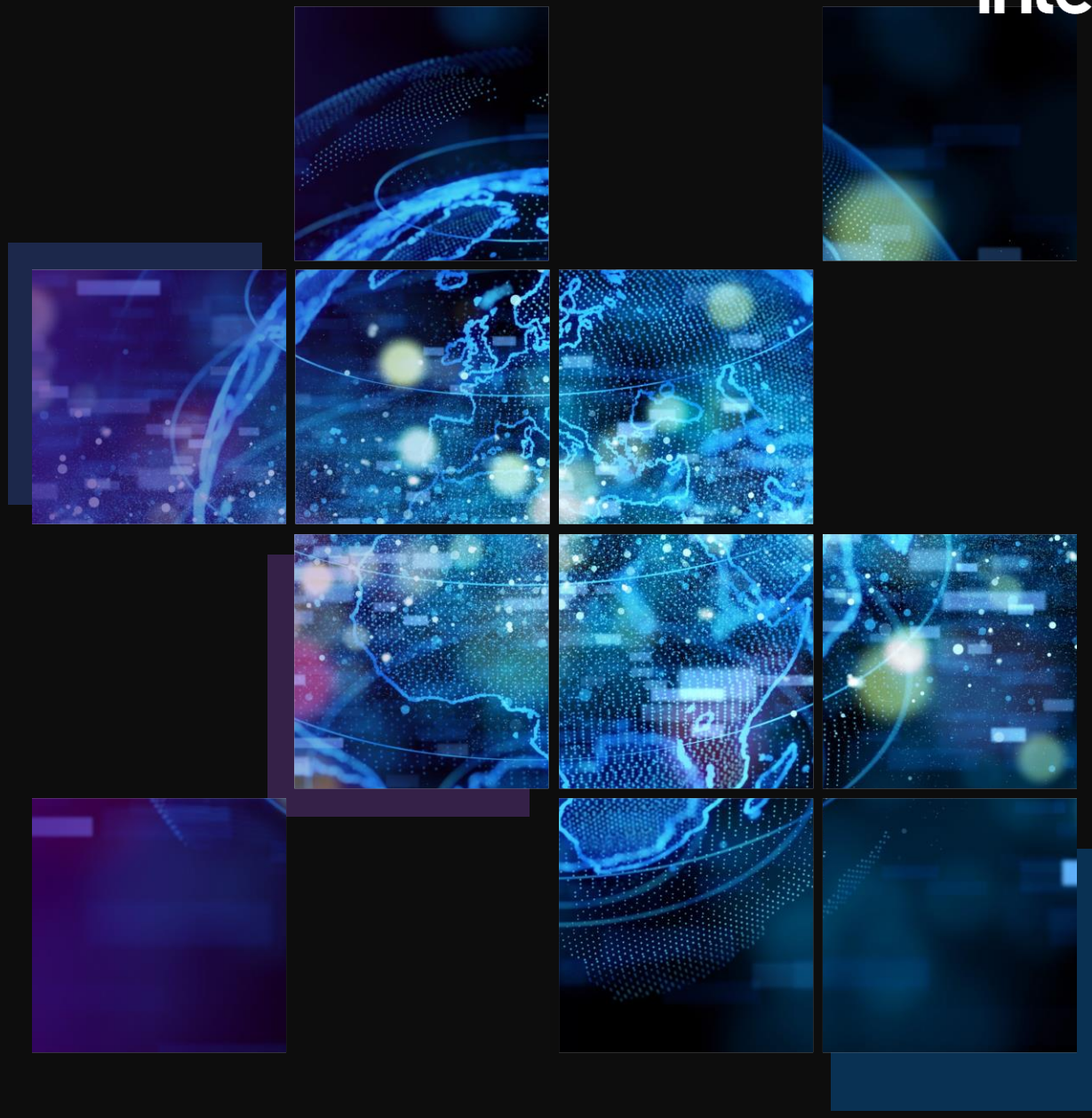
oneAPI

intel

Sesión 1: DPC++

Carlos García Sánchez



-

Conoce de la mano de expertos oneAPI

La solución para programadores de Intel

Danysoft

Haciendo visible lo invisible

intel
software elite reseller

Agenda

- SYCL y DPC++
- Anatomía DPC++
- Modelo Memoria
- Modelo Ejecución

OBJETIVO

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

RETOS PROGRAMMING EN ARQUITECTURAS PARALELAS

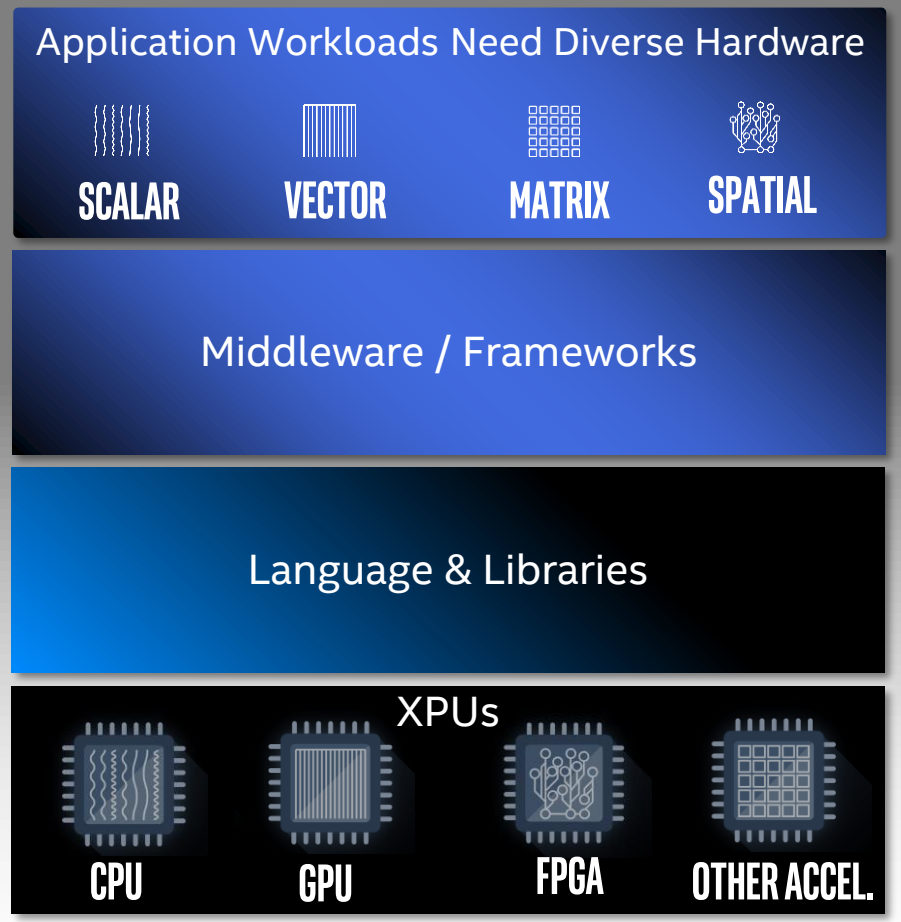
Crecimiento de cargas de trabajo especializadas

