Aceleración por Hardware: GPU y FPGA

PEDRO IGNACIO MARTOS (PMARTOS@FI.UBA.AR)

Temario

- **Introducción**
- GPU
  - CUDA
  - OpenCL
- FPGA – High Level Synthesis (HLS)
  - Vivado HLS – IP integrable (Orientado a Hardware)
  - Vitis HLS – Generación de Kernels en lógica programable (Orientado a Software)

Introducción

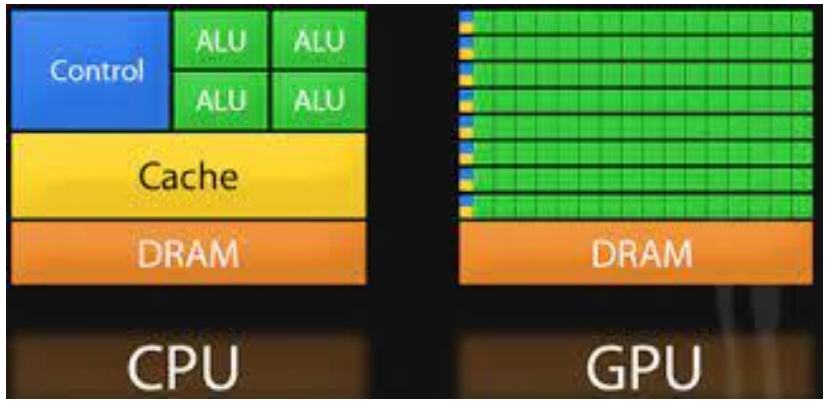
- La aceleración por hardware consiste en implementar todo o parte de un algoritmo de procesamiento de señales en un dispositivo de cómputo optimizado para su ejecución.
- Este concepto surge como mejora a la implementación de algoritmos en dispositivos de cómputo de propósito general, tales como los procesadores.
- Inicialmente se utilizaba hardware específicamente diseñado (Aplication Specific Integrated Circuits – ASICs), actualmente se emplean GPUs y FPGAs.
- Si bien hay distintas implementaciones de este concepto (CUDA, OpenCL, High Level Synthesis, etc), en todas se propone un modelo de dispositivo de cómputo abstracto (Hardware Abstraction Layer – HAL) que enmascara las diferencias entre los distintos hardware sobre los que funciona la implementación; y una interfaz de software (API) que unifica la forma de acceder a ese hardware.

Temario

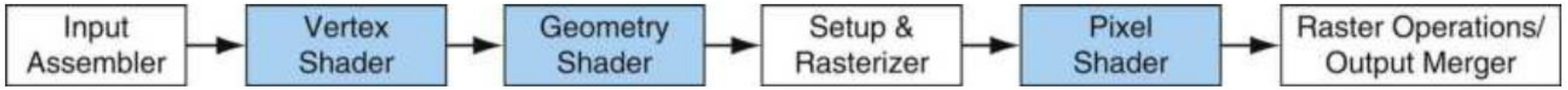
- Introducción
- **GPU**
  - CUDA
  - OpenCL
- FPGA – High Level Synthesis (HLS)
  - Vivado HLS – IP integrable (Orientado a Hardware)
  - Vitis HLS – Generación de Kernels en lógica programable (Orientado a Software)

GPU

- Los dispositivos de cómputo tipo GPU (Graphics Processing Unit) surgen como hardware dedicado y optimizado para aplicaciones gráficas (principalmente videojuegos).
- Estos dispositivos consisten en una cantidad muy grande de unidades de cómputo de arquitectura simple que trabajan en paralelo, a esto se lo denomina “Processor Array”

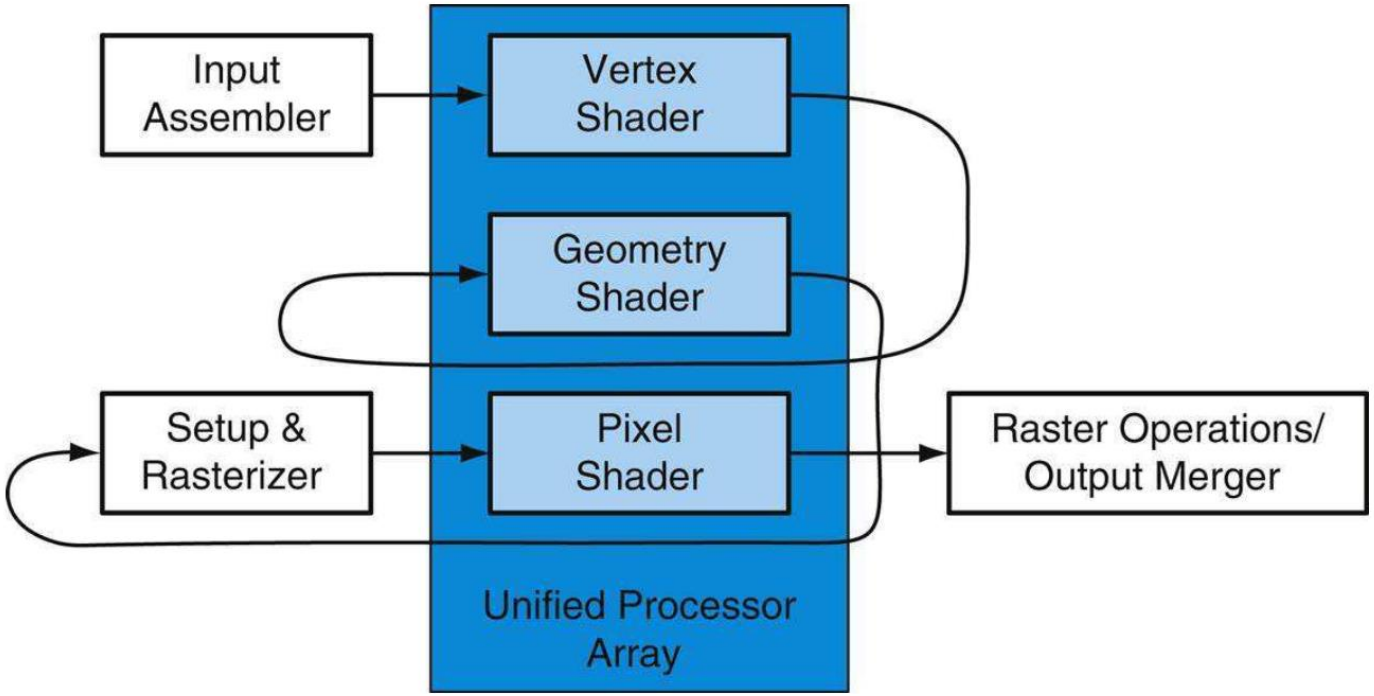


- Su arquitectura esta optimizada para procesar la proyección 2D de imágenes 3D (computo de Vertice, Geometría y Pixel)



GPU

- Esta arquitectura puede aprovecharse para realizar cómputo general



GPU – Comparación con CPU

- Las GPU son aceleradores que suplementan a las CPU, por lo que no necesitan ser capaces de realizar todas las tareas que realiza una CPU. De esta manera hay tareas que pueden hacer de forma poco eficiente o no realizarlas, dado que en un sistema CPU+GPU, el CPU puede realizarlas.
- La GPU no utiliza varios niveles de cache para ocultar la latencia de acceso a memoria, sino que realiza hardware multithreading; es decir, desde el momento en que se hace el acceso a memoria hasta que la información está disponible, la GPU ejecuta otros procesos de cómputo que son independientes de este acceso.
- La memoria de la GPU está orientada al mayor ancho de banda, es normal utilizar memorias con buses de datos de 256 o 512 bits de tamaño.
- Los procesadores GPU pueden realizar multithreading a mucha mayor escala que un CPU. Mientras que un CPU pueden manejar decenas de threads simultáneamente (16/32/64), un GPU puede manejar miles de threads simultáneamente (1024/4096/8192)

GPU – Ventajas y Desventajas

Ventajas:

- Paralelismo Masivo: Se dispone de miles de procesadores trabajando simultáneamente.
- Alto desempeño: Los dispositivos de cómputo dentro de la GPU están optimizados para operaciones en punto flotante.
- Precio: Es una solución de cómputo económica.
- Facilidad de uso: Se utilizan variaciones del lenguaje “C” para su programación (CUDA, OpenCL, etc.)

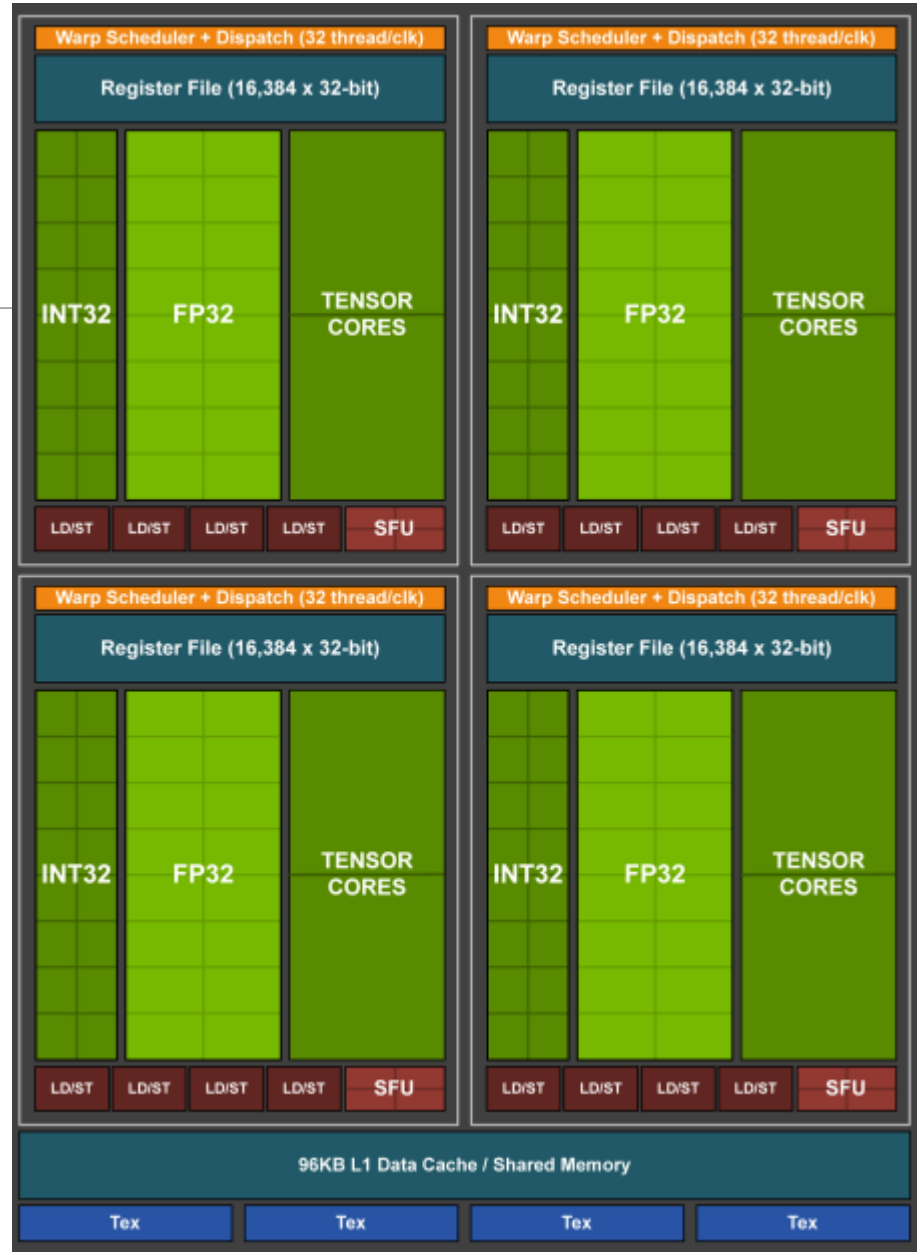
Desventajas:

- Tienen arquitecturas tipo SIMD, es decir, todos los procesadores deben ejecutar el mismo programa al mismo tiempo, no hacerlo así genera una penalización en desempeño
- Su procesamiento paralelo masivo solo es aplicable a algoritmos en los que el espacio de datos de entrada se puede descomponer o parcializar.
- Esta optimizada la transferencia de datos hacia la memoria de la GPU, la transferencia de datos desde la GPU es mucho más lenta.

GPU – Ejemplo de Arquitecturas

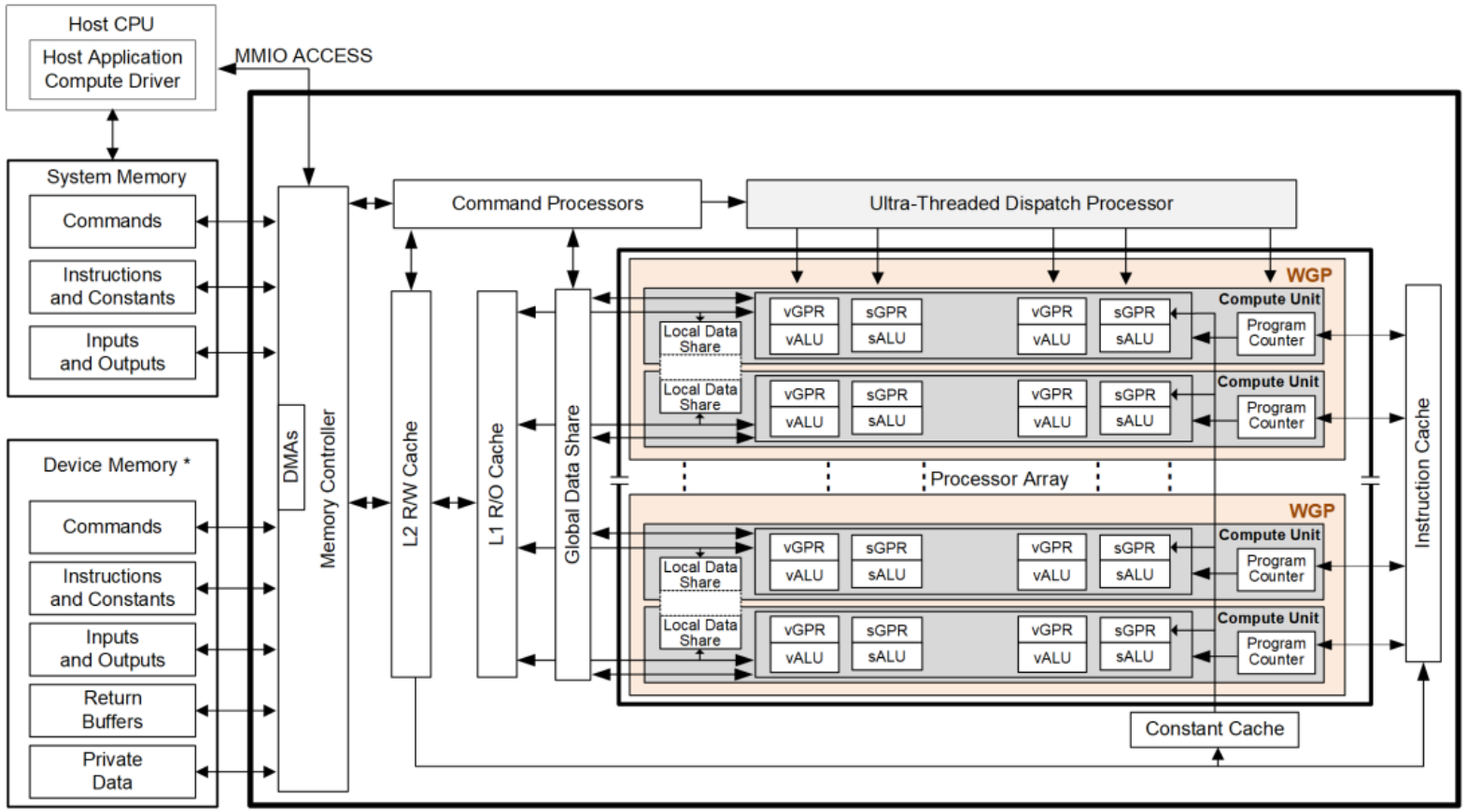
- Nvidia Turing: Tiene núcleos optimizados para calculo con números enteros, números en punto flotante y con tensores/matrices

- TU102:
  - 4608 Núcleos CUDA (INT32/FP32)
  - 576 Núcleos para tensores/matrices
  - 65536 registros de 32bits
  - Potencia de cómputo bruta: 13.4 TeraFLops (FP32)



GPU – Ejemplo de Arquitecturas

- AMD RDNA3: Tiene núcleos separados para operaciones escalares: sALU y sGPR (128 Regs x núcleo) y vectoriales: vALU y vGPR (1024 Regs x núcleo)
- Radeon RX7900XTX:
  - 96 Compute Units
  - 6144 Núcleos en total
  - Potencia de cómputo bruta: 61 TeraFlops (FP32)



GPU - Software

- CUDA (Compute Unified Device Architecture)
  - Solución propietaria de Nvidia
  - SDK más maduro y estable
  - interfaz con otros lenguajes y entornos (Python, Tensorflow, etc.) en forma transparente
  - Solo puede utilizarse con hardware de Nvidia (placas graficas RTX, SoCs Tegra, etc)
- OpenCL
  - Estándar abierto apoyado por múltiples proveedores (Intel, AMD, Apple, ARM, Google, Qualcomm, Samsung, Nvidia, etc.)
  - Cada fabricante debe proveer su propia implementación: SDKs de calidad variable
  - Interfaz única basada en C, se pueden realizar bindings a otros lenguajes o entornos
  - Gran variedad de hardware (computación heterogénea): CPU, GPU, FPGA, SoC, etc.
- APIs orientadas a gráficos (OpenGL, DirectX, Vulkan, etc.)

Temario

- Introducción
- GPU
  - **CUDA**
  - OpenCL
- FPGA – High Level Synthesis (HLS)
  - Vivado HLS – IP integrable (Orientado a Hardware)
  - Vitis HLS – Generación de Kernels en lógica programable (Orientado a Software)

CUDA

En CUDA se puede implementar sobre dos API:

- Runtime API: Es más simple, pero con menos versatilidad. Las inicializaciones y la gestión de la ejecución y manejo de memoria suelen ser implícitas.
- Driver API: API con mayor control sobre la inicialización, gestión de la ejecución y gestión de la memoria.

```
// Runtime API:
int* a;
cudaMalloc(&a, 4);
cudaMemcpy(a, c, 4, cudaMemcpyHostToDevice);
cudaDeviceSynchronize();

// Driver API:
CUdeviceptr b;
cuMemAlloc(&b, 4);
cuMemcpyHtoD(b, c, 4);
cuCtxSynchronize();
```

CUDA – Desarrollo

- CUDA Toolkit
- Entorno de desarrollo para C/C++ como ser GCC, ICC (Intel C Compiler), etc.
- Opcionalmente se puede usar Python como entorno, pero se pierde flexibilidad.
- Los archivos fuente tienen extensión .CU y pueden tener tanto código para CPU como para GPU
- La parte de GPU se compila en forma separada con el compilador de CUDA (Nvidia CUDA Compiler – NVCC)
- NVCC tiene una sintaxis similar a GCC: nvcc <fuente.cu> –o <ejecutable>
- CMAKE provee soporte para compilar con NVCC para automatizar el proceso de compilado

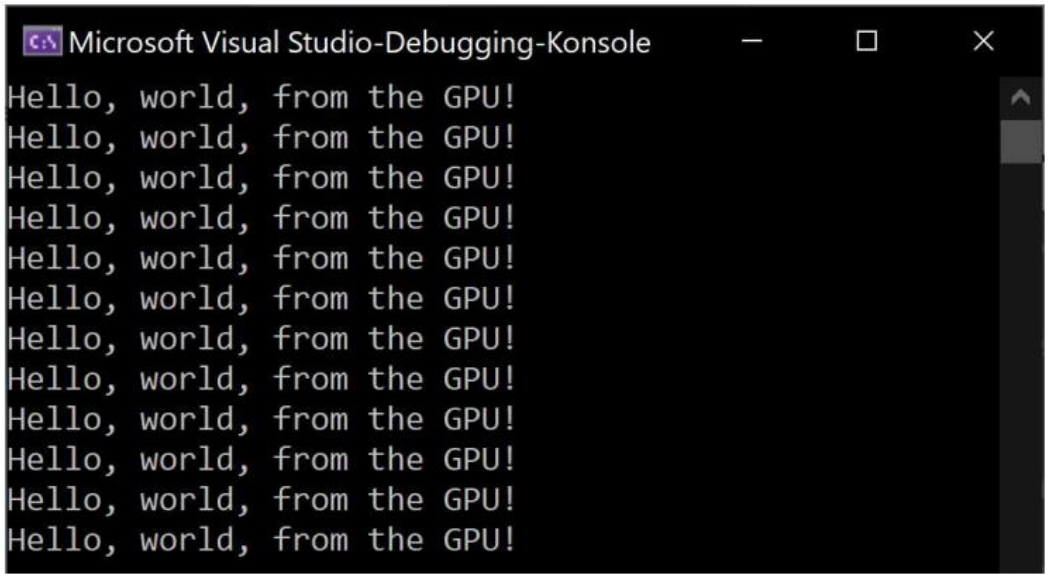
CUDA – Modelo de Ejecución

- El sistema basado en CPU donde está instalada la GPU se denomina HOST
- Un Host puede tener varias GPU, a cada GPU individual se la denomina DEVICE
- Hay dos tipos de funciones que se pueden definir:
  - Funciones tipo Kernel (__global__): reciben argumentos, no pueden devolver resultados, pueden ejecutarse en el Host o en un Device
  - Funciones tipo Dispositivo (__device__): solo pueden recibir parámetros definidos en otras funciones Kernel o Dispositivo, pueden devolver cualquier tipo de resultado, solo pueden ejecutarse en un Device

Ejemplo de Kernel

- Se lanzan 12 instancias en paralelo en la GPU y se utiliza una sincronización entre Device y Host para continuar después que todas las instancias en la GPU finalizaron.

```
__global__ void HelloWorldGPU()
{
    printf("Hello, world, from the GPU!\n");
}

int main()
{
    HelloWorldGPU<<<1,12>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

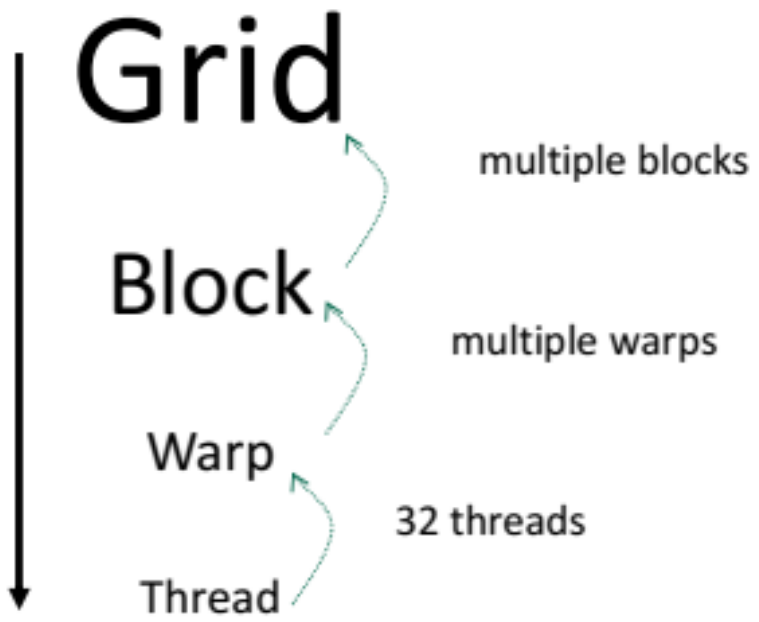


Granularidad de la ejecución

- Execution occurs in a hierarchical model

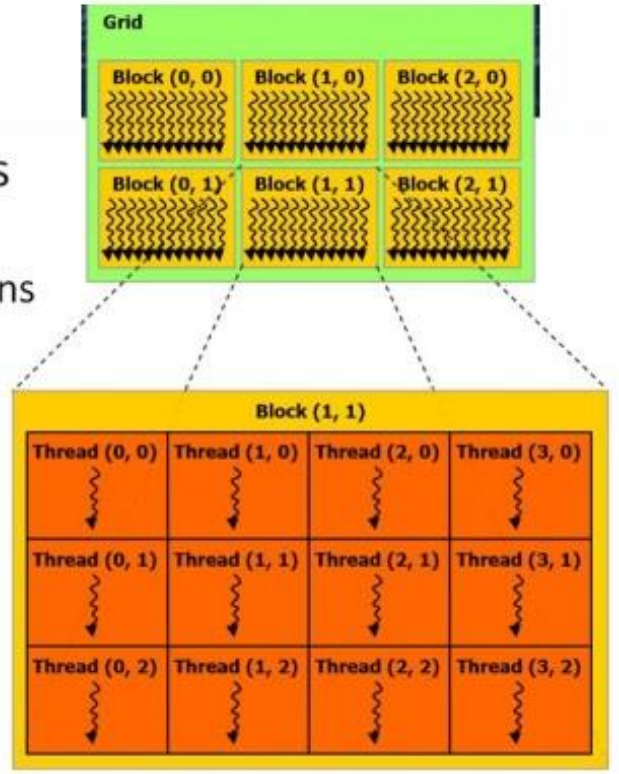
- CUDA distinguishes four granularities:
  - Grid (launch configuration)
  - Block (cooperative threads)
  - Thread (isolated execution state)

- In-between: warps
  - Groups of 32 threads, enable SIMD execution
  - Implicitly defined as parts of a block



Granularidad de la ejecución

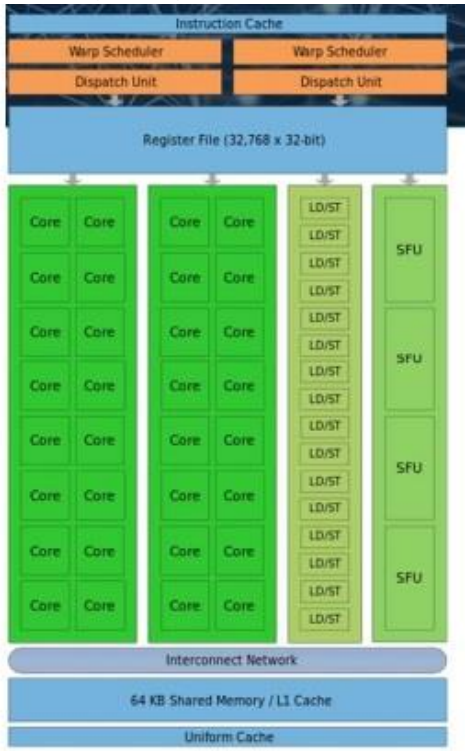
- Grid defines total number of launched threads
  - Indirectly, via the number of blocks
  - Complete grid defined by grid and block dimensions
- Threads within a block can synchronize
- Up to 32 threads (a warp) execute the same instruction on the same SIMD compute unit



Distribución de la Ejecución

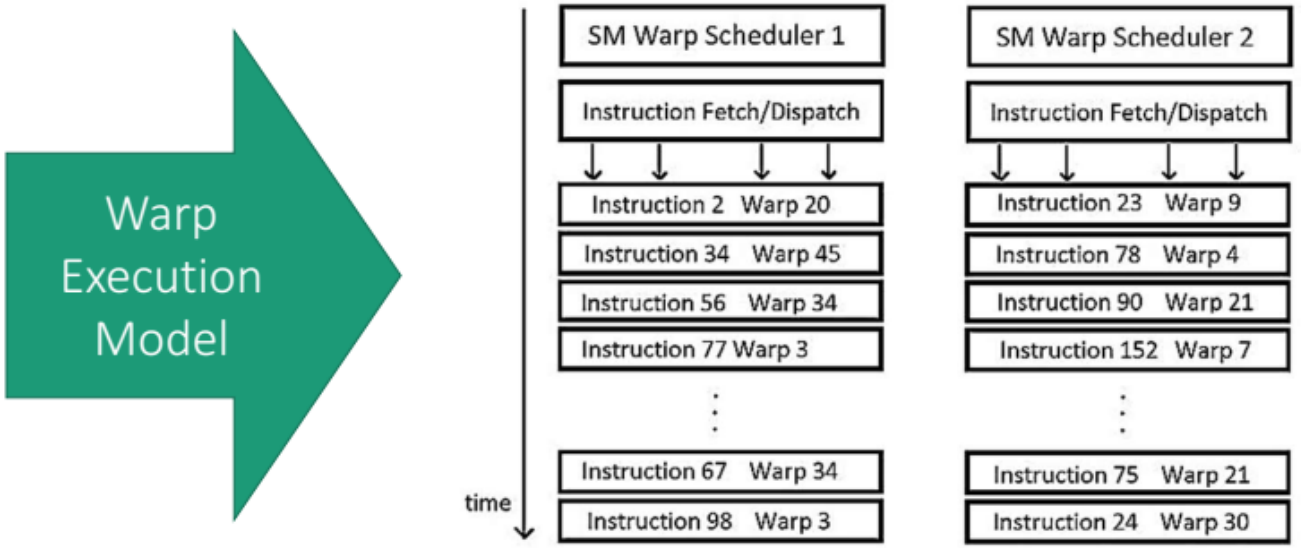
- El procesador de la GPU se denomina Streaming Processor (SM)

- CUDA cores: basic integer/floating point arithmetic – high throughput, low latency
- Load/Store (LD/ST): issues memory accesses to appropriate controller – possibly high latency
- Special Function Unit (SFU): trigonometric math functions, etc – reduced throughput
- Since Turing and Volta, also include special **tensor cores** (not explicitly shown here)



Distribución de la ejecución

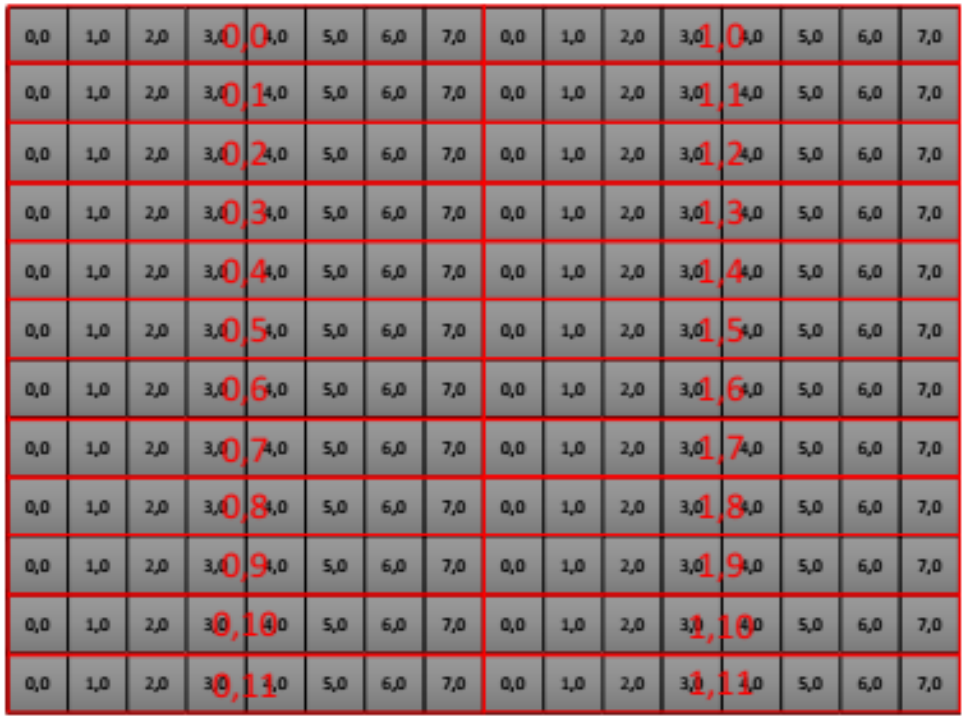
- El comportamiento de un thread suele estar determinado por un identificador único (global o local)
- Todos los threads de un warp ejecutan la misma instrucción (convergencia). Hay un solo Program Counter por warp
- Distintos warps pueden estar en distintos puntos de ejecución del programa (por ejemplo por accesos a memoria) (divergencia)



Identificadores en threads

- Program flow can vary depending on `threadIdx` and `blockIdx`, `gridDim` and `blockDim`

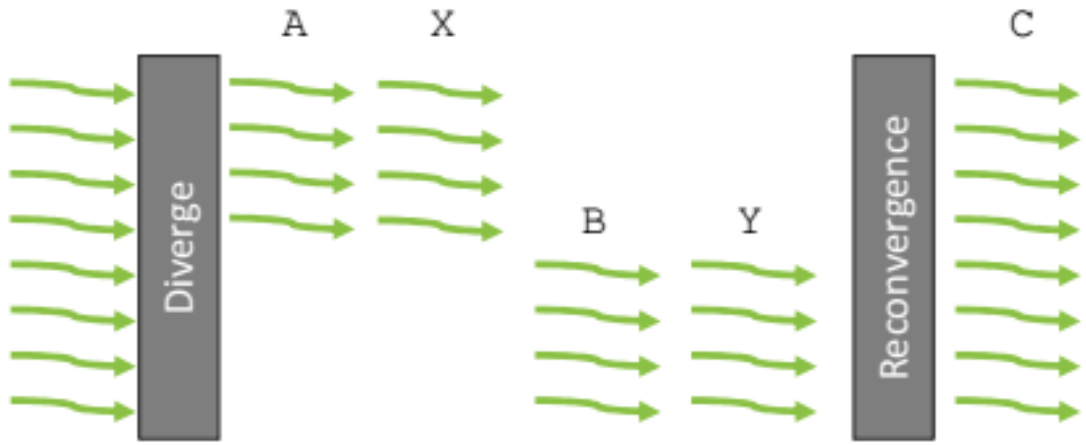
```
__global__ void PrintIDs()
{
    auto tID = threadIdx;
    auto bID = blockIdx;
    printf("Thread Id: %d,%d\n", tID.x, tID.y);
    printf("Block Id: %d,%d\n", bID.x, bID.y);
}

int main()
{
    …
    dim3 gridSize = { gridX, gridY, gridZ };
    dim3 blockSize = { blockX, blockY, blockZ };
    PrintIDs<<<gridSize, blockSize>>>();
    cudaDeviceSynchronize();
    …
```



Divergencia

- Cuando hay divergencia de ejecución, primero se ejecutan todas las variantes y luego se converge. Todos los threads, independientemente del camino de ejecución, se sincronizan implícitamente antes de ejecutar “C()”

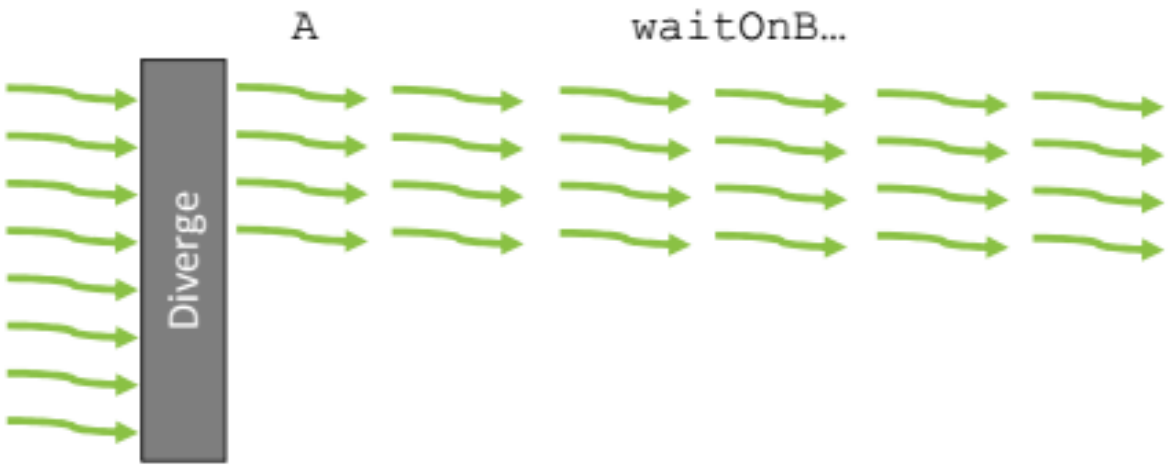
```
if(threadIdx.x & 0x4)
{
    A();
    X();
}
else
{
    B();
    Y();
}
C();
```



Deadlocks

- La capacidad de divergencia de los threads puede producir deadlocks

```
if(threadIdx.x & 0x4)
{
    A();
    waitOnB();
}
else
{
    B();
    waitOnA();
}
C();
```



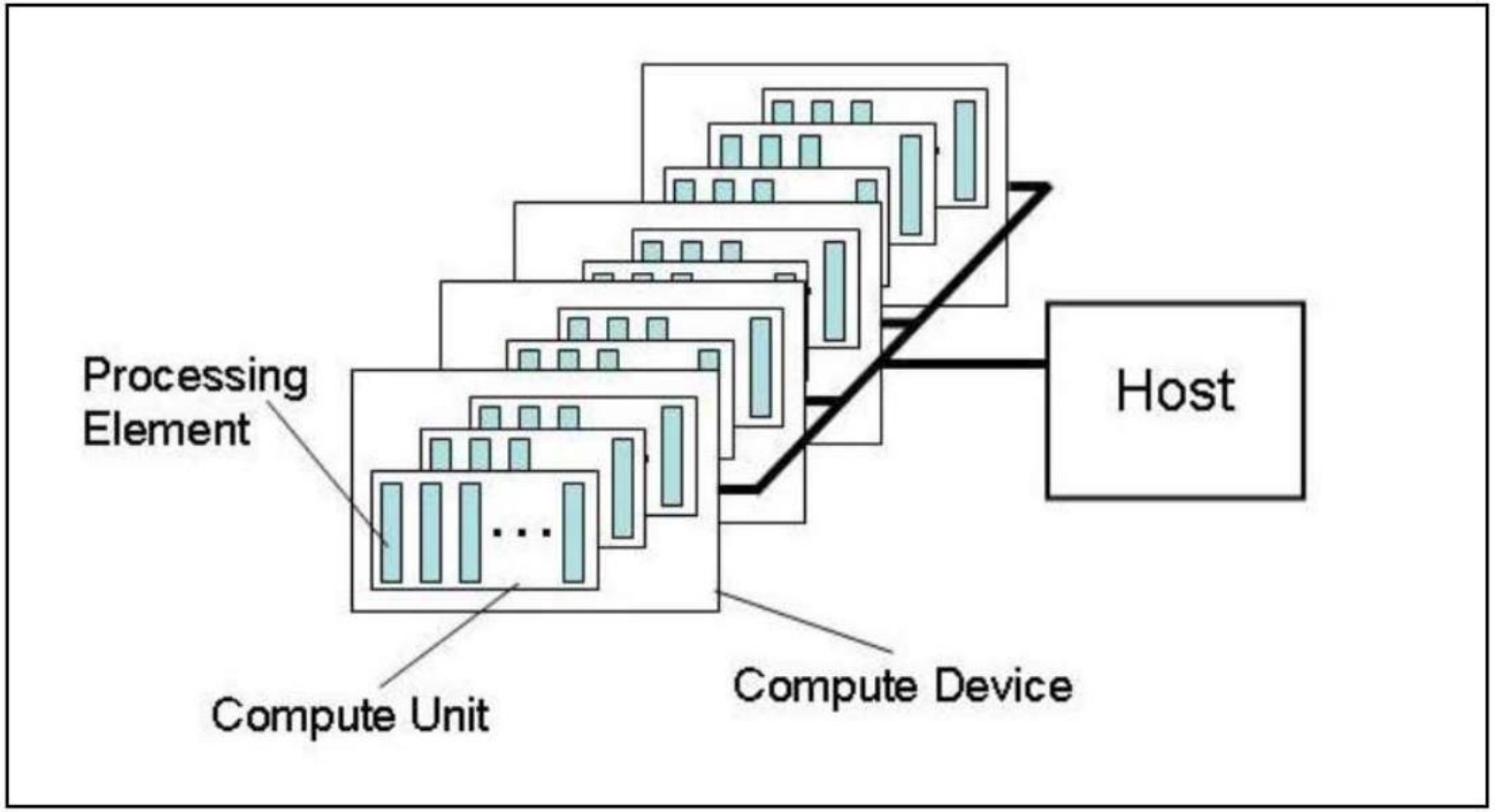
Temario

- Introducción
- GPU
  - CUDA
  - **OpenCL**
- FPGA – High Level Synthesis (HLS)
  - Vivado HLS– IP integrable (Orientado a Hardware)
  - Vitis HLS – Generación de Kernels en lógica programable (Orientado a Software)

OpenCL

- Es un estándar abierto para programación paralela sobre distintos tipos de dispositivos (computación heterogénea)
- Los dispositivos pueden ser CPUs, GPUs, Procesadores embebidos, SoCs, etc.
- El paralelismo puede ser basado en tareas (múltiples tareas ejecutándose simultáneamente) y basado en datos (una misma tarea operando con distintos datos)
- Se logra una abstracción independiente del hardware
- Se tiene un control granular sobre la precisión de las operaciones de punto flotante

Modelo de abstracción del hardware



Modelo de abstracción del hardware

- Host: sistema basado en CPU que integra un dispositivo compatible con OpenCL (GPU, FPGA, etc).
- Compute Device: un dispositivo compatible con OpenCL.
- Compute Unit: Un procesador dentro de un Compute Device.
- Processing Element: Un dispositivo de cómputo elemental.
- En un sistema hay un Host y uno o varios Device.
- La memoria se divide en Host Memory (accesible por la CPU) y Device Memory (accesible por el Device)

OpenCL

- Se define un dominio computacional de dimensión N
- Se ejecuta el algoritmo sobre cada elemento del dominio

Traditional loops

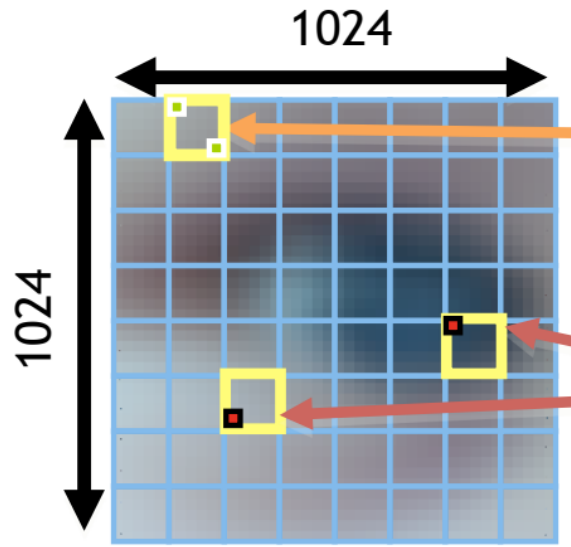
```
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

Data Parallel OpenCL

```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] * b[id];

} // execute over "n" work-items
```

Partición del dominio

- **Global** Dimensions:
  - 1024x1024 (whole problem space)
- **Local** Dimensions:
  - 128x128 (**work-group**, executes together)



Synchronization between work-items possible only within work-groups: barriers and memory fences

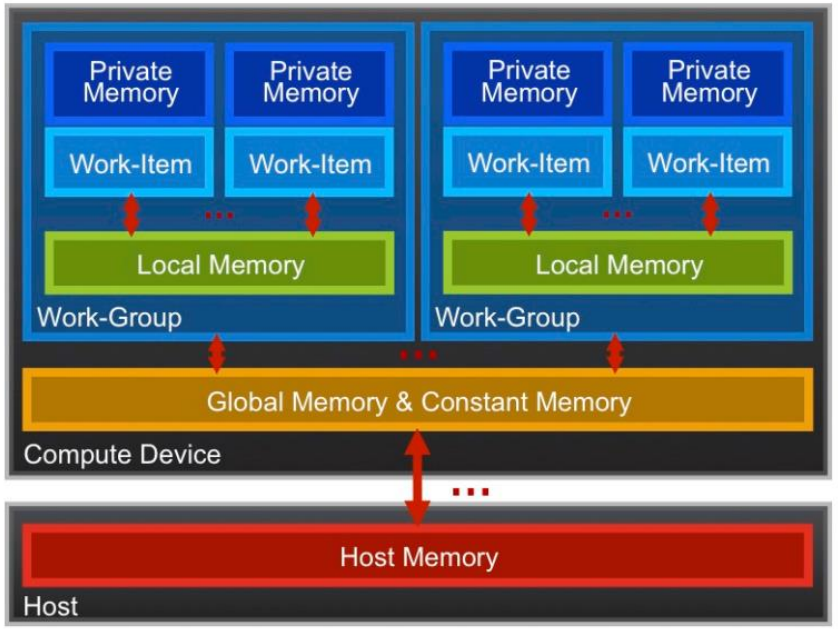
Cannot synchronize between work-groups within a kernel

Modelo de memoria

La gestión de memoria es explicita, las transferencias entre las distintas zonas de memoria (host, global y local) se debe realizar mediante llamadas a funciones.

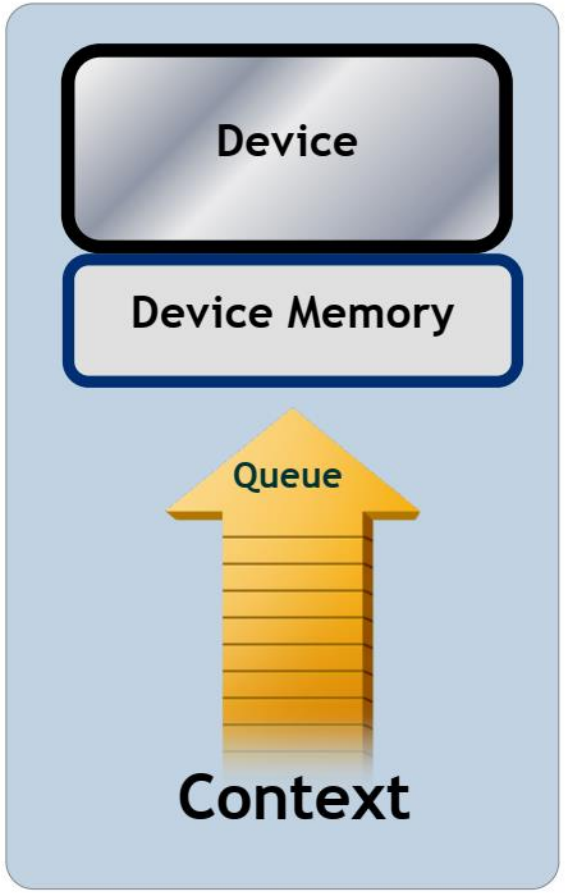
OpenCL Memory model

- ***Private Memory***
  - Per work-item
- ***Local Memory***
  - Shared within a work-group
- ***Global Memory***
  Constant Memory
  - Visible to all work-groups
- ***Host memory***
  - On the CPU



Ejecución

- Contexto (Context): El entorno donde se ejecutan los kernel y se realiza la sincronización y gestión de la memoria.
- El contexto incluye:
  - Uno o mas Device
  - Device Memory
  - Una o mas Command Queue
- Los comandos a un Device (ejecución de un kernel, sincronización y operaciones de transferencia de memoria) se realizan a través de una Command Queue
- Cada Command Queue está asociada a un solo Device dentro de un Contexto

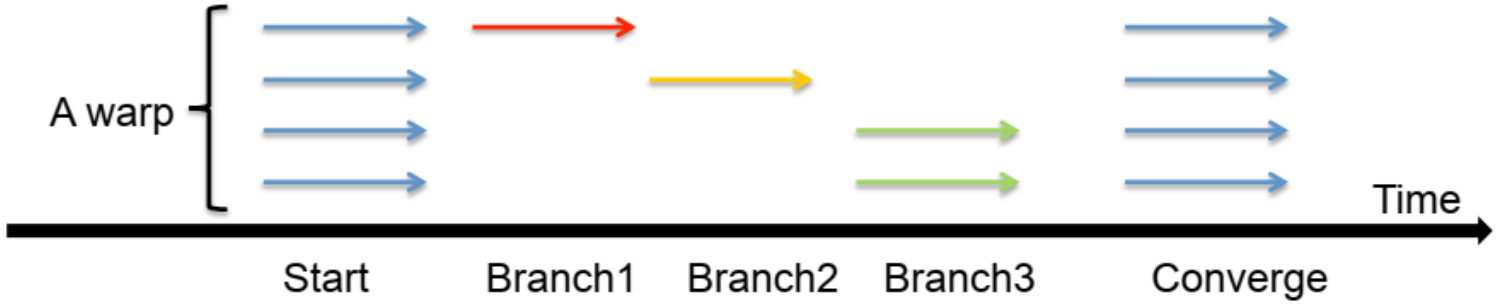


Ejecución

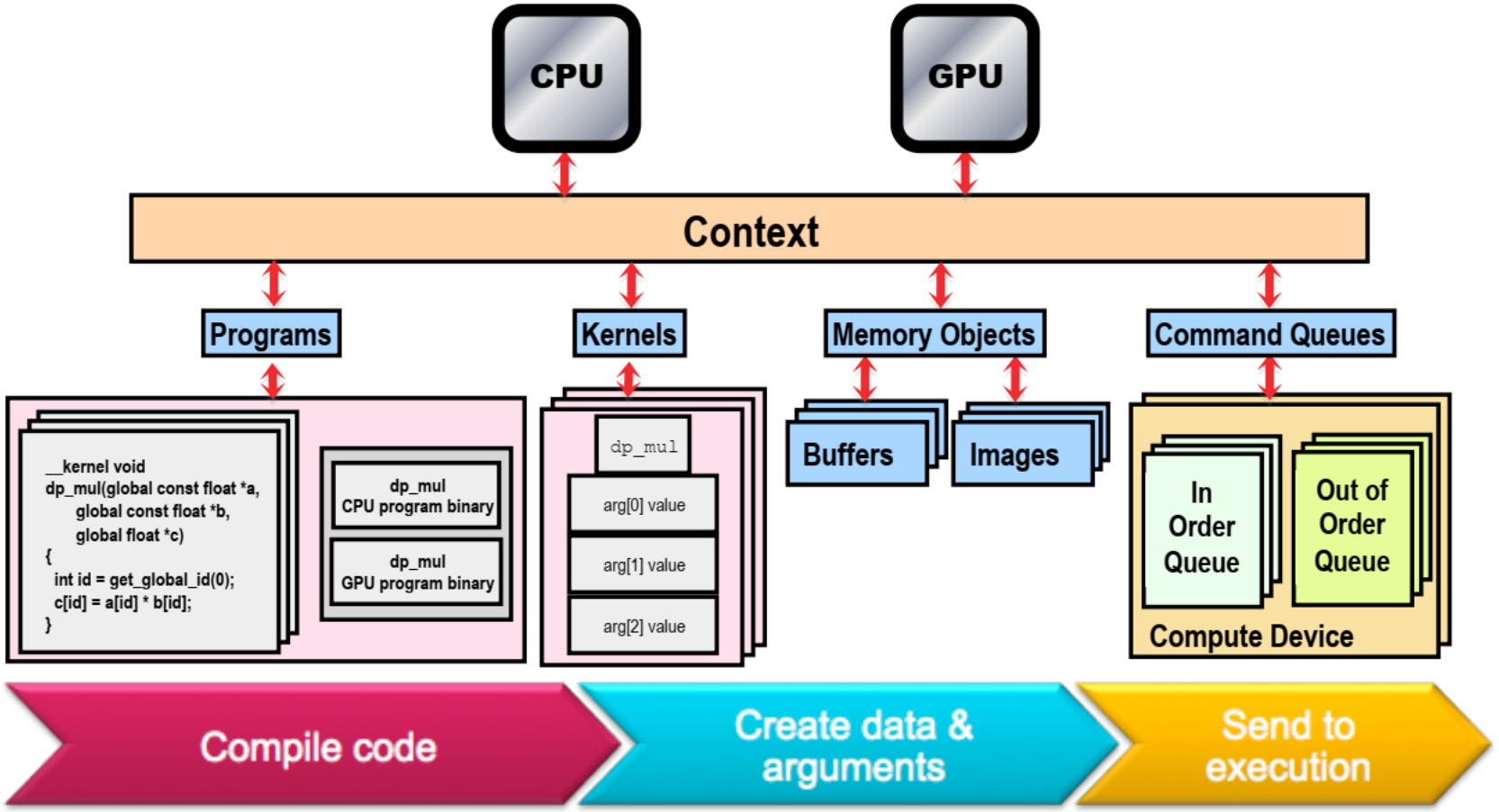
- El Software se divide en Host Code y Device Code (Kernel)
- El Host Code se ejecuta en el Host y genera el contexto y crea los kernel. La compilación de los kernel es en run-time para generar Device Code de acuerdo al Device que se utilice.
- El Host Code realiza las siguientes tareas
  - Definir la Plataforma (Contexto, Devices y Command Queues)
  - Compilar el Device Code
  - Configurar los Memory Objects (buffers para transferencia entre Host y Device)
  - Generar el Kernel (Device Code + Argumentos)
  - Enviar comandos al Device mediante Command Queue (transferencias de datos, ejecución de kernels, sincronización, etc)
- El Device Code se ejecuta en el Device y es el que realiza el cómputo

Ejecución – Divergencia

- Cada Work Item es una colección que se ejecuta concurrentemente en el hardware (es el análogo de un Warp de CUDA)
- Los Work Item se ejecutan en una modalidad SIMD con cada elemento del Work Item ejecutándose a partir de la misma dirección de memoria de programa
- Cada Work Item tiