDS ON CODING

- Acceder a [Intel DevCloud](#)
- Jupyter Notebook: [Essentials of Data Parallel C++](#)
- Para actualizar los Jupyter Notebooks ejecutar el script:
`/data/oneapi_workshop/get_jupyter_notebooks.sh`
- Iniciar sesión en [Intel DevCloud](#): **UUID** recibido por mail



HANDS ON CODING

- Acceder a [Intel DevCloud](#)
- Jupyter Notebook: [Essentials of Data Parallel C++](#)
- 02: Program structure



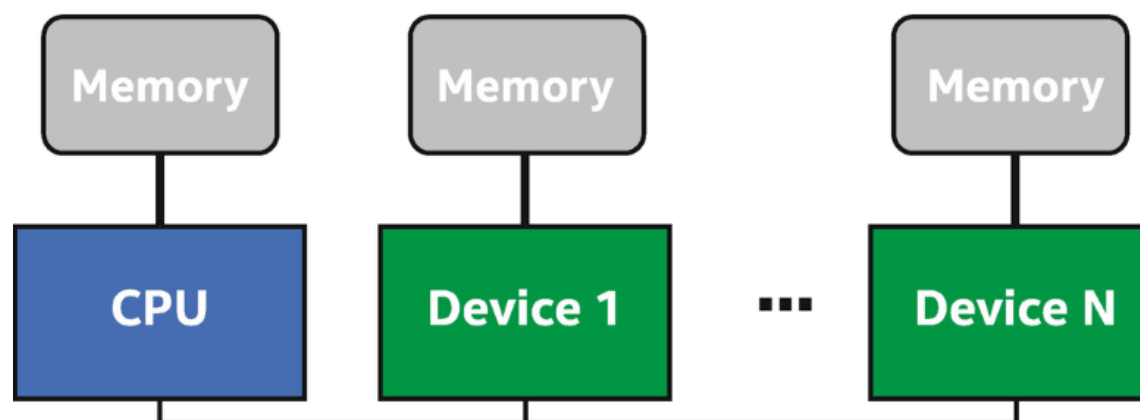
The screenshot shows the Intel oneAPI Toolkits website. The main heading is 'Essentials of Data Parallel C++'. Below it, a subheading reads: 'Learn the fundamentals of this language designed for data parallel and heterogeneous computing through hands-on practice in this guided learning path.' There are three tabs: 'Overview', 'DPC++ Language', and 'Modules'. The 'Overview' tab is selected. It contains two columns of text. The left column, titled 'Overview', describes Data Parallel C++ (DPC++) as a consistent programming language across CPU, GPU, FPGA, and AI accelerators. It mentions that the language and API extensions enable different development use cases, including development of new offload acceleration or heterogeneous compute applications, conversion of existing C or C++ code to SYCL and DPC++, and migrating from other accelerator languages or frameworks. It also states that users can get hands-on practice with the essentials of DPC++ using Jupyter* Notebooks on Intel* DevCloud. The right column, titled 'Objectives', has two sections: 'Who is this for?' which states that developers will learn the basics of DPC++ for heterogeneous computing (CPU, GPU, FPGA, and AI accelerators), and 'What will I be able to do?' which states that users will practice the essential concepts and features of DPC++ with live sample code on Intel DevCloud. Below the text, there is a section titled 'Start Learning DPC++' which says 'Get hands-on practice with code samples in Jupyter Notebooks running live on Intel DevCloud.' and includes 'Intel* DevCloud' with 'Sign Up' and 'Sign In' buttons. At the bottom, there is a 'To get started:' section with a numbered list of three steps: 1. Sign in to Intel DevCloud, select One Click Log In for JupyterLab, select Launch Server (if needed), and then from the launcher, select Terminal. 2. At the command prompt, enter /data/oneapi_workshop/get_jupyter_notebooks.sh, and then press Enter. 3. Open the oneAPI_Essentials folder, and then double-click 00_Introduction_to_Jupyter to open the folder. To the right of the text, there is a small image showing a Jupyter Notebook interface.



■ Manejo datos

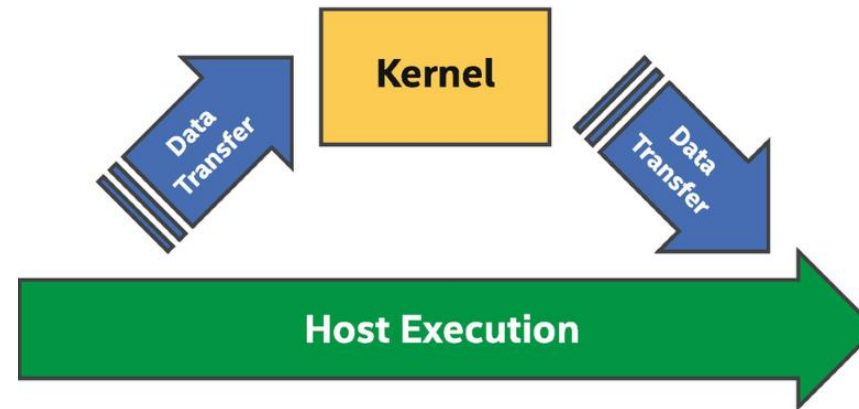
TRATAMIENTO DATOS

- Históricamente, los modelos de memoria compartida proporcionan una vista única de la memoria
 - Los dispositivos aceleradores (por ejemplo, GPU integradas) comparten memoria con una CPU host
 - Los aceleradores discretos tienen sus propias memorias locales separados de la de la CPU como se ve en la figura



TRATAMIENTO DATOS

- Manejo de multiples memorias
 - Movimiento explícito de datos: control complejo pero tedioso y propenso a errores



- Movimiento de datos implícito: el runtime es responsable de transferir los datos necesarios

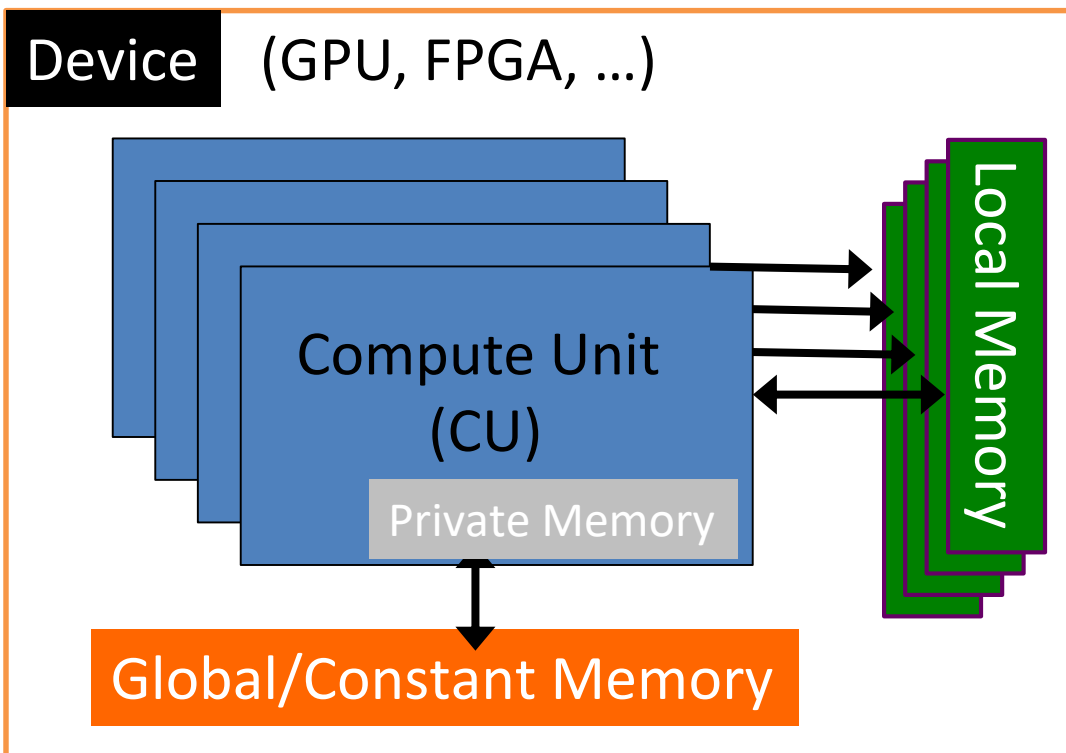
TRATAMIENTO DATOS

- Tres abstracciones
 - Memoria compartida unificada (Unified Shared Memory - USM)
 - Fácil integración que están familiarizados con C++
 - Búfer
 - Plantilla en matrices 1D, 2D y 3D
 - No se accede directamente al búfer: mediante descriptores de acceso (***accessors***)
 - Imágenes
 - Incluye soporte especial para el formato de imágenes

MODELO DE MEMORIA

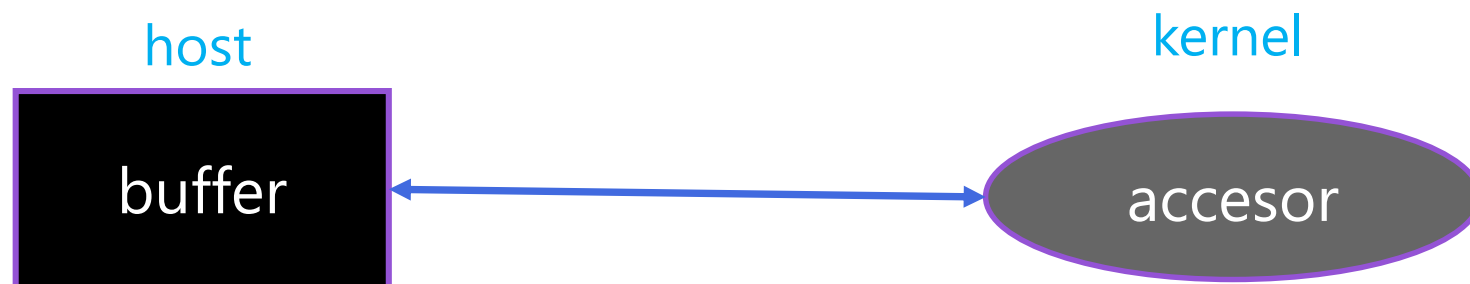
Memoria Global:

- Accesible por todos los work-ítems en todos los work-groups
- Las lecturas y escrituras pueden cachearse
- Perdura entre invocaciones de *kernels*



MODELO DE MEMORIA

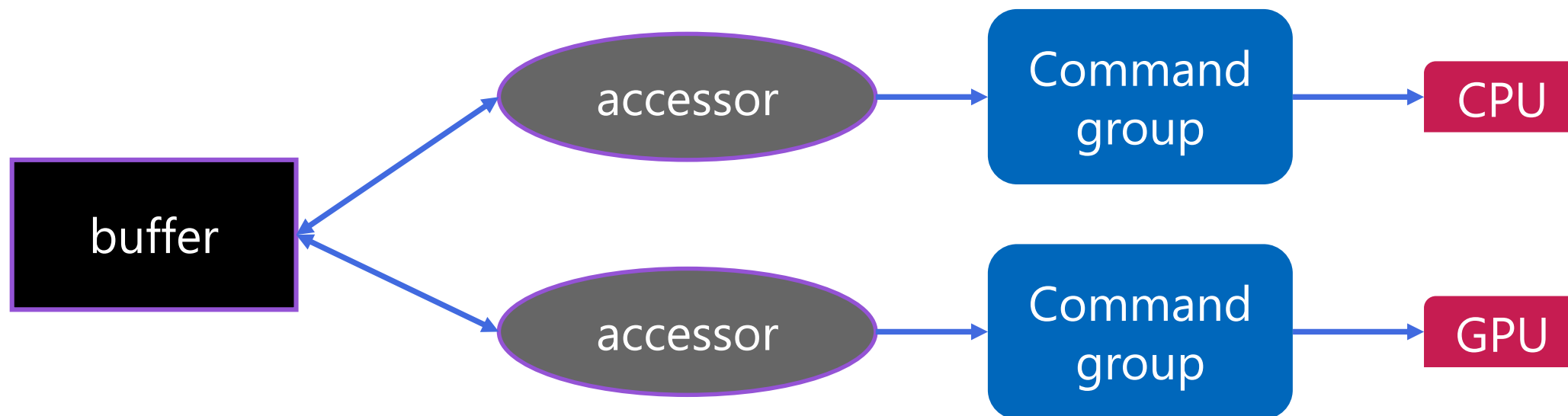
- Acceso a la memoria & Consistencia
 - La aplicación que se ejecuta host puede usar **buffer** para asignar memoria en el espacio de memoria global
 - Para acceder a los datos de los búferes dentro de un kernel, el usuario debe crear un objeto de descriptor de acceso (**accessor**)
 - Read-only/write-only/read-write
 - atomic



Cualquier variable definida dentro del **parallel_for** se almacena en mem. privada del device

MODELO DE MEMORIA

- Manejo de memoria entre host y devices
 - El almacenamiento y el acceso a la memoria se separan a través de búferes y descriptores de acceso



TRATAMIENTO DATOS

```
#include <CL/sycl.hpp>
#include<array>
using namespace sycl;
constexpr int N = 42;

int main() {
    queue Q;

    std::array<int,N> host_array;
    int *device_array = malloc_device<int>(N, Q);

    for (int i = 0; i < N; i++)
        host_array[i] = N;

    // We will learn how to simplify this example later
    Q.submit([&](handler &h) {
        // copy hostArray to deviceArray
        h.memcpy(device_array, &host_array[0], N * sizeof(int));
    });
    Q.wait();

    Q.submit([&](handler &h) {
        h.parallel_for(N, [=](id<1> i) { device_array[i]++; });
    });
    Q.wait();

    Q.submit([&](handler &h) {
        // copy deviceArray back to hostArray
        h.memcpy(&host_array[0], device_array, N * sizeof(int));
    });
    Q.wait();

    free(device_array, Q);
    return 0;
}
```

Movimiento de
datos explícito

TRATAMIENTO DATOS

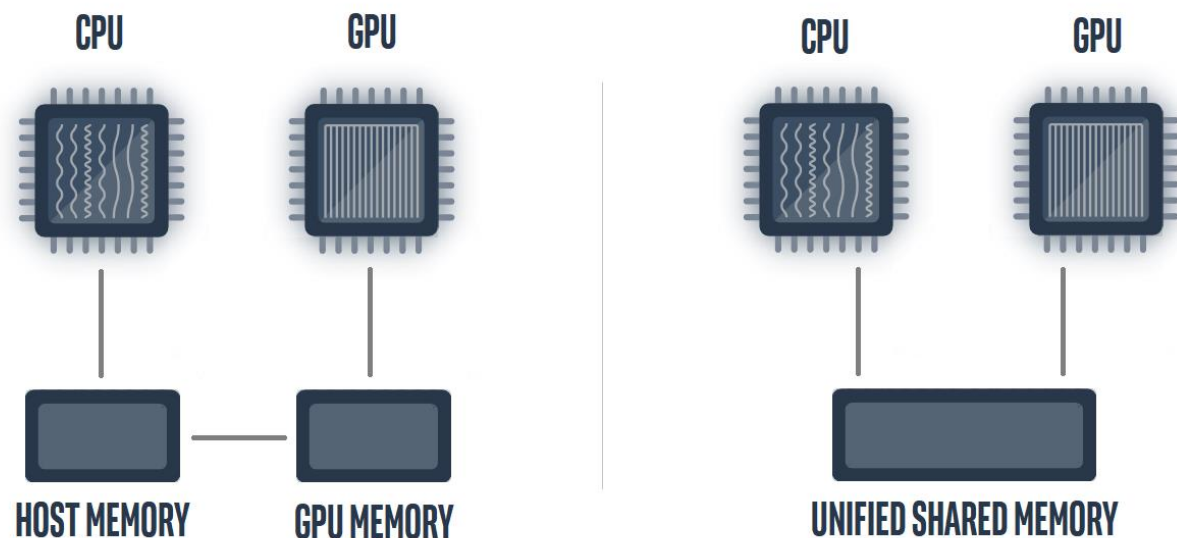
- Unified Shared Memory (USM)
 - Sólo en DPC++ (no forma parte de la especificación SYCL 1.2.1)
 - Requiere soporte hw para espacio de direcciones virtuales unificadas

Type	Description	Accessible on host?	Accessible on device?	Located on
device	Allocations in device memory	✗	✓	device
host	Allocations in host memory	✓	✓	host
shared	Allocations shared between host and device	✓	✓	Can migrate between host and device

TRATAMIENTO DATOS

- Visión de la USM

- Desarrolladores puede hacer referencia al mismo objeto de memoria en el código del host y del dispositivo mediante **Unified-Shared-Memory**
- Puntero a la memoria que facilitar la adopción de computación heterogénea



TRATAMIENTO DATOS

Manejo de memoria entre host y devices requiere la definición de los buffers y accessors: **sincronización requerida**

Host memory setup	→	<code>int main() { float *A = (float)malloc(1024*sizeof(float)); float *B = (float)malloc(1024*sizeof(float)); float *C = (float)malloc(1024*sizeof(float));</code>
Host initialization	→	<code>for (int i = 0; i < 1024; i++) A[i] = B[i] = I;</code>
Create buffers	→	<code>{ buffer<float, 1> bufA { A, range<1> {1024} }; buffer<float, 1> bufB { B, range<1> {1024} }; buffer<float, 1> bufC { C, range<1> {1024} }; queue q; q.submit([& (handler& h) { auto A = bufA.get_access<dpc_r>(h); auto B = bufB.get_access<dpc_r>(h); auto C = bufC.get_access<dpc_w>(h); h.parallel_for(range<1> {1024}, [=] (id<1> i) { C[i] = A[i] + B[i]; }); });</code>
Create accessors	→	<code>}</code>
Buffer destruction	→	<code>for (int i = 0; i < 1024; i++) std::cout << "C[" << i << "] = " << C[i] << std::endl;</code>
Host has output	→	<code>}</code>

TRATAMIENTO DATOS

- Unified Shared Memory
 - Sigue la sintaxis tipo C++/C

```
constexpr int N = 42;

queue Q;

// Allocate N floats

// C-style
float *f1 = static_cast<float*>(malloc_shared(N*sizeof(float), Q));

// C++-style
float *f2 = malloc_shared<float>(N, Q);

// C++-allocator-style
usm_allocator<float, usm::alloc::shared> alloc(Q);
float *f3 = alloc.allocate(N);

// Free our allocations
free(f1, Q.get_context());
free(f2, Q);
alloc.deallocate(f3, N);
```

TRATAMIENTO DATOS

- SYCL proporciona un modelo de **abstracción para la memoria: buffer**
 - Dependencias de datos entre *kernels* de forma elegante
- **Pero...**
 - Reemplazar todos los punteros y los arrays por buffers en un programa C++ puede ser una tarea **tediosa para los programadores**
- USM es una alternativa al esquema de punteros en DPC++
 - **Simplifica portabilidad** en el acelerador
 - Proporciona al programador el nivel de **control** deseado
 - **Complementario** al modelo con buffers

TRATAMIENTO DATOS

```
#include <CL/sycl.hpp>
using namespace sycl;
constexpr int N = 42;

int main() {
    queue Q;
    int *host_array = malloc_host<int>(N, Q);
    int *shared_array = malloc_shared<int>(N, Q);

    for (int i = 0; i < N; i++) {
        // Initialize hostArray on host
        host_array[i] = i;
    }

    // We will learn how to simplify this example later
    Q.submit([&](handler &h) {
        h.parallel_for(N, [=](id<1> i) {
            // access sharedArray and hostArray on device
            shared_array[i] = host_array[i] + 1;
        });
    });
    Q.wait();

    for (int i = 0; i < N; i++) {
        // access sharedArray on host
        host_array[i] = shared_array[i];
    }

    free(shared_array, Q);
    free(host_array, Q);
    return 0;
}
```

USM (malloc)

shared_array

Acceso explícito en el device

TRATAMIENTO DATOS

USM

```
constexpr int N = 42;

queue Q;

std::array<int, N> host_array;
int *device_array = malloc_device<int>(N, Q);
for (int i = 0; i < N; i++)
    host_array[i] = N;

Q.submit([&](handler& h) {
    // copy hostArray to deviceArray
    h.memcpy(device_array, &host_array[0], N * sizeof(int));
});

Q.wait(); // needed for now (we learn a better way later)

Q.submit([&](handler& h) {
    h.parallel_for(N, [=](id<1> i) {
        device_array[i]++;
    });
});

Q.wait(); // needed for now (we learn a better way later)

Q.submit([&](handler& h) {
    // copy deviceArray back to hostArray
    h.memcpy(&host_array[0], device_array, N * sizeof(int));
});

Q.wait(); // needed for now (we learn a better way later)

free(device_array, Q);
```

Implicit

```
constexpr int N = 42;
```

```
queue Q;
```

```
int* host_array = malloc_host<int>(N, Q);
int* shared_array = malloc_shared<int>(N, Q);
for (int i = 0; i < N; i++)
    host_array[i] = i;
```

```
Q.submit([&](handler& h) {
    h.parallel_for(N, [=](id<1> i) {
        // access sharedArray and hostArray on device
        shared_array[i] = host_array[i] + 1;
    });
});
```

```
Q.wait();
```

```
free(shared_array, Q);
free(host_array, Q);
```

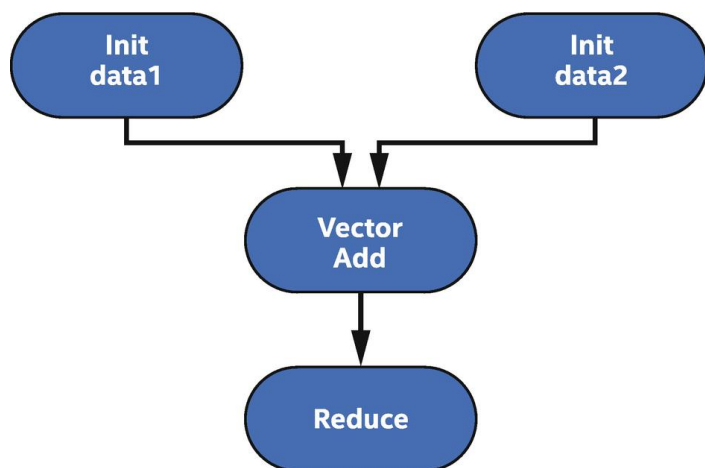
Explicit



■ Exec. & Kernel schedule

PLANIFICACIÓN DE KERNELS

- Cómo las Command Groups declaran las dependencias (in-order queue)



Ejecución no puede Solaparse si hay tareas con dependencias

```

constexpr int N = 42;

queue Q{property::queue::in_order()};

int *data1 = malloc_shared<int>(N, Q);
int *data2 = malloc_shared<int>(N, Q);

Q.parallel_for(N, [=](id<1> i) { data1[i] = 1; });
Q.parallel_for(N, [=](id<1> i) { data2[i] = 2; });
Q.parallel_for(N, [=](id<1> i) { data1[i] += data2[i]; });

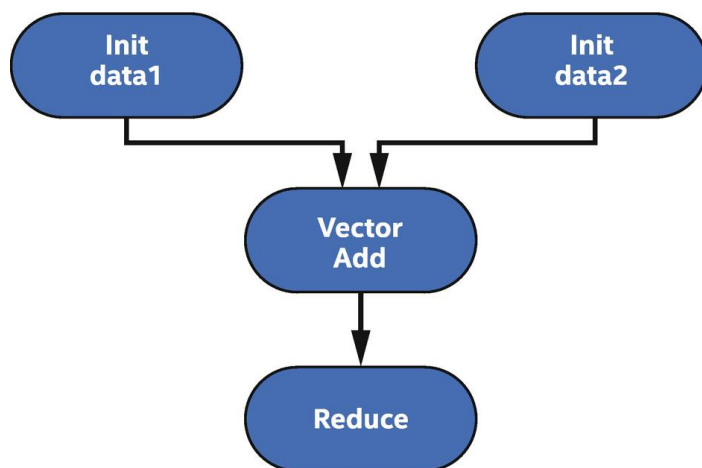
Q.single_task([=]() {
    for (int i = 1; i < N; i++)
        data1[0] += data1[i];

    data1[0] /= 3;
});

Q.wait();
assert(data1[0] == N);
  
```

PLANIFICACIÓN DE KERNELS

- Cómo las Command Groups declaran las dependencias (out-of-order queue)



```

constexpr int N = 42;
queue Q;

int *data1 = malloc_shared<int>(N, Q);
int *data2 = malloc_shared<int>(N, Q);

auto e1 = Q.parallel_for(N,
    [=](id<1> i) { data1[i] = 1; });

auto e2 = Q.parallel_for(N,
    [=](id<1> i) { data2[i] = 2; });

auto e3 = Q.parallel_for(range{N}, {e1, e2},
    [=](id<1> i) { data1[i] += data2[i]; });

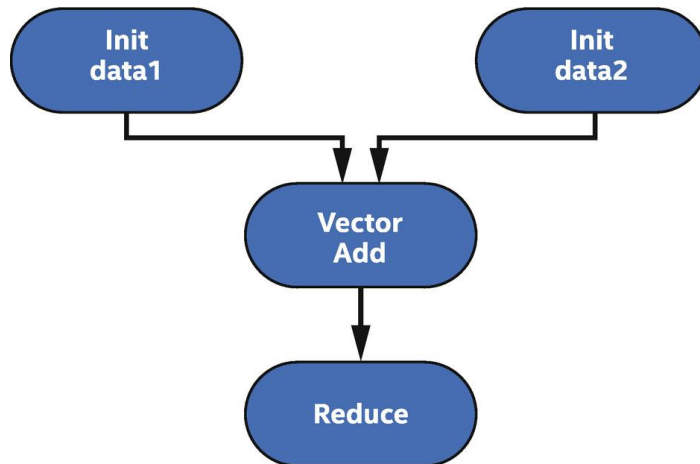
Q.single_task(e3, [=]() {
    for (int i = 1; i < N; i++)
        data1[0] += data1[i];

    data1[0] /= 3;
});

Q.wait();
assert(data1[0] == N);
  
```

PLANIFICACIÓN DE KERNELS

- Cómo las Command Groups declaran las dependencias (out-of-order queue)



```

constexpr int N = 42;
queue Q;

int *data1 = malloc_shared<int>(N, Q);
int *data2 = malloc_shared<int>(N, Q);

auto e1 = Q.parallel_for(N,
    [=](id<1> i) { data1[i] = 1; });
auto e2 = Q.parallel_for(N,
    [=](id<1> i) { data2[i] = 2; });
auto e3 = Q.parallel_for(range{N}, {e1, e2},
    [=](id<1> i) { data1[i] += data2[i]; });

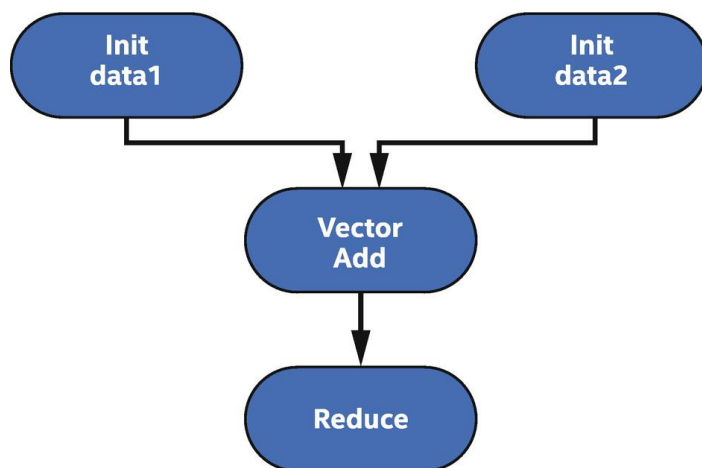
Q.single_task(e3, [=]() {
    for (int i = 1; i < N; i++)
        data1[0] += data1[i];

    data1[0] /= 3;
});

Q.wait();
assert(data1[0] == N);
  
```


PLANIFICACIÓN DE KERNELS

- Como las Command Groups declaran las dependencias (out-of-order queue)



depends_on() método interno a un command group también disponible
(consultar: [Unified Shared Memory \(intel.com\)](https://www.intel.com/content/www/us/en/develop/articles/unified-shared-memory.html))

```

constexpr int N = 42;
queue Q;

int *data1 = malloc_shared<int>(N, Q);
int *data2 = malloc_shared<int>(N, Q);

auto e1 = Q.parallel_for(N,
    [=](id<1> i) { data1[i] = 1; });

auto e2 = Q.parallel_for(N,
    [=](id<1> i) { data2[i] = 2; });

auto e3 = Q.parallel_for(range{N}, {e1, e2},
    [=](id<1> i) { data1[i] += data2[i]; });

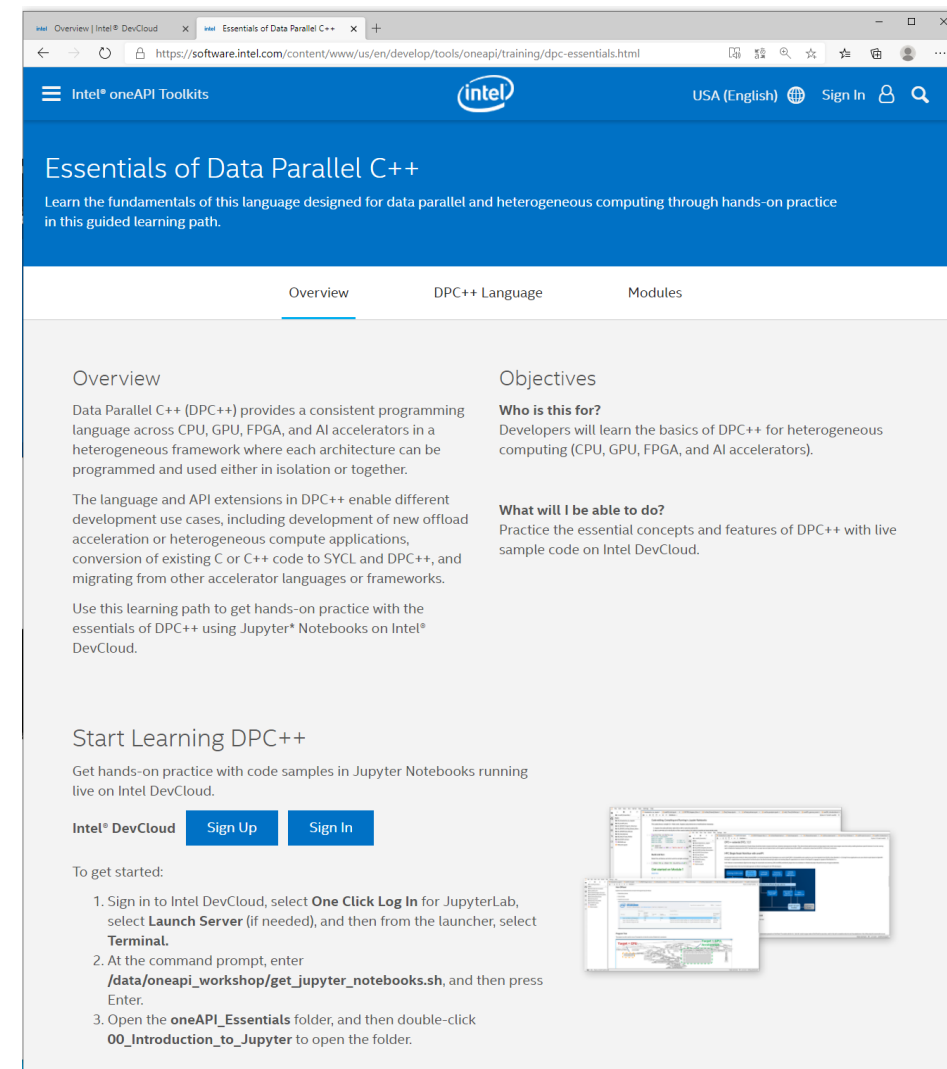
Q.single_task(e3, [=]() {
    for (int i = 1; i < N; i++)
        data1[0] += data1[i];

    data1[0] /= 3;
});

Q.wait();
assert(data1[0] == N);
  
```

HANDS ON CODING

- Acceder a [Intel DevCloud](#)
- Jupyter Notebook: [Essentials of Data Parallel C++](#)
- 03: DPCPP Unified Shared Memory



The screenshot shows the Intel oneAPI Toolkits website. The main heading is 'Essentials of Data Parallel C++'. Below it, a subheading reads: 'Learn the fundamentals of this language designed for data parallel and heterogeneous computing through hands-on practice in this guided learning path.' The page has a navigation bar with 'Overview', 'DPC++ Language', and 'Modules'. The 'Overview' section is active and contains the following text:

Overview

Data Parallel C++ (DPC++) provides a consistent programming language across CPU, GPU, FPGA, and AI accelerators in a heterogeneous framework where each architecture can be programmed and used either in isolation or together.

The language and API extensions in DPC++ enable different development use cases, including development of new offload acceleration or heterogeneous compute applications, conversion of existing C or C++ code to SYCL and DPC++, and migrating from other accelerator languages or frameworks.

Use this learning path to get hands-on practice with the essentials of DPC++ using Jupyter® Notebooks on Intel® DevCloud.

Objectives

Who is this for?
Developers will learn the basics of DPC++ for heterogeneous computing (CPU, GPU, FPGA, and AI accelerators).

What will I be able to do?
Practice the essential concepts and features of DPC++ with live sample code on Intel DevCloud.

Start Learning DPC++

Get hands-on practice with code samples in Jupyter Notebooks running live on Intel DevCloud.

Intel® DevCloud [Sign Up](#) [Sign In](#)

To get started:

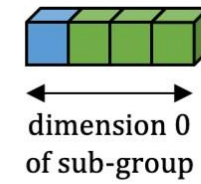
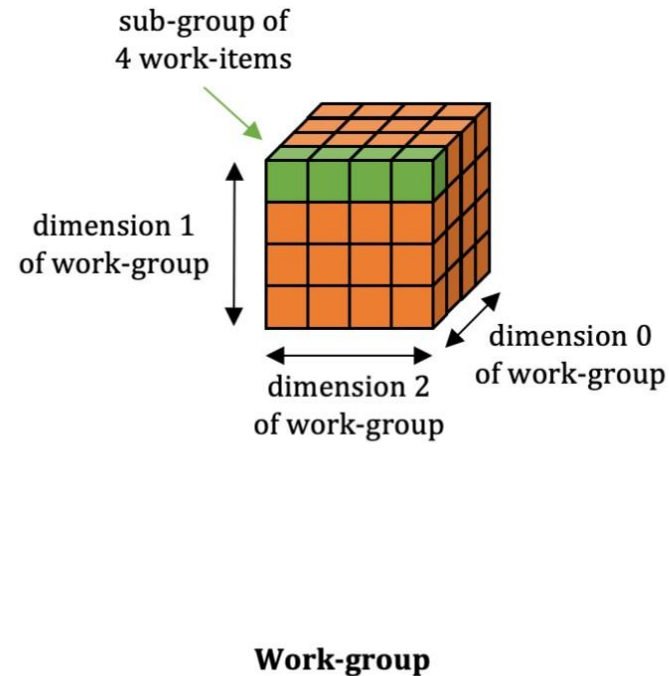
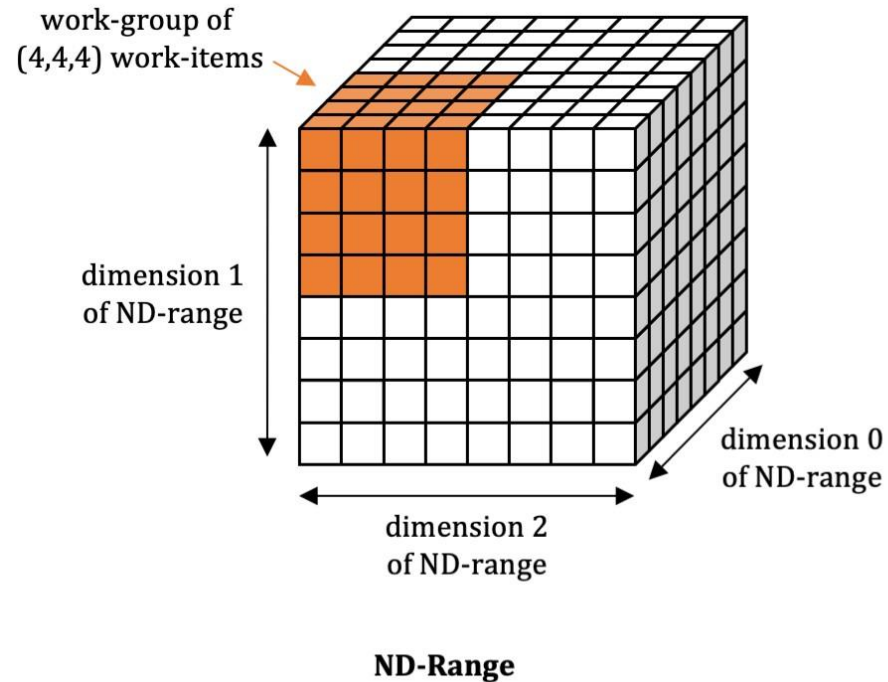
1. Sign in to Intel DevCloud, select **One Click Log In** for JupyterLab, select **Launch Server** (if needed), and then from the launcher, select **Terminal**.
2. At the command prompt, enter `/data/oneapi_workshop/get_jupyter_notebooks.sh`, and then press Enter.
3. Open the **oneAPI_Essentials** folder, and then double-click **00_Introduction_to_Jupyter** to open the folder.

On the right side of the page, there is a small image showing a Jupyter Notebook interface with code and output.



■ Sub-groups

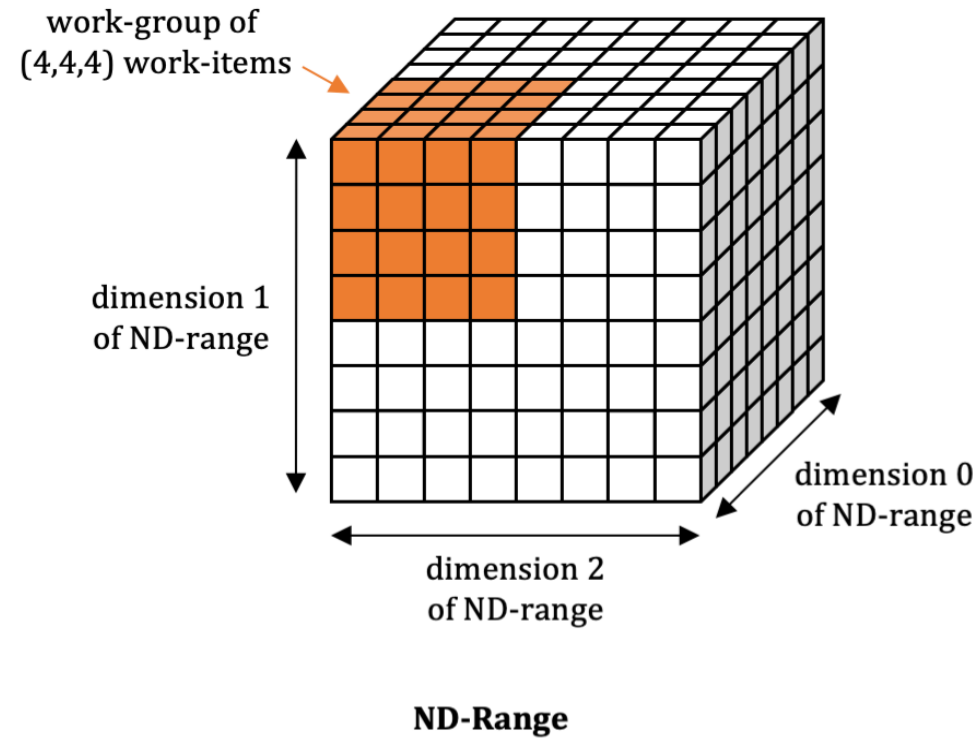
CONSTRUCCIÓN DPC++



Vocabulario DPC++ sigue y extiende el modelo CUDA, OpenCL, SYCL.

CONSTRUCCIÓN DPC++

- Las funciones kernels son invocadas con la clase **ND-Range**.

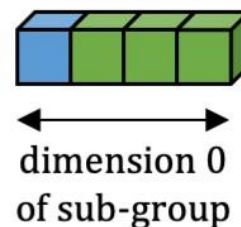


CONSTRUCCIÓN DPC++

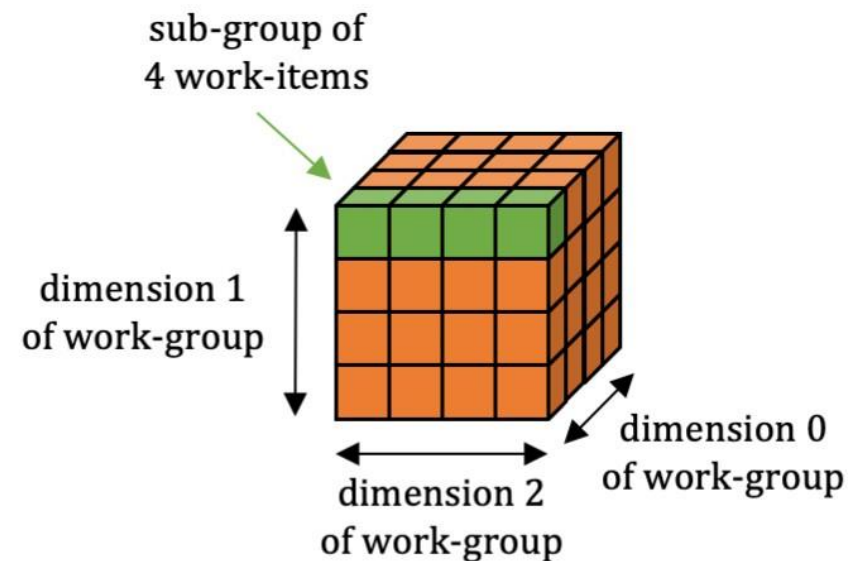
- **Work-item**
 - Representa una estancia individual de la función kernel
- **Work-group**
 - Todo el espacio de iteración se divide en grupos más pequeños denominados work-groups, work-item dentro de un work-group se planifican en una sola unidad de cómputo (Compute Unit-CU) del hw
- **Sub-group**
 - Un subconjunto de work-items dentro de un work-group se ejecuta simultáneamente, y puede mapearse en un vector hw (DPC++)

CONSTRUCCIÓN DPC++

- SYCL (concurrentemente) proporciona work-groups
 - work-items pueden agruparse en single items (1D)
 - ... pero
 - work-groups pueden ser 3D (o 2D)
 - work-subgroup (solo en DPC++) nos dan la opción 2D (útil en espacios 3D)



Sub-group

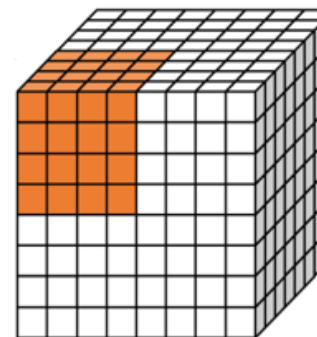


Work-group

¿CÓMO SE MAPEA EN EL HW (INTEL GEN11 GRAPHICS)?



All work-items in a **work-group** are scheduled on one Compute Unit, which has its own local memory

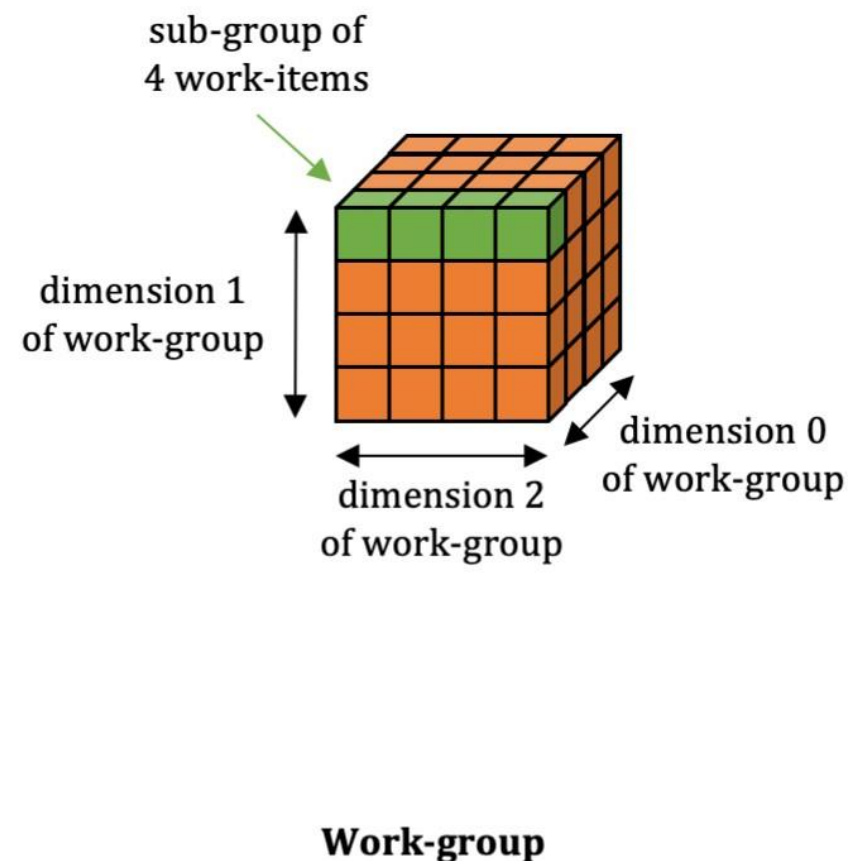


All work-items in a **sub-group** are mapped to vector hardware



SUB-GROUPS

- Un subconjunto de work-items en un work-group **puede asignarse al hardware vectorial**
- ¿Por qué usar subgrupos?
 - Los work-items de un sub-group pueden comunicarse directamente mediante operaciones **shuffle**, sin operaciones con memoria explícitas
 - Los work-items de un subgrupo pueden sincronizarse mediante barreras de subgrupo y **garantizar la coherencia de la memoria** mediante vallas de memoria de subgrupos
 - Los work-items de un subgrupo tienen acceso **operaciones colectivas** a nivel de subgrupos, lo que proporciona una rápida implementación de patrones paralelos comunes



SUB-GROUPS

- Sub-Groups
 - El identificador de la clase `sub_groups` se puede obtener con el `nd_item` usando `get_sub_group()`
 - Una vez disponible en manejador del sub-group, más información está disponible e incluso se pueden hacer operaciones `shuffle` o `collectivas`

```
h.parallel_for(nd_range<1>(N,B), [=](nd_item<1> item){  
  
    ONEAPI::sub_group sg = item.get_sub_group();  
    // KERNEL CODE  
});
```

SUB-GROUPS

- El manejador del subgroup puede solicitar información extra:
 - `get_local_id()` devuelve el índice del work-item con el subgroup
 - `get_local_range()` devuelve el tamaño del sub-grupo
 - `get_group_id()` devuelve el índice el sub-grupo
 - `get_group_range()` devuelve el número del sub-grupo con el work-group asociado

```
h.parallel_for(nd_range<1>(N,B), [=](nd_item<1> item){
```

```
    ONEAPI::sub_group sg = item.get_sub_group();
    if(sg.get_local_id()[0] == 0){
        out << "sub_group id: " << sg.get_group_id()[0]
            << " of " << sg.get_group_range()[0]
            << ", size=" << sg.get_local_range()[0]
            << endl;
    }
});
```

```
Device : Intel(R) Gen9
sub_group id: 0 of 4, size=16
sub_group id: 3 of 4, size=16
sub_group id: 1 of 4, size=16
sub_group id: 2 of 4, size=16
```

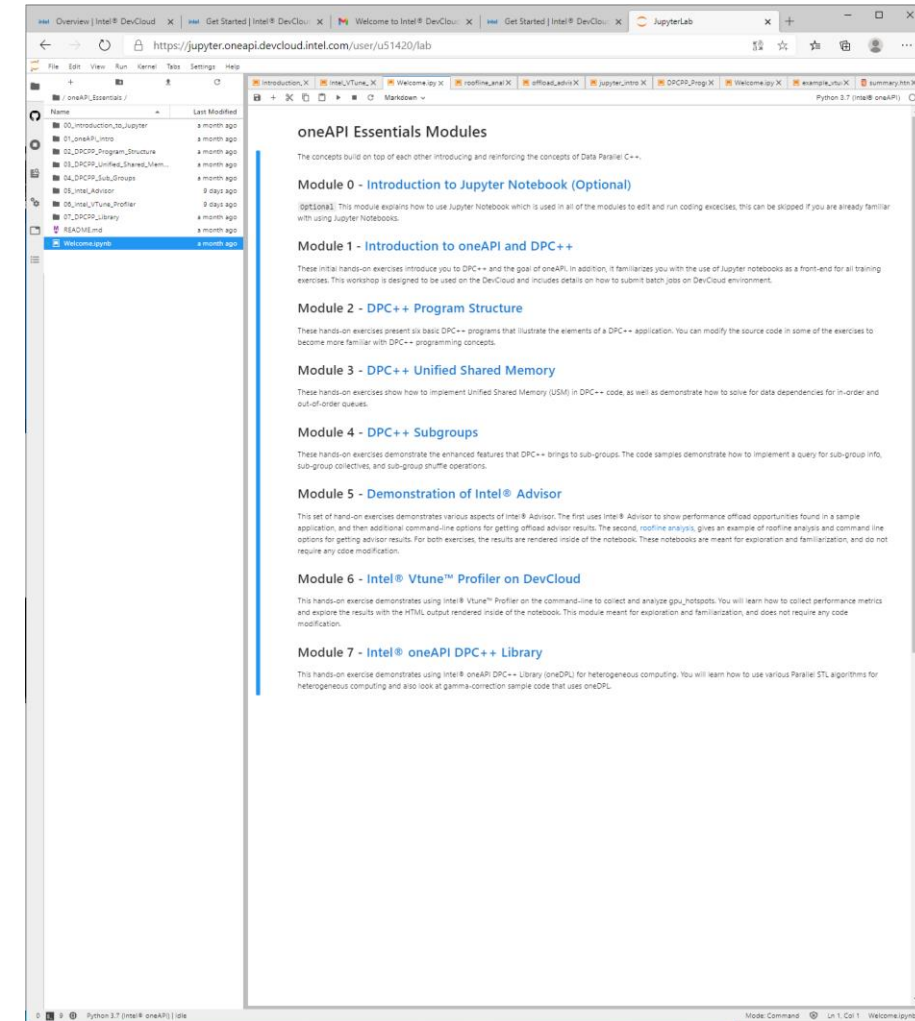
SUB-GROUPS

- Op. Colectivas Sub-Group
 - Las funciones colectivas proporcionan la implementación de **patrones paralelos comunes**
 - Proporcionan implementaciones eficientes como la **función incremento**

```
h.parallel_for(nd_range<1>(N,B), [=](nd_item<1> item){  
  
    ONEAPI::sub_group sg = item.get_sub_group();  
    size_t i = item.get_global_id(0);  
  
    /* Collectives */  
    data[i] = reduce(sg, data[i], std::plus<>());  
    // data[i] = reduce(sg, data[i], std::maximum<>());  
    // data[i] = reduce(sg, data[i], std::minimum<>());  
});
```

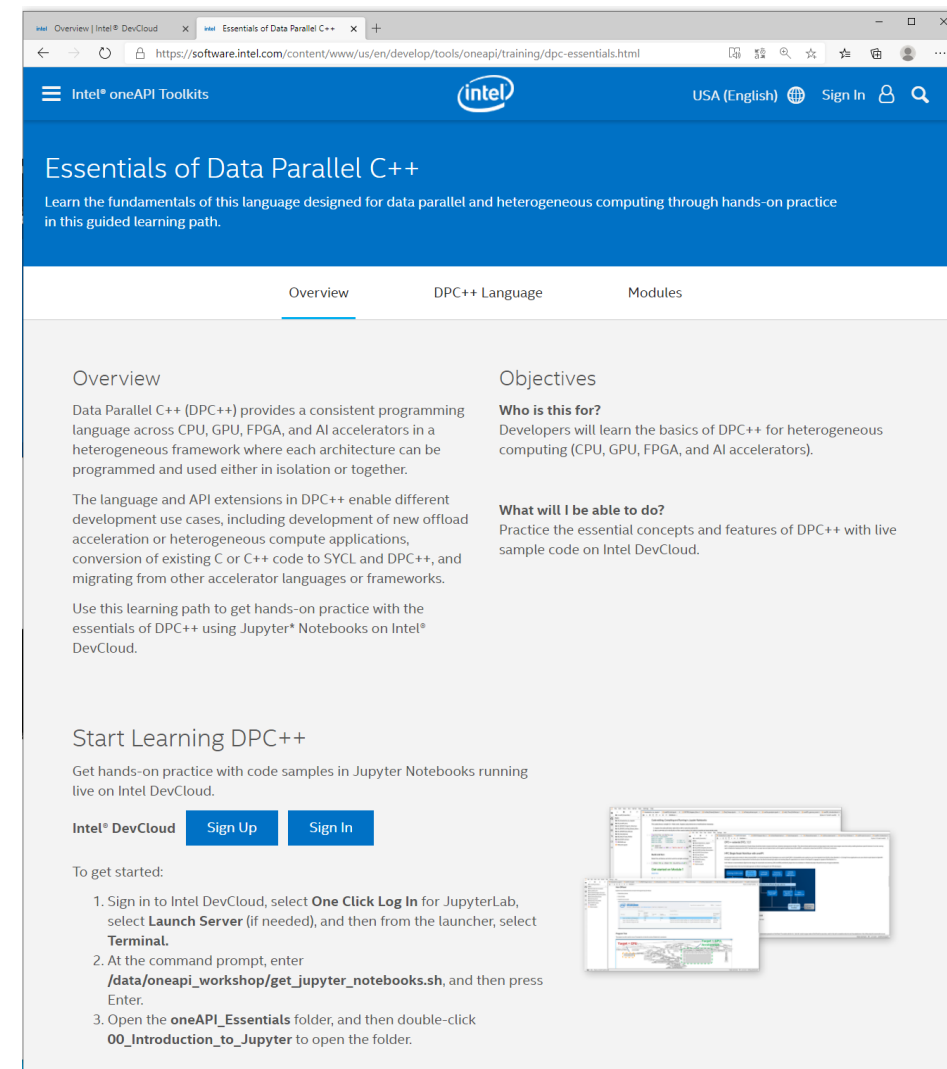
HANDS ON CODING

- Acceder a [Intel DevCloud](#)
- Jupyter Notebook: [Essentials of Data Parallel C++](#)
- Para actualizar los Jupyter Notebooks ejecutar el script:
/data/oneapi_workshop/get_jupyter_notebooks.sh
- Iniciar sesión en [Intel DevCloud](#): **UUID** recibido por mail



HANDS ON CODING

- Acceder a [Intel DevCloud](#)
- Jupyter Notebook: [Essentials of Data Parallel C++](#)
- 04: DPC++ Sub-groups



The screenshot shows the Intel oneAPI Toolkits website. The main heading is 'Essentials of Data Parallel C++'. Below it, a subheading reads: 'Learn the fundamentals of this language designed for data parallel and heterogeneous computing through hands-on practice in this guided learning path.' The page has three tabs: 'Overview', 'DPC++ Language', and 'Modules'. The 'Overview' tab is active. It contains an 'Overview' section with text about Data Parallel C++ (DPC++) and its use across different hardware. It also has an 'Objectives' section with two sub-sections: 'Who is this for?' and 'What will I be able to do?'. At the bottom, there is a 'Start Learning DPC++' section with instructions on how to get started, including signing up for Intel DevCloud and running a Jupyter Notebook. A small image of a Jupyter Notebook interface is shown on the right side of the page.

Intel® oneAPI Toolkits

USA (English) Sign In

Essentials of Data Parallel C++

Learn the fundamentals of this language designed for data parallel and heterogeneous computing through hands-on practice in this guided learning path.

Overview DPC++ Language Modules

Overview

Data Parallel C++ (DPC++) provides a consistent programming language across CPU, GPU, FPGA, and AI accelerators in a heterogeneous framework where each architecture can be programmed and used either in isolation or together.

The language and API extensions in DPC++ enable different development use cases, including development of new offload acceleration or heterogeneous compute applications, conversion of existing C or C++ code to SYCL and DPC++, and migrating from other accelerator languages or frameworks.

Use this learning path to get hands-on practice with the essentials of DPC++ using Jupyter® Notebooks on Intel® DevCloud.

Objectives

Who is this for?
Developers will learn the basics of DPC++ for heterogeneous computing (CPU, GPU, FPGA, and AI accelerators).

What will I be able to do?
Practice the essential concepts and features of DPC++ with live sample code on Intel DevCloud.

Start Learning DPC++

Get hands-on practice with code samples in Jupyter Notebooks running live on Intel DevCloud.

Intel® DevCloud [Sign Up](#) [Sign In](#)

To get started:

1. Sign in to Intel DevCloud, select **One Click Log In** for JupyterLab, select **Launch Server** (if needed), and then from the launcher, select **Terminal**.
2. At the command prompt, enter `/data/oneapi_workshop/get_jupyter_notebooks.sh`, and then press Enter.
3. Open the **oneAPI_Essentials** folder, and then double-click **00_Introduction_to_Jupyter** to open the folder.

NOTICES & DISCLAIMERS

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request. No product or component can be absolutely secure. Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Copyright ©, Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Core, VTune, and OpenVINO are trademarks of Intel Corporation or its subsidiaries in the U.S. and other countries.

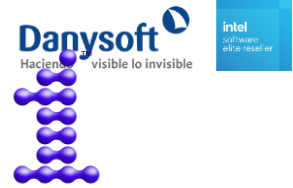
Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

ONEAPI RESOURCES

Use *Slideshow mode* to click links



oneAPI Industry Initiative

[oneAPI Initiative site](#) [Overview video](#) [3.40]

[oneAPI Industry Specification](#)

[Ecosystem Support](#)

Data Parallel C++ (DPC++)

▪ Videos

[DPC++ Overview](#) [3.41]

[DPC++: Open Alternative for Cross-Architecture Development](#)

[Q&A - Intel Senior Fellow Geoff Lowney](#) [12.05]

[DPC++ open source project](#) on Github

[oneAPI Programming Guide](#)

▪ DPC++ book [4 preview chapters](#)

Intel® oneAPI Products

Includes domain-specific toolkits

▪ [Intel® oneAPI Toolkits](#)

– [Product Brief](#)

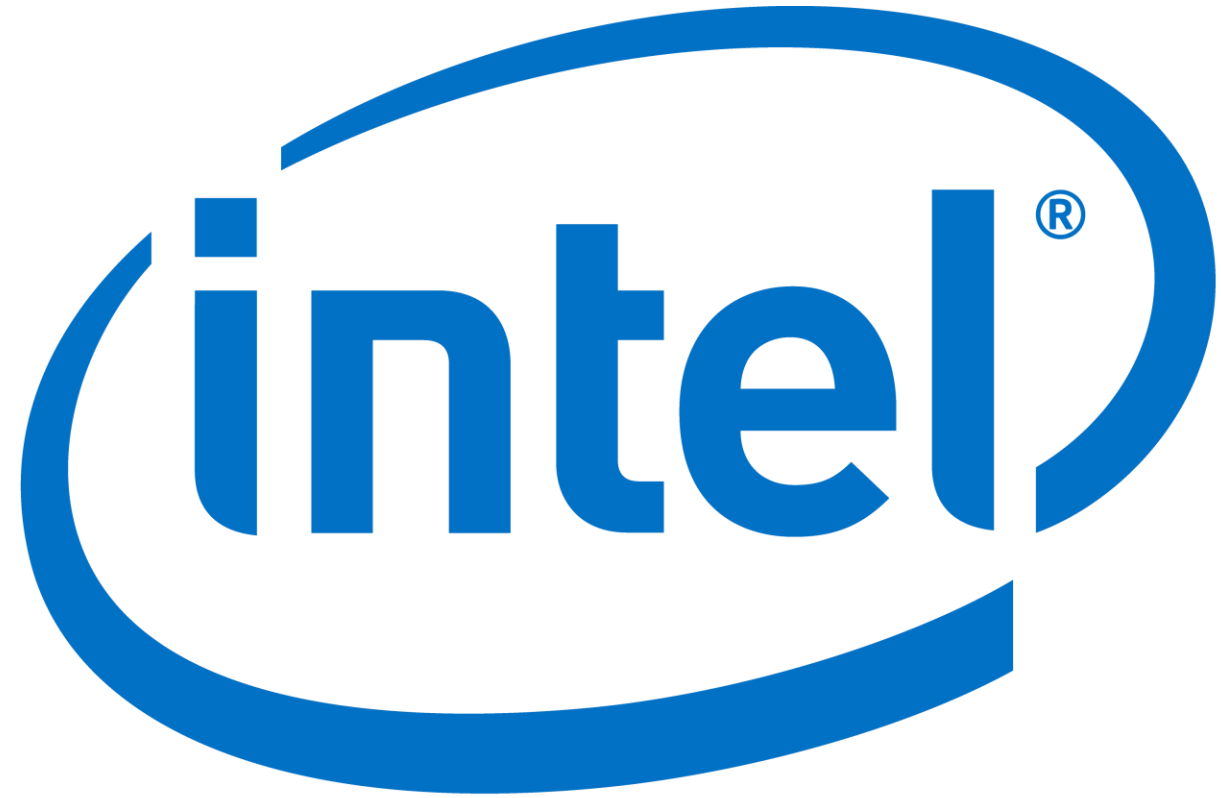
– [Documentation](#)

– [Training](#)

– [Code Samples](#) to get started (see domain-specific toolkits for their samples)

▪ [Intel® DevCloud](#) – Test workloads, code & oneAPI tools on a variety of Intel® architecture - free-of-charge

Free oneAPI, DPC++ & Intel oneAPI Products [webinars & quick how-to's](#)



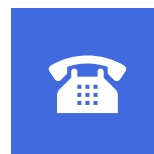
Software

Contacto



Dirección

Avda. de la industria 4, edif. 1
28108 Alcobendas | Madrid | España



Teléfono

[+34] 91 663 8683



Correo:

info@danysoft.com



Sitio Web

www.danysoft.com/intel





intel®

Danysoft 
Haciendo visible lo invisible

Gracias