Variedad de hardware

No hay lenguaje de programación común o API

No existen herramientas en todas las plataformas

Cada plataforma requiere una adaptar el software



ONEAPI INICIATIVA INDUSTRIAL
ALTERNATIVA A SOLUCIÓN DE ÚNICO PROVEEDOR

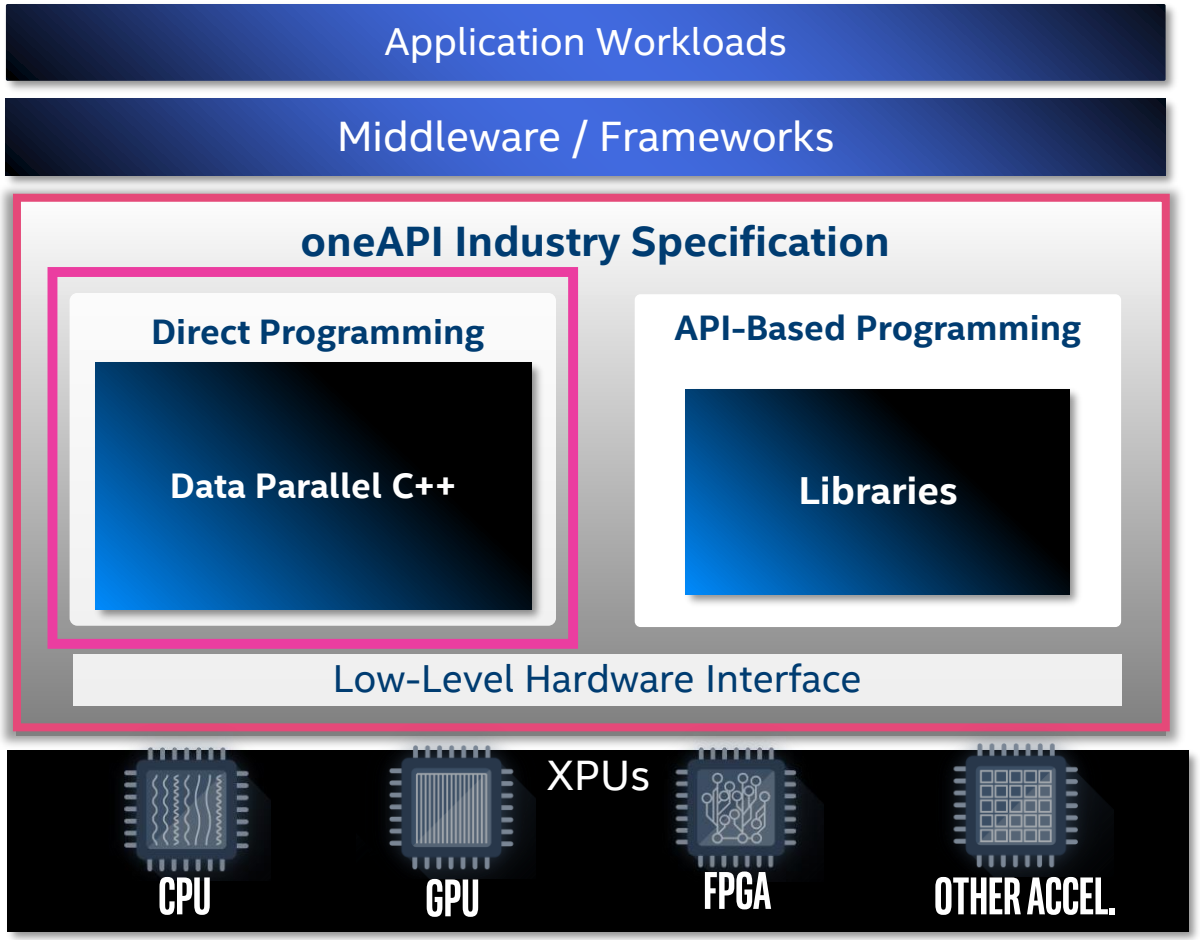
Un lenguaje basado en estándares, DPC++, basado en C++ y SYCL

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

Algunas capacidades pueden diferir según la arquitectura y el ajuste personalizado seguirá siendo necesario.

- SYCL y DPC++

¿ QUÉ ES DATA PARALLEL C++?

- Data Parallel C++ ⇔ DPC++
  - C++ y el estándar de SYCL con algunas extensiones
- Basado en C++
    - Beneficios en la productividad al soportar construcciones C++
- Incorpora el estándar SYCL
    - Con soporte de paralelismo de datos y la programación heterogénea

¿QUÉ ES SYCL?

- SYCL es propuesta de estandarización para definir modelo de programación que soporte paralelismo datos
  - Pronunciado como `sickle'  `sick ell' /"sik(@)l/
  - Capa abstracción para paralelismo de datos multi-plataforma
- Código fuente en un único fichero
- Extensión de C++ (últimas versiones soportadas)
- Standard definido por el Khronos Group
  - Intel es participante del standard
- La mayoría de DPC++ es una parte de SYCL
  - Intel contribuye con nuevas características

DPC++ EXTIENDE SYCL 1.2.1

- Mejora Productividad
  - Las cosas simples deben ser simples de expresar
  - Reduzca *verbosidad* y la sobrecarga de portabilidad del programador
- Mejora Rendimiento
  - Los programadores controlan la ejecución del programa
  - Explota características específicas de hardware
- DPC++: colaboración abierta para mejorar el estándar SYCL
  - Implementación de código abierto con compatibilidad LLVM
  - Las extensiones DPC++ tienen como objetivo convertirse en extensiones principales de SYCL

KERNELS PARALELOS

- Expresar paralelismo mediante *kernels* permite que varias instancias de una operación se ejecuten en paralelo
- Útil para descargar la ejecución paralela de un bucle for-loop con iteraciones independiente
- Los *kernels* paralelos se expresan utilizando la función parallel_for

for-loop in CPU application

```
for (int i=0; i<1024; i++) {
    a[i] = b[i] + c[i];
}
```

Offload to Accelerator using **parallel_for**

```
h.parallel_for(range<1>(1024), [=] (id<1> i){
   a[i] = b[i] + c[i];
});
```

KERNELS PARALELOS BÁSICOS

- La funcionalidad de los *kernels* se expresa a través de clases de rango, id y elementos
  - range se utiliza para describir el espacio de iteración de la ejecución paralela
  - id se utiliza para indexar una instancia individual de un kernel en una ejecución paralela
  - item representa una instancia individual de la función del kernel

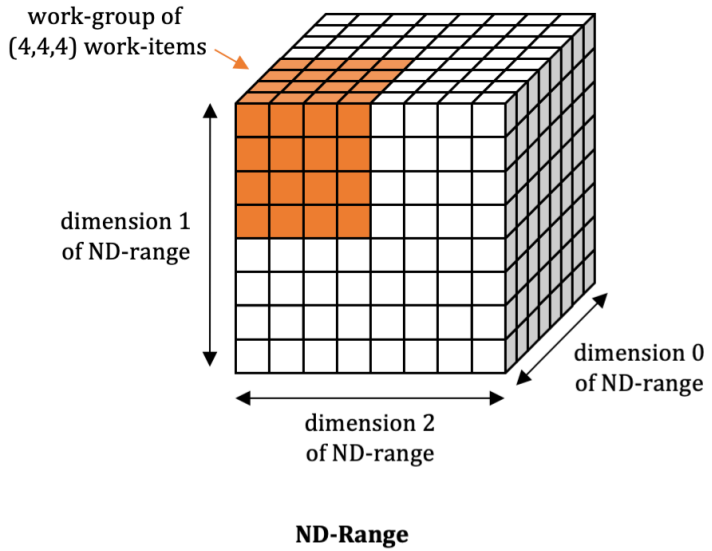
```
h.parallel_for(range<1>(1024), [=] (id<1> i){
   // CODE THAT RUNS ON DEVICE
});
```

```
h.parallel_for(range<1>(1024), [=] (id<1> i){
   auto idx = item.get_id();
   auto R = item.get_range();
    // CODE THAT RUNS ON DEVICE
});
```

ND-RANGE KERNELS

- Los *kernels* paralelos son una manera fácil de paralelizar un bucle *for*, pero no permiten la optimización del rendimiento a bajo nivel (explotación hardware)
- El ND-Range es otra forma de expresar el paralelismo que permite el ajuste del rendimiento de bajo nivel al proporcionar acceso a la memoria local y a la asignación de las unidades de proceso del hardware

- Todo el espacio de iteración se divide en grupos más pequeños denominados work-groups, los work-items se mapean en una sola unidad de proceso en el hardware
- La agrupación de work-items en work-groups permite controlar el uso de recursos y equilibrar la carga de las distribuciones de trabajo



ND-Range

ND-RANGE KERNELS

- La funcionalidad de nd_range del kernel se expresa con las clases nd_range y nd_item

```
h.parallel_for(nd_range<1> (range <1>(1024), range<1>(64) )), [=]
(nd_item<1> item){
   auto idx = item.get_id();
   auto R = item.get_range();
    // CODE THAT RUNS ON DEVICE
});
```

- nd_range representa la ejecución agrupada utilizando el rango de ejecución global y el rango de ejecución local de cada work-group
- nd_item representa la ejecución de una instancia individual del kernel y permite consultar el rango de work-group y su índice

- Modelo Memoria (buffer)

MODELO DE MEMORIA CON BUFFER

- Buffers: encapsula los datos accesibles por una aplicación SYCL
  - Permite la interacción de los datos desde los dispositivos y el host
- Accessors: mecanismo para acceder a los datos del buffer
  - Genera el grafo de dependencias que ordena las ejecuciones de los kernels

```
queue q;
std::vector<int> v(N, 10); {
   buffer buf(v);
   q.submit([&](handler& h){
      accesor a(buf, h, write_only);
      h.parallel_for(N, [=](auto i){a[i] = i;});
   });
}
for (int i=0; i<N; i++) std:cout << v[i] << " ";
```

Anatomía del Código DPC++

ANATOMÍA DEL CÓDIGO DPC++

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

- Los programas basados en oneAPI requieren la inclusión de la cabecera CL/sycl.hpp
  - En caso de usar FPGAs además
    - #include <CL/sycl/intel/fpga_extensions.hpp>
- Se útil añadir la declaración *namespace* para permitir las referencias a sycl

ANATOMÍA DEL CÓDIGO DPC++

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
int main(int argc, char *argv[]) {

  ...

}
```

- Cada ejecución en el dispositivo se asocia a una cola.
- Una cola se conecta a un único dispositivo (por ejemplo, GPU, FPGA, IA, CPU, host).

ANATOMÍA DEL CÓDIGO DPC++

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
int main(int argc, char *argv[]) {

 ...
 queue myQueue{...};

 ...

 myQueue.submit([&](handler &cgh) {

  // accessors (for connecting to memory via buffers)
  // kernel defined here (with lambda -
  // by value captures only)

 });


}
```

- La cola acepta solicitudes de trabajo
- Las líneas resaltadas son el ámbito del *command group*.
- Los envíos finalizan de forma asincrónica
- ¡Solo un kernel se ejecuta simultaneamente en cada cola!

ANATOMÍA DEL CÓDIGO DPC++

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
int main(int argc, char *argv[]) {

 ...
 queue myQueue{...};

 ...

 myQueue.submit([&](handler &cgh) {




 });



}
```

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

cgh.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)
});

cgh.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)
});

cgh.parallel_for_work_group(
  range<2>(1024,1024),      // using 2D in this example
  [=](group<2> myGroup) {
    // 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
});
```

ANATOMÍA DEL CÓDIGO DPC++

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
int main(int argc, char *argv[]) {

 // define buffers!!!
 queue myQueue{...};

 ...

 myQueue.submit([&](handler &cgh) {

  // accessors (for connecting to memory via buffers)
  // kernel defined here (with lambda -
  // by value captures only)

 });



}
```

- El ámbito de los búferes fija la sincronización con la ejecución del host

ANATOMÍA DEL CÓDIGO DPC++

```
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;
}
```

Host code

Accelerator device code

Host code

UN PROGRAMA COMPLETO DPC++

Código host y device (mismo fichero)

- El código del host y los kernels (computación heterogénea) se pueden mezclar en los mismos archivos

Sintaxis C++

- Las construcciones agregan funcionalidad, como:

Construcción	Propósito
queue	Código objetivo
buffer	Manejo datos
parallel_for	Parallelismo

```
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;
}
```

Host code

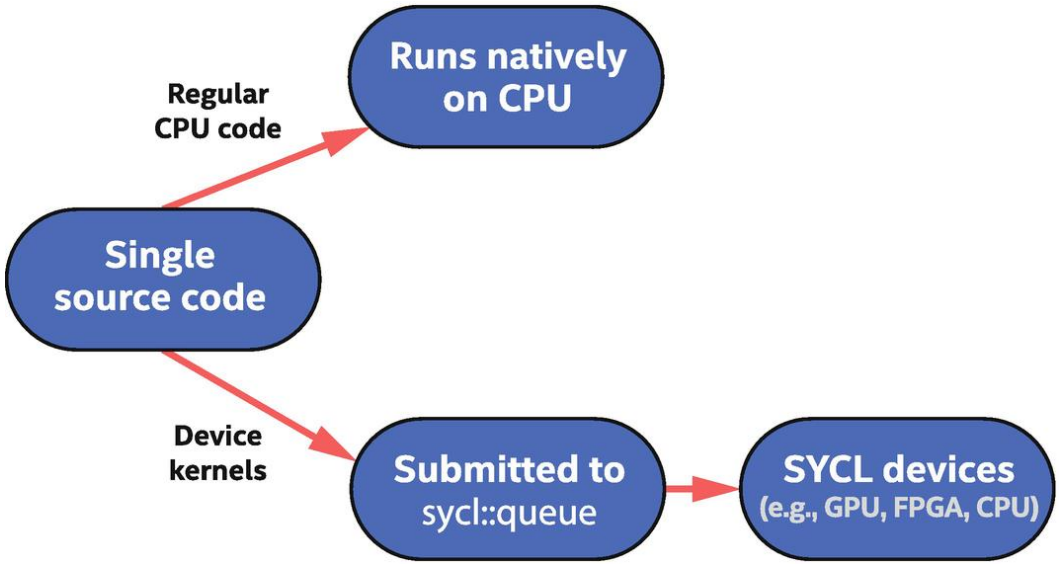
Accelerator device code

Host code

- Modelo ejecución

¿DÓNDE SE EJECUTA?

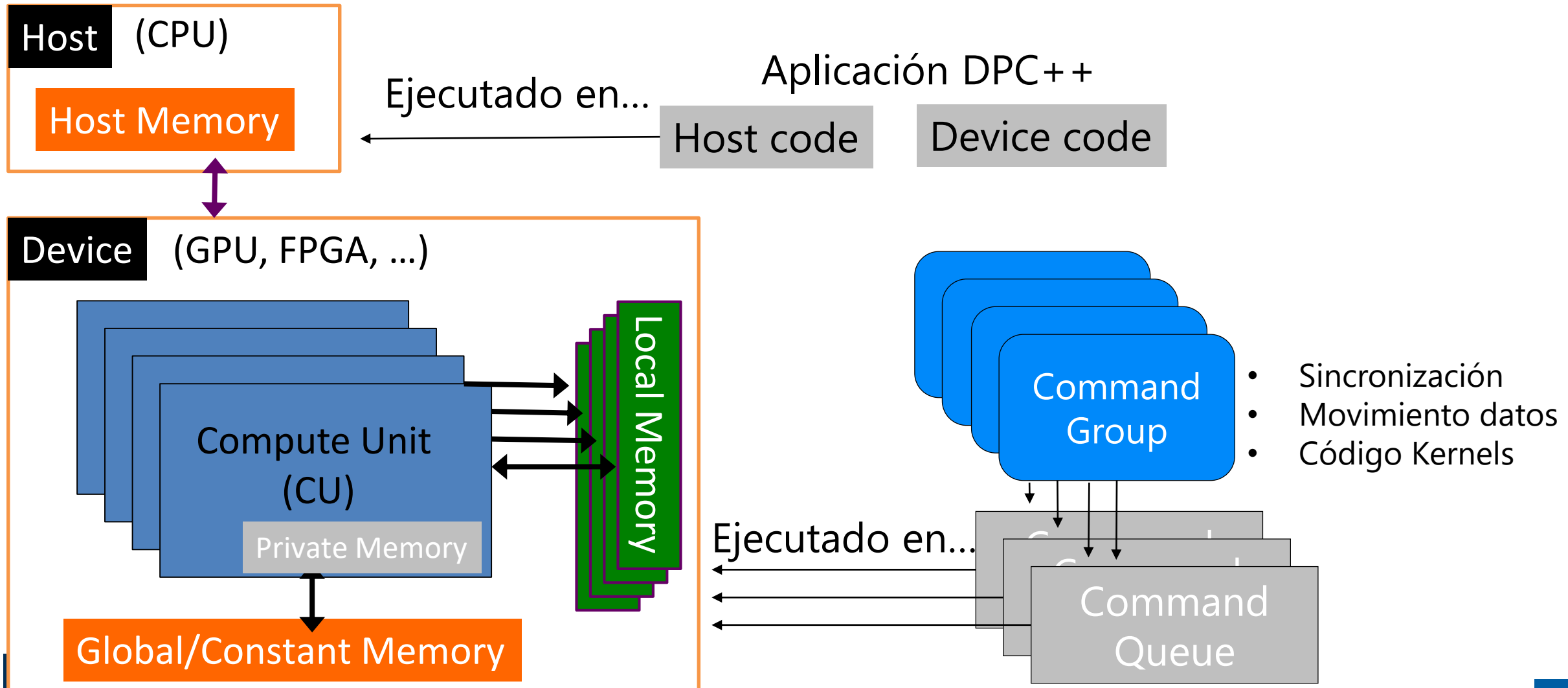
- Código fuente en un único fichero
  - El código de los *kernels* define el trabajo a descargar
  - Combinación de código de dispositivo y host en el mismo fichero fuente



¿DÓNDE SE EJECUTA?

- Código de host
  - Ejecutado en CPU(s)
- Dispositivo
  - Colas: mecanismo para enviar a un dispositivo
    - Asíncrono desde el punto de vista del host
  - Elegir dispositivos
    - Ejecutar el código "en algún lugar" → no importa el dispositivo
    - Explícitamente (host, gpu, fpga...)

¿DÓNDE SE EJECUTA? MODELO EJECUCIÓN



¿DÓNDE SE EJECUTA? SELECIÓN DISPOSITIVO

Dispositivo (cualquiera):	`queue q (); // default_selector{}`

Clases de dispositivo:	`queue q(gpu_selector{});`
	`queue q(accelerator_selector{});`
	`queue q(cpu_selector{});`
	`queue q(host_selector{});`

Selector personalizado:	`class custom_selector : public device_selector {`
	`  int operator()(……`
	`…`
	`queue q(custom_selector{});`

default_selector

- El runtime de DPC++ puntúa los dispositivos disponibles y seleciona el que tiene ranking mayor
- Selección por variable de entorno

export SYCL_DEVICE_TYPE=GPU | CPU | HOST

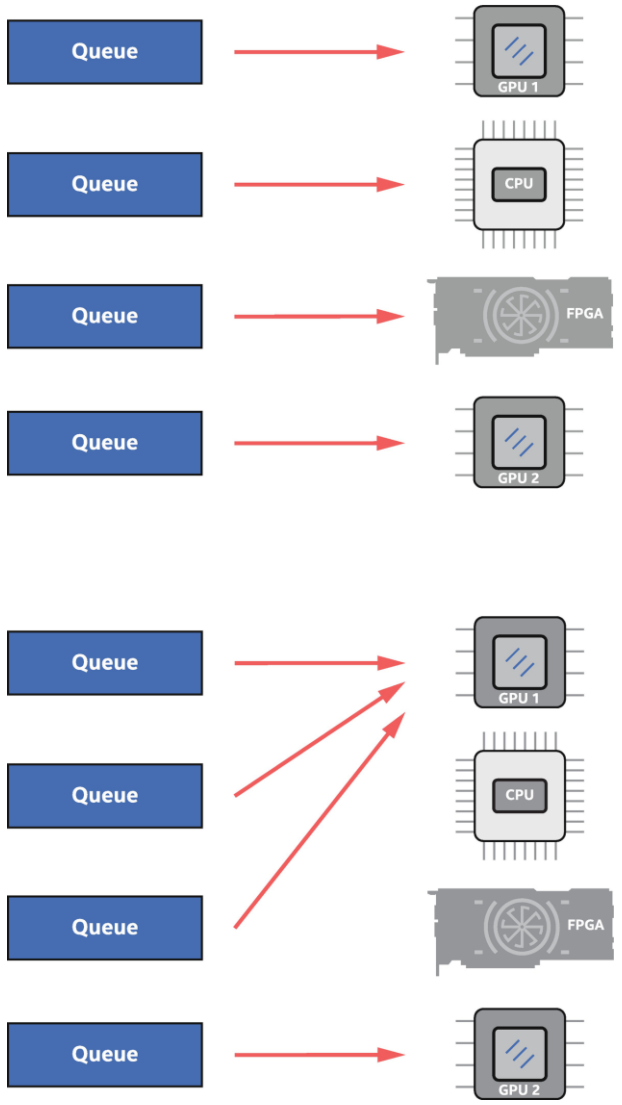
¿DÓNDE SE EJECUTA?

```
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(gpu_selector{});
        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;
}
```

¿DÓNDE SE EJECUTA?

Colas

- Un programa puede crear varias colas
- Una cola no puede enlazar con más de un dispositivo
- Múltiples colas pueden enlazar a un solo dispositivo
- El dispositivo host siempre está disponible: *Debug*



¿DÓNDE SE EJECUTA?

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

int main() {
        // Create queue to use the CPU device explicitly
        queue Q{ cpu_selector{} };
        std::cout << "Selected device: " <<
                Q.get_device().get_info<info::device::name>() << "\n";
        std::cout << " -> Device vendor: " <<
                Q.get_device().get_info<info::device::vendor>() << "\n";
        return 0;
}

Possible Output:
Selected device: Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz
-> Device vendor: Intel(R) Corporation
```

¿DÓNDE SE EJECUTA?

```
#include <CL/sycl.hpp>
#include <CL/sycl/INTEL/fpga_extensions.hpp>// For fpga_se
#include <iostream>
#include <string>
using namespace sycl;

void output_dev_info( const device& dev, const std::string
          std::cout << selector_name << ": Selected device
                    dev.get_info<info::device::name>() <<
          std::cout << " -> Device vendor: " <<
                    dev.get_info<info::device::vendor>() <
}

int main() {
          output_dev_info( device{ default_selector{}}, "d
          output_dev_info( device{ host_selector{}}, "host
          output_dev_info( device{ cpu_selector{}}, "cpu_s
          output_dev_info( device{ gpu_selector{}}, "gpu_s
          output_dev_info( device{ accelerator_selector{}}, "accelerat selector"
);
          output_dev_info( device{ INTEL::fpga_selector{}}, "fpga_selec
          return 0;
}
```

Possible Output:

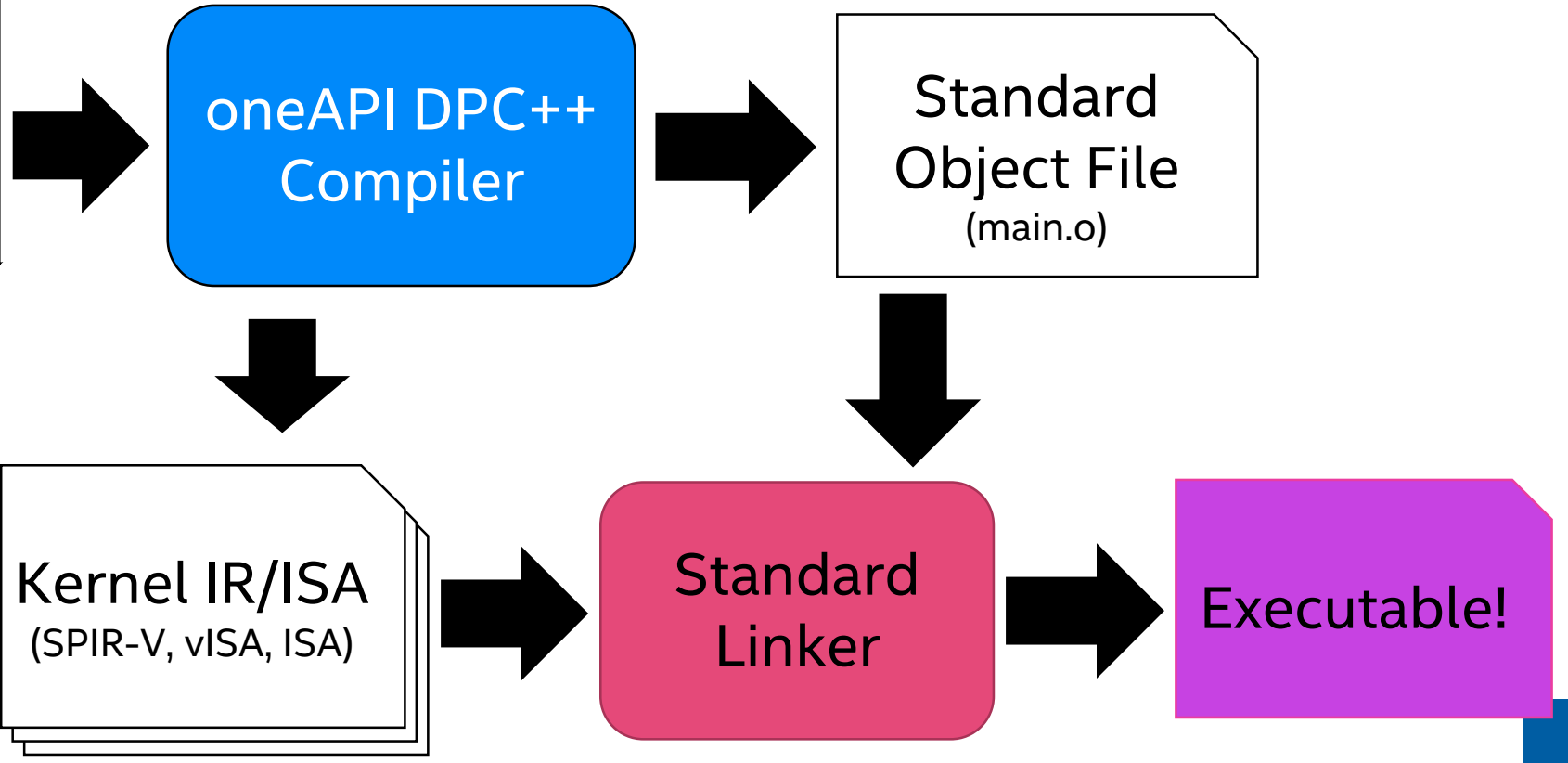
```
default_selector: Selected device: Intel(R) Gen9 HD Graphics NEO
          -> Device vendor: Intel(R) Corporation
host_selector: Selected device: SYCL host device
          -> Device vendor:
cpu_selector: Selected device: Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz
          -> Device vendor: Intel(R) Corporation
gpu_selector: Selected device: Intel(R) Gen9 HD Graphics NEO
          -> Device vendor: Intel(R) Corporation
accelerator_selector: Selected device: Intel(R) FPGA Emulation Device
          -> Device vendor: Intel(R) Corporation
fpga_selector: Selected device: pac_a10 : PAC Arria 10 Platform
          -> Device vendor: Intel Corp
```

FLUJO DE COMPILACIÓN

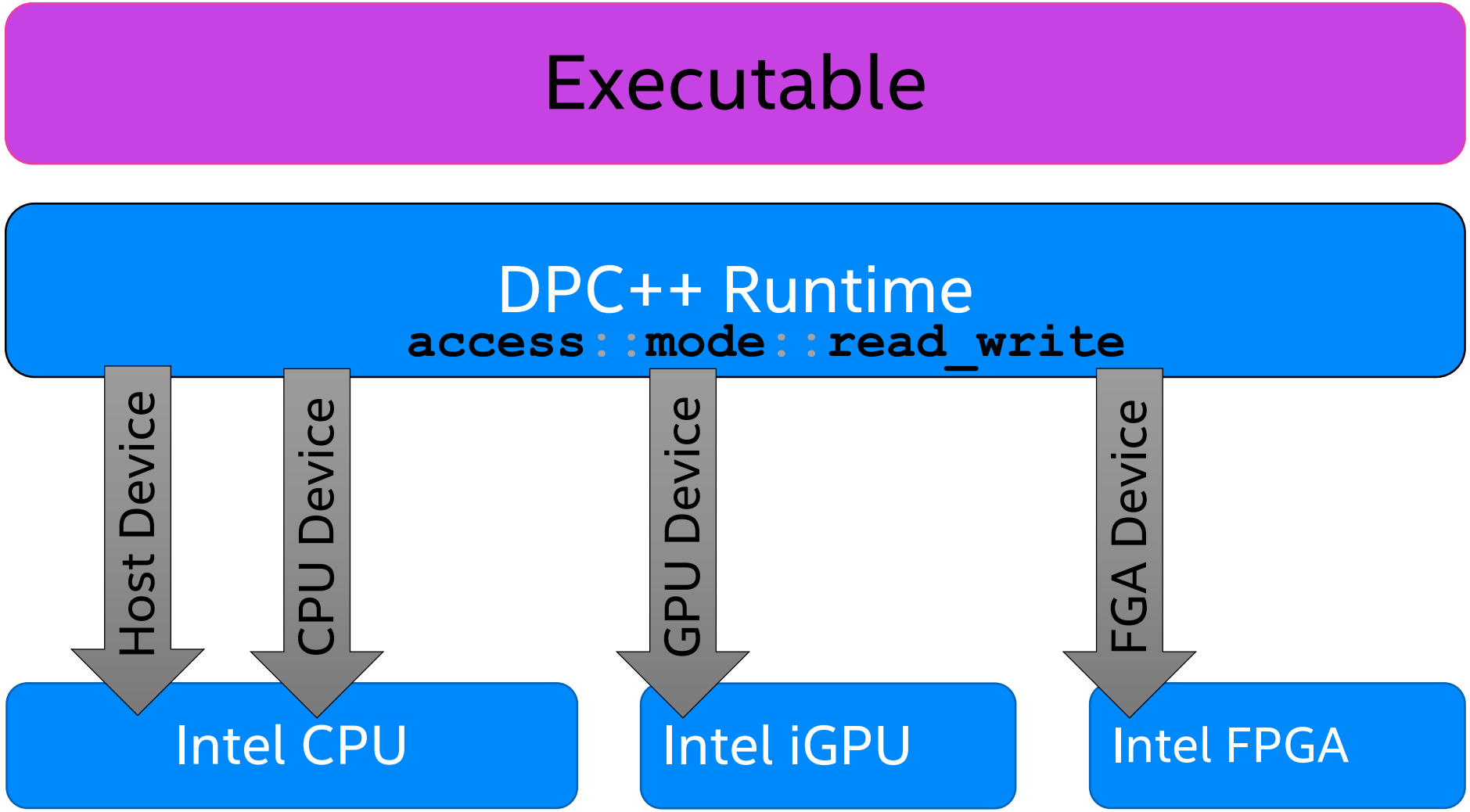
main.cpp:

```
#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;
}
```

dpcpp main.cpp

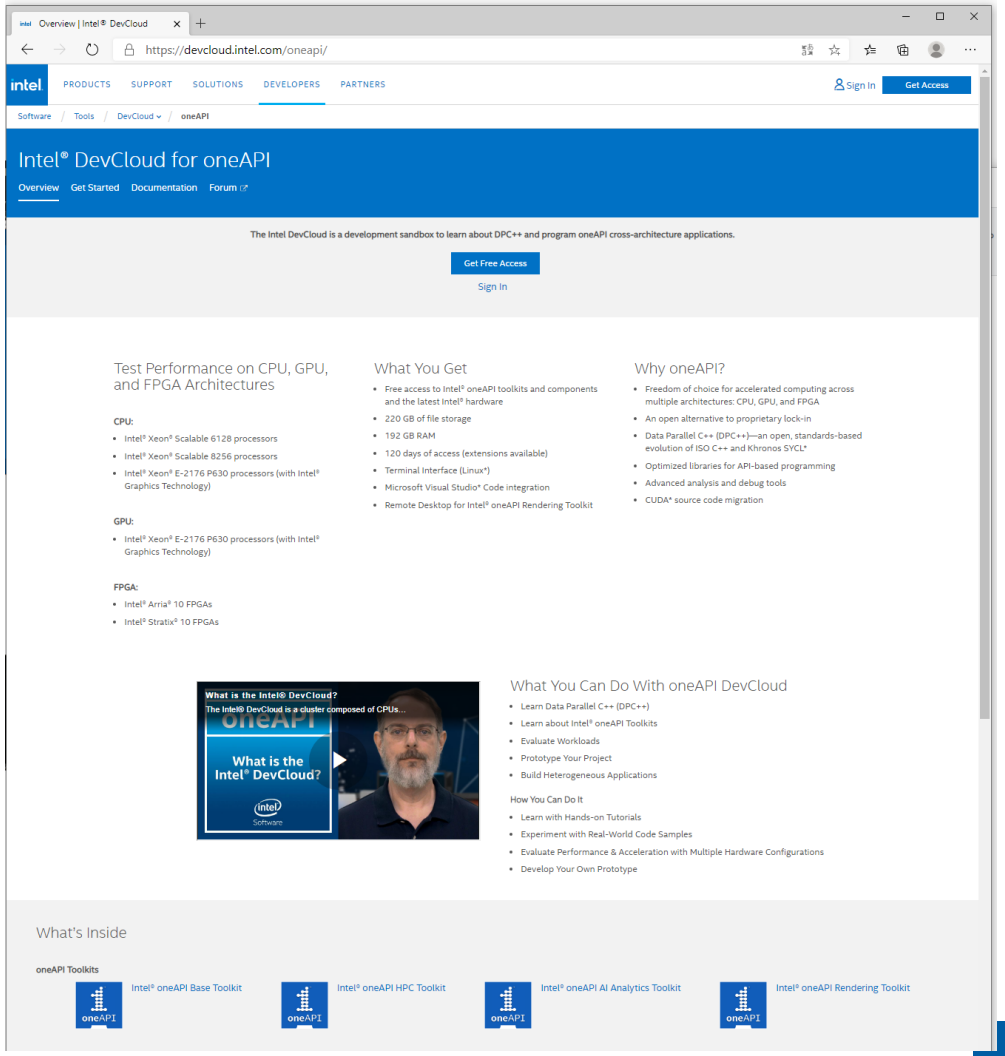


FLUJO DE EJECUCIÓN



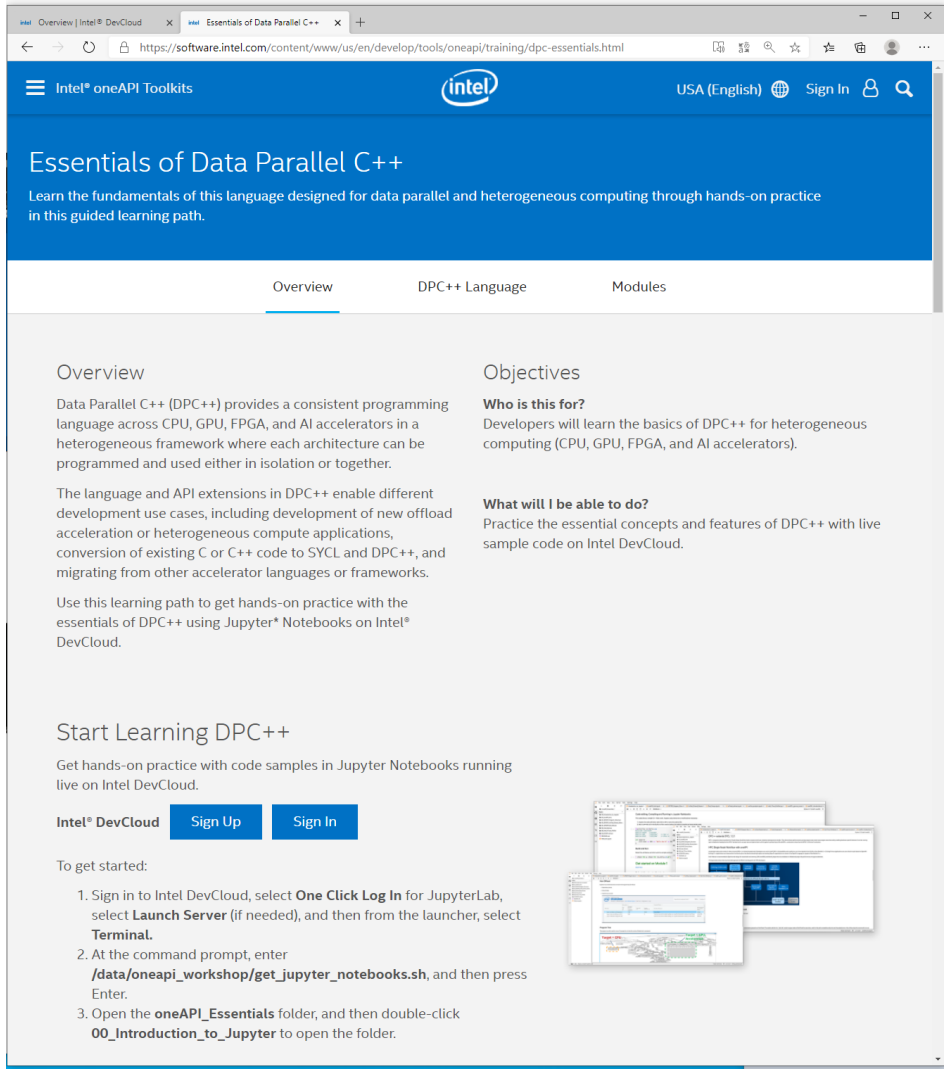
HANDS ON CODING

- Acceder a Intel DevCloud
  - CPU:
    - Intel® Xeon® Scalable 6128
    - Intel® Xeon® Scalable 8256
      Intel® Xeon® E-2176 P630
  - GPU:
    - Intel® Xeon® E-2176 P630 (con Intel® Graphics)
  - FPGA:
    - Intel® Arria® 10 FPGAs
    - Intel® Stratix® 10 FPGAs



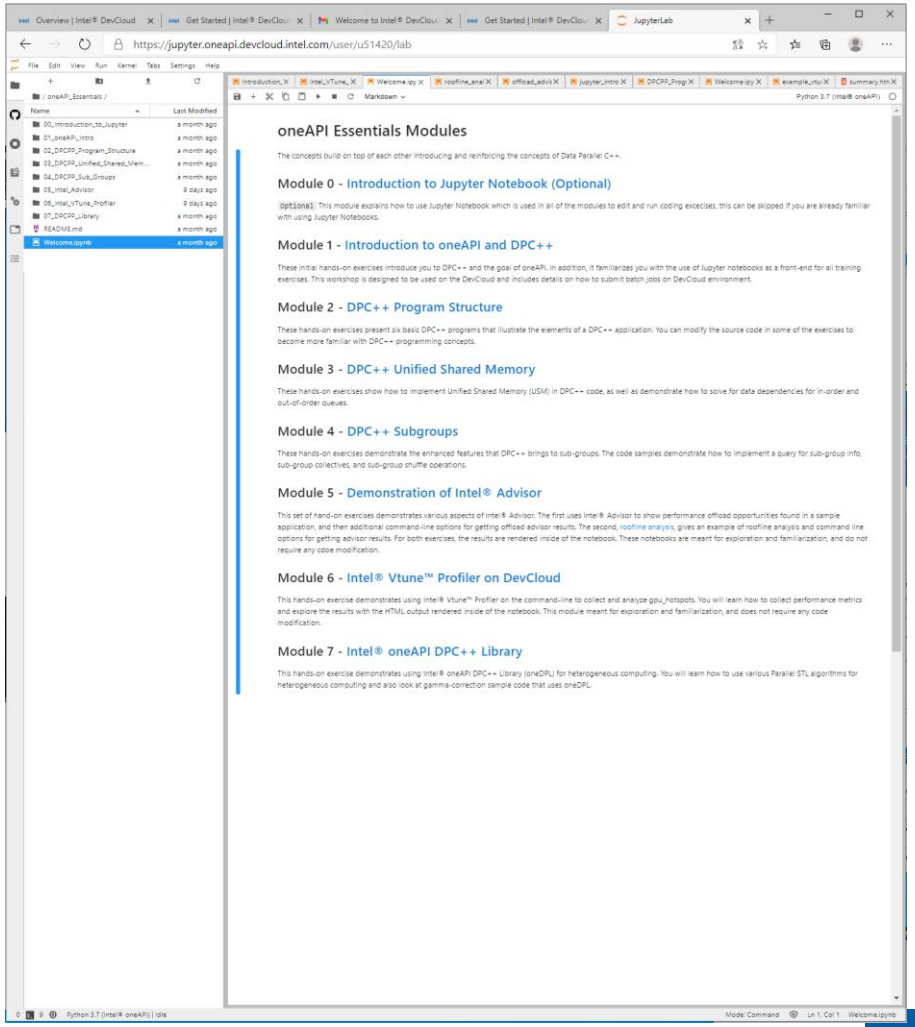
HANDS ON CODING

- Acceder a Intel DevCloud
- Jupyter Notebook: Essentials of Data Parallel C++
  - 00: Introduction to JupyterLab and Notebooks
  - 01: Introduction to DPC++
  - 02: DPCPP Program Structure



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 ONEAPI

- To start on oneAPI Base Toolkit
  - oneAPI CLI Samples
    1. Open Terminal
    2. Environment variables:

```
>> source /opt/intel/oneapi/setvars.sh
```

    3. Invoke oneAPI client with examples

```
>> oneapi-cli
```

HANDS ON ONEAPI

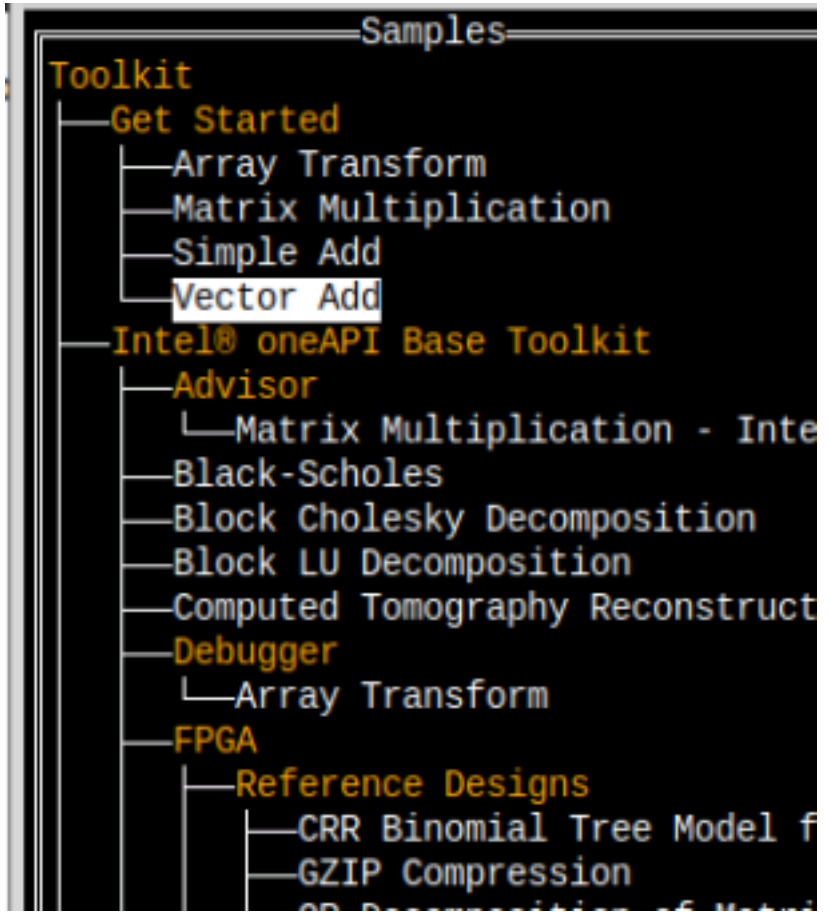
- Ejemplos en el oneAPI Base Toolkit
  - oneAPI CLI Samples
    1. Abrir terminal
    2. Environment variables:

```
>> source /opt/intel/oneapi/setvars.sh
```

    3. Invocar el cliente de los ejemplos:

```
>> oneapi-cli
```

    4. Seleccionar un ejemplo
    5. Descargarlo
    6. Compilar y ejecutar



HANDS ON ONEAPI

- Job Submission en Intel-DevCloud
  - Submit un batch job

```
>> qsub -l nodes=1:gpu:ppn=2 -d . job.sh
```

  - Crear un trabajo interactivo

```
>> qsub -I -l nodes=1:gpu:ppn=2 -d .
```

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.

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

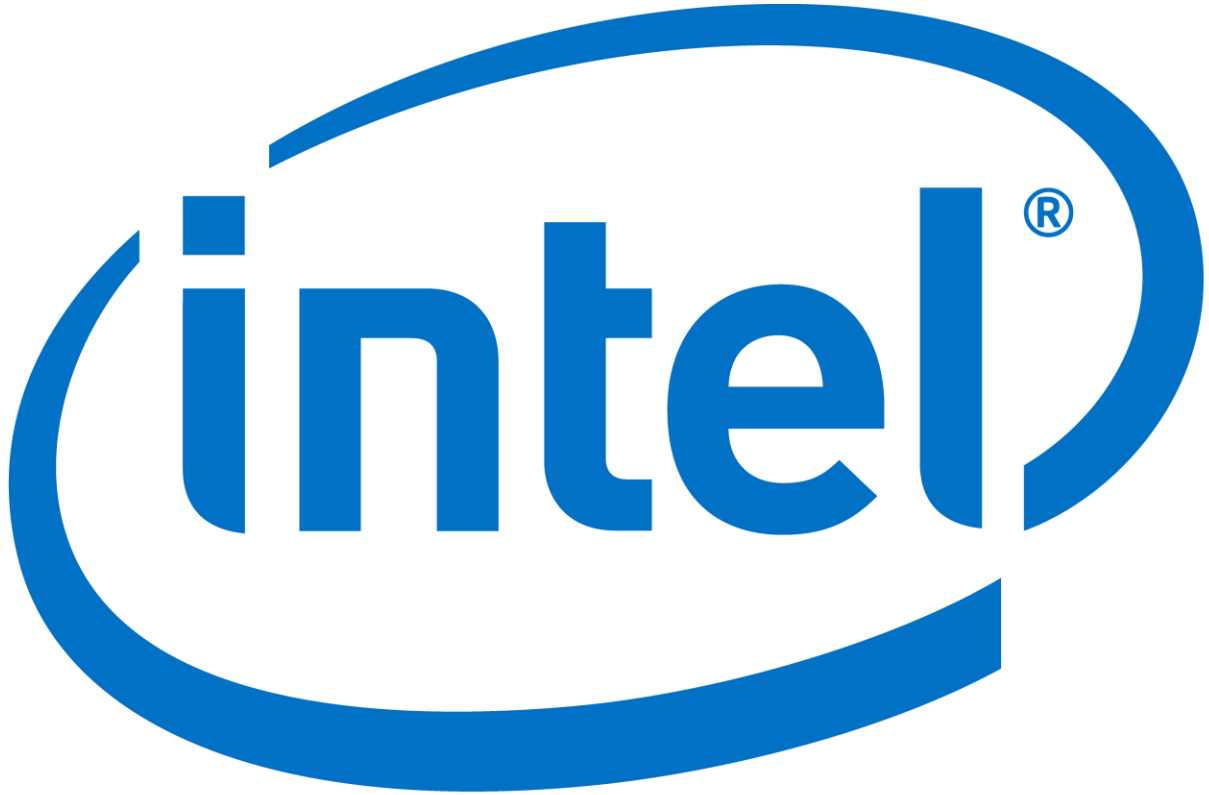
  oneAPI Programming Guide
- DPC++ book 4 preview chapters

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