DCP++ y FPGAs

Máster en Ingeniería Informática 2022-23

Alonso Núñez, Mario

Índice de contenidos

Introducción a DPC++

Situación actual

Crecimiento de las cargas de trabajo especializadas

Aumento en la variedad de hardware utilizada

La inexistencia de una API de programación común

La utilización de herramientas especializadas para cada plataforma

La necesidad de adaptar el software para cada plataforma de manera independiente

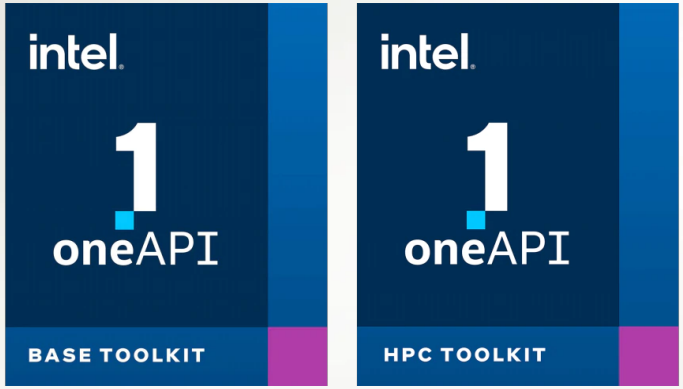
¿Qué es OneApi?

OneAPI es una iniciativa industrial aportada por Intel, la cual pretende dar solución a la problemática actual.

Se puede definir como un conjunto de dos aspectos:

- ***Un standar abierto aceptado por gran parte de la industria.***
- ***Un producto de Intel para programación Heterogénea.***

Se encuentra organizado por sets y proporciona tanto bibliotecas de programación como herramientas de análisis.



DPC++ y SYCL

Data Parallel C++ es el lenguaje principal utilizado por oneAPI.

Esta formado en base al lenguaje C++ y contiene una capa superior de implementación del estándar SYCL.

Contiene librerías y extensiones para aumentar su alcance.

Pj. Traducir códigos de otros lenguajes.

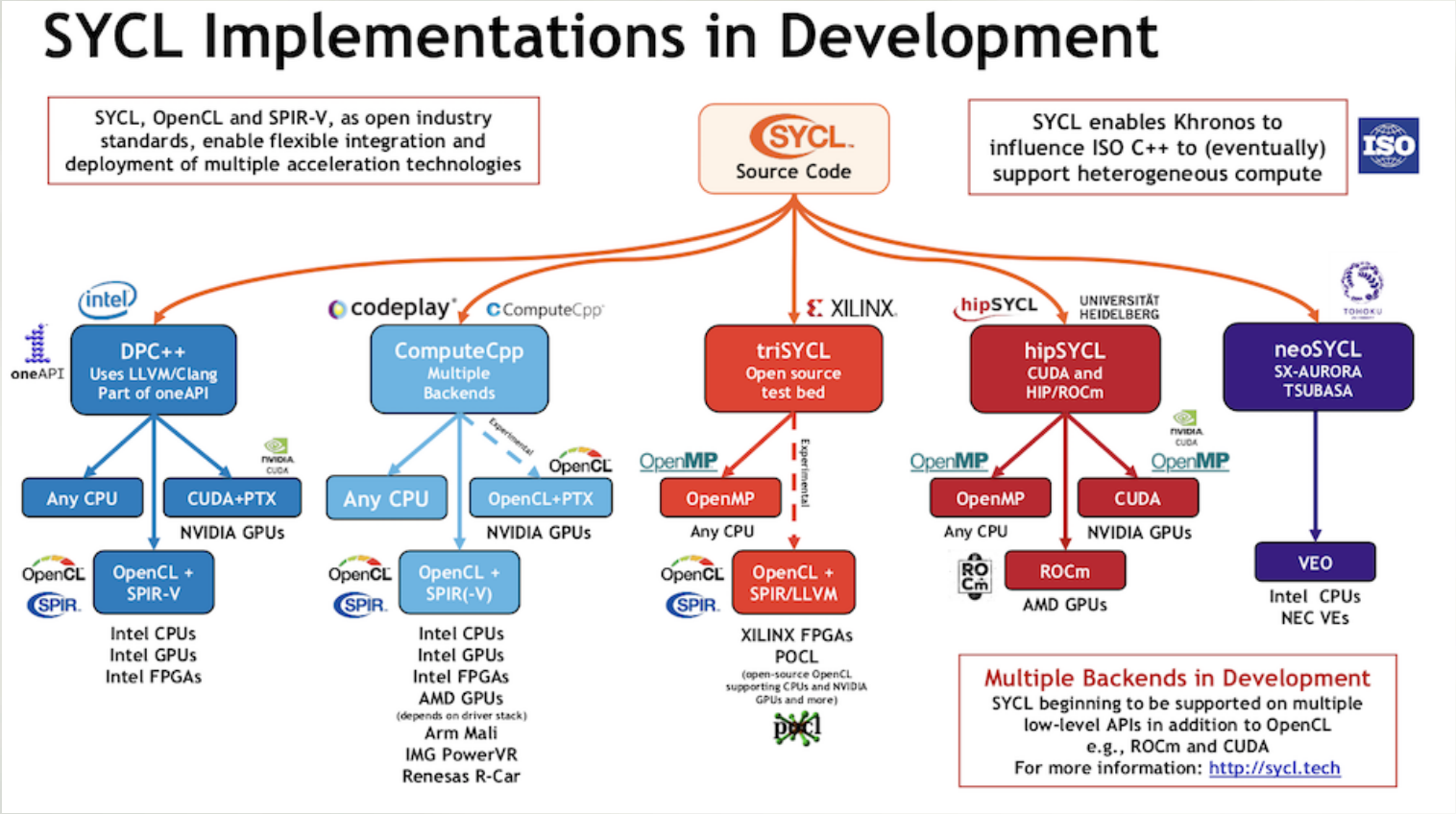
El standar SYCL

SYCL es una propuesta de estandarización con el objetivo de definir un modelo de programación heterogénea y centrada en el paralelismo de datos.

Es definido y soportado por el grupo Kronos, en el cual participa Intel.

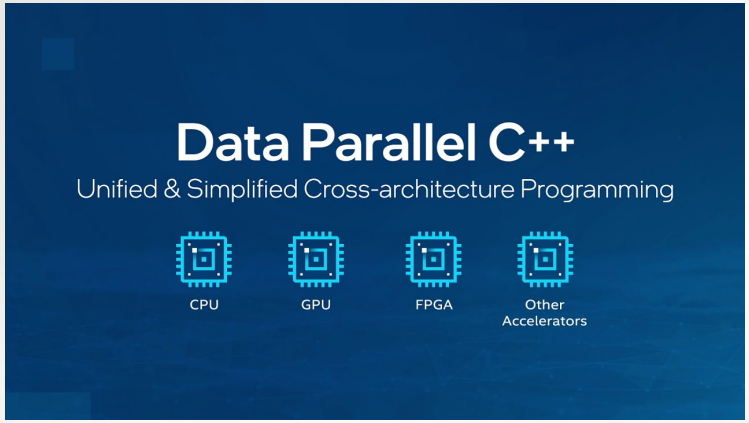
La gran mayoría de las características de DPC++ están definidas en el estándar SYCL.

Proyectos que implementan de SYCL



Puntos fuertes de DPC++

- ***Lenguaje de alto nivel que proporciona una programación paralela y heterogénea con pocas lineas de código.***
- ***Gran rendimiento sin un gran coste de programación.***
- ***Portabilidad hacia gran cantidad de arquitecturas y sistemas, sobre todo CPUs, GPUs y FPGAs.***
- ***Nos permite bajar de nivel para explotar las características de un hardware concreto.***
- ***Colaboración abierta que mejora el estándar SYCL.***



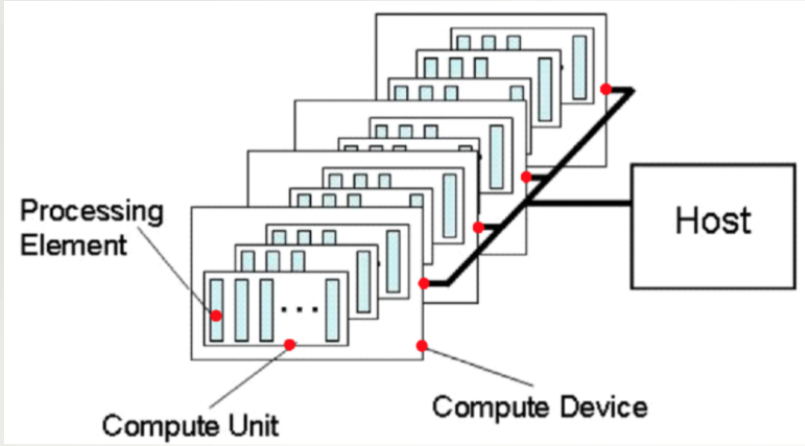
Funcionamiento de DPC++

Uso de Kernels

DPC++ utiliza un modelo de programación Host-Device basado en el uso de kernels.

Los kernels permiten la ejecución de las operaciones que contienen en un dispositivo del tipo indicado.

Podemos utilizar varios tipos de kernels, tanto para ejecuciones paralelas como secuenciales.



Funcionalidad de los Kernels

La funcionalidad de los kernels se expresa mediante clases:

- ***Range: Describe el espacio abarcado por la iteración del kernel, el cual puede indicarse mediante dimensiones.***
- ***Id: Permite indexar una instancia individual del kernel en la ejecución paralela.***
- ***Item: Referencia a una instancia individual del kernel.***

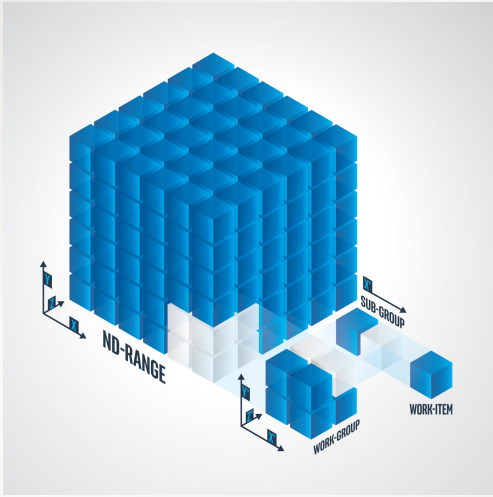
```
//# Offload parallel computation to device
q.parallel_for(range<1>(N), [=] (id<1> i){
  a[i] = b[i] + c[i];
}).wait();
```

El ND-Range

El uso del ND-Range nos permite ajustar el rendimiento a bajo nivel mediante la gestión del acceso y asignación de las unidades de computo del dispositivo.

El espacio de iteraciones esta compuesto por unidades de trabajo denominadas work-items, las cuales se agrupan en grupos llamados work-groups.

Podemos controlar la asignación de los recursos a los distintos work-groups, evitando desequilibrios.



Funcionalidad del ND-Range

La funcionalidad del ND-Range del kernel se expresa mediante el uso de dos clases:

- ***Nd_range: Representa la ejecución de toda la carga de trabajo y utiliza el rango de ejecución global y el rango de ejecución de cada work-group.***
- ***Nd_item: Representa la ejecución de una instancia individual del kernel.***

```
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();
});
```

Modelo de memoria

Al trabajar en un modelo Host-Device no podremos operar con referencias, sino que necesitamos proporcionar los datos al dispositivo. Para hacer esto utilizaremos dos elementos:

- ***Buffers**: Encapsulan los datos para que puedan ser intercambiados entre el host y el device.*
- ***Accessors**: Contienen la referencia a la estructura de los datos dentro del buffer y permiten el acceso a los mismos.*

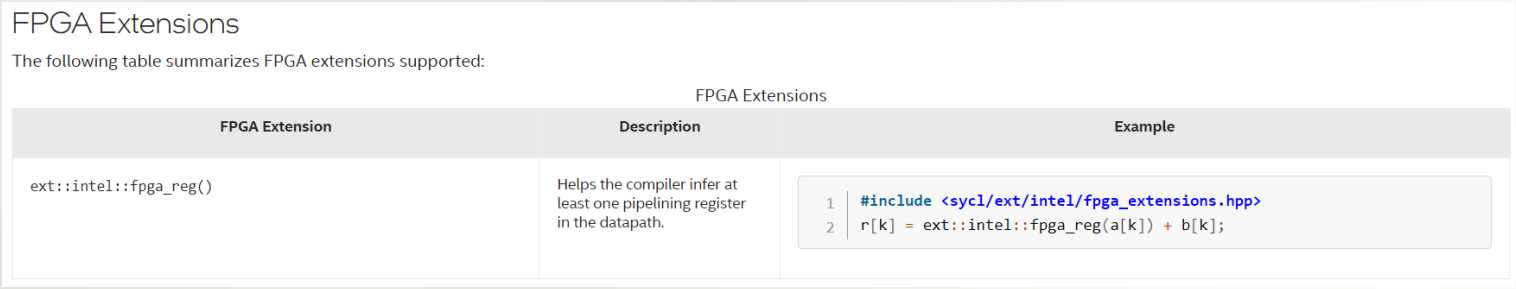
```
queue q;
std::vector<int> v(N, 10); {
    buffer buf(v);
    q.submit([&](handler& h){
        accesor a(buf, h, write_only);
        h.parallel_for(N, [=](auto i){a[i] = i;});
    });
}
```

Programación en DPC++

Inclusión de cabeceras

Para programar mediante DPC++ es necesario la utilización de las librerías incluidas dentro de el kit básico de OneAPI.

Para la utilización de dichas librerías será necesario su importación mediante la cabeceras, siendo la principal <CL/sycl.hpp>.



FPGA Extensions

The following table summarizes FPGA extensions supported:

FPGA Extensions

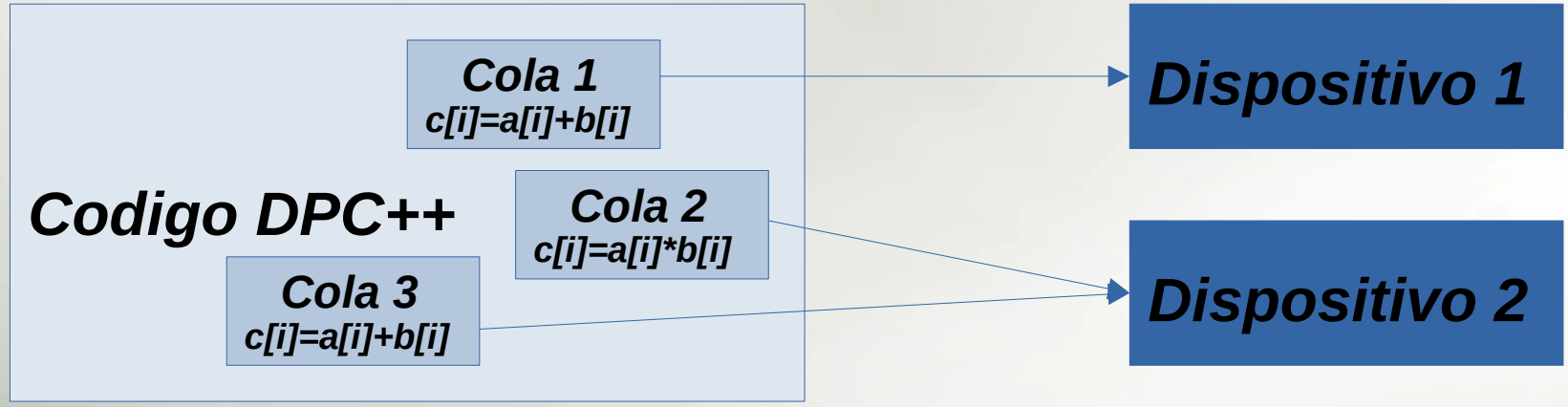
FPGA Extension	Description	Example
`ext::intel::fpga_reg()`	Helps the compiler infer at least one pipelining register in the datapath.	`#include <sycl/ext/intel/fpga_extensions.hpp>` `r[k] = ext::intel::fpga_reg(a[k]) + b[k];`

Uso de las colas

Para poder lanzar la ejecución de un kernel sobre un dispositivo necesitaremos utilizar colas.

Las colas constituyen el mecanismo que asocia el código del host con el dispositivo donde de quiere ejecutar.

Una cola solo puede asociarse a un dispositivo, pero un mismo dispositivo puede tener asociadas más de una cola.



Definición de colas

Dentro de las colas deberemos especificar los elementos necesarios para el tratamiento con el dispositivo:

- ***El buffer donde almacenar los datos.***
- ***Los accessor para el tratamiento de los datos.***
- ***El kernel a ejecutar.***

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

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

cgh.parallel_for_work_group( range<2>(1024,1024), [=](group<2> myGroup) {
    // kernel function is executed once per work-group
});

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

Interdependencia entre colas

Cuando enviamos una cola a un dispositivo, la ejecución de su kernel se realiza de forma asíncrona.

Esto provoca que se utilicen mecanismos de sincronización entre el host y el dispositivo, en el caso de necesitar utilizar los datos procesados.

Los buffers están ligados a un ámbito, de modo que cuando este finaliza, los buffers son destruidos.

Ejemplo de colas

```
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;

}
```

Mapeado de datos

Creación de la cola

Ámbito del buffers

Definición de accesos y su tipo

Ámbito del kernel

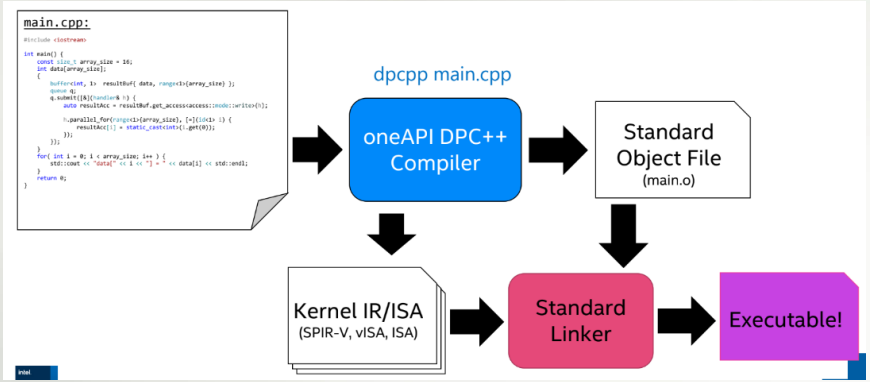
Modelo de ejecución de DPC++

Generación del ejecutable

DPC++ tiene la ventaja de tener tanto el código del host como el código del dispositivo agrupados en un mismo fichero.

Al compilar el código, se generan los diferentes ficheros para cada uno de los potenciales dispositivos a utilizar.

Finalmente se crea un ejecutable unificado, de modo que se elegirá en tiempo de ejecución, los dispositivos a utilizar.



Elección del dispositivo

La elección del dispositivo se realizará en tiempo de ejecución, lo cual es necesario para mantener la portabilidad.

El runtime siempre elegirá el Dispositivo más conveniente para la ejecución de un kernel en concreto, pero podemos indicarle nuestra inclinación hacia un tipo u otro.

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{});`

```
int main() {
    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;
}
```

```
## u180325 is compiling DPCPP_Essentials Module1 -- oneAPI Intro sample - 1 of 1 simple.cpp
Selected device: 11th Gen Intel(R) Core(TM) i9-11900KB @ 3.30GHz
 -> Device vendor: Intel(R) Corporation
```

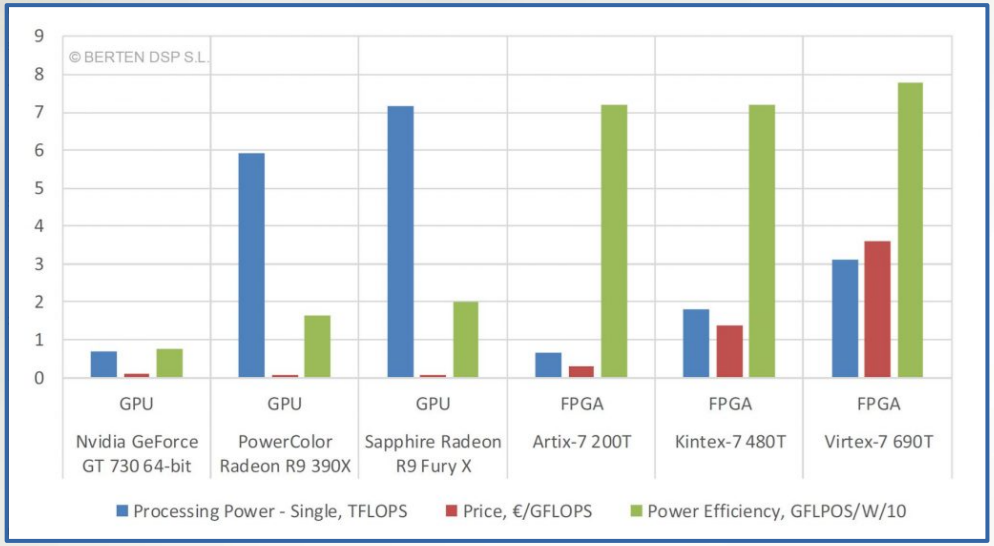
El uso de las FPGA

FPGAs en altas prestaciones

Actualmente, los sistemas pensado para computo masivo utilizan aceleradores de computación paralela para obtener un mayor rendimiento en sus operaciones.

El uso de estos dispositivos conlleva un alto consumo energético, sin embargo, las FPGAs necesitan una cantidad mucho menor de energía.

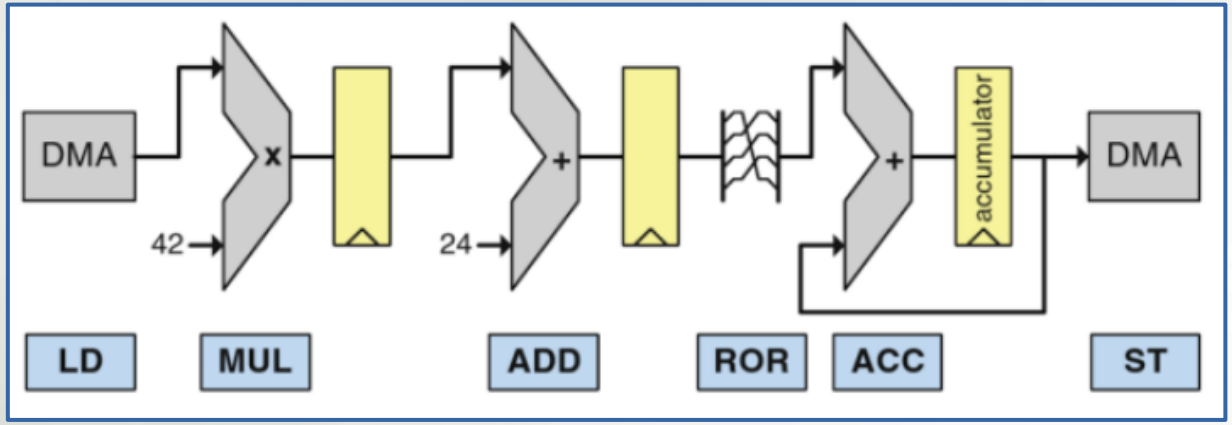
Empiezan a surgir servidores de computo masivo basados en el uso de FPGAs.



Paralelismo en FPGA

El potencial de las FPGAs no reside sobre múltiples núcleos, sino en la versatilidad de uso de sus componentes.

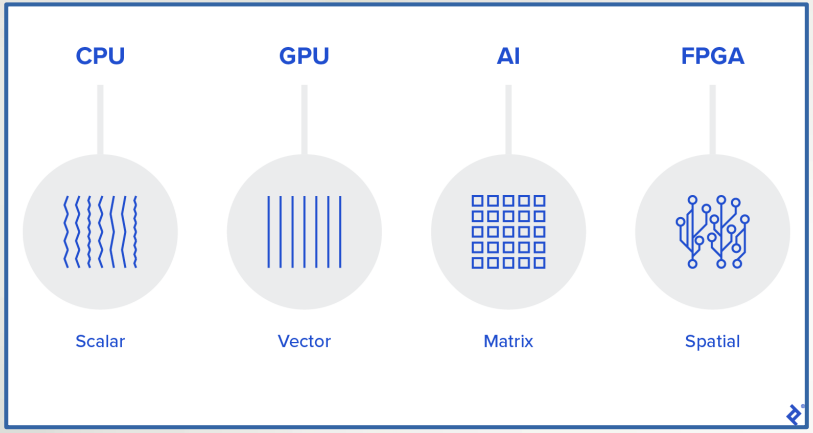
Explotar el rendimiento de una FPGA se apoya sobre el ILP, gracias a poder modificar las conexiones entre las fases, hasta el punto de poder eliminarlas por completo.



Intel y las FPGAs

La apuesta de Intel sobre la generación de una API única para cualquier tipo de arquitectura conlleva aumentar el soporte de la misma hacia nuevos tipos de dispositivos.

OneAPI proporciona soporte total sobre dispositivos CPU, GPY y FPGA, además de traductores que acomodan código de otros dispositivos específicos, como CUDA.



Ejemplo programación FPGA

```
sycl::INTEL::fpga_selector device_selector;

queue_event = device_queue.submit([&](sycl::handler &cgh) {
    auto _pixels = pixels_buf.get_access<sycl::access::mode::read>(cgh);
    auto _accumulators = accumulators_buf.get_access<sycl::access::mode::read_write>(cgh);

    cgh.single_task<class Hough_transform_kernel>([=]() {
      for (uint y=0; y<HEIGHT; y++) {
        for (uint x=0; x<WIDTH; x++){
          unsigned short int increment = 0;
          if (_pixels[(WIDTH*y)+x] != 0) {
            increment = 1;
          } else {
            increment = 0;
          }
          for (int theta=0; theta<THETAS; theta++){
            int rho = x*_cos_table[theta] + y*_sin_table[theta];
            _accumulators[(THETAS*(rho+RHOS))+theta] += increment;
          }
        }
      }

    });

});
```

Indicación de la FPGA como dispositivo

Definición de accesos y su tipo

Ejecución de forma secuencial

Ámbito del kernel

Bibliografía

- *Taller: oneAPI como Plataforma de Desarrollo en FPGAs*
- *Taller: Compiladores C/C++ y Fortran - Herramienta Data Parallel C++*
- *Taller: Ventajas de uso de DPC++*
- *Recursos OneApi Intel DevCloud*
- *Recursos Jupiter - oneAPI essentials*
- *Recursos Jupiter - oneAPI FPGAs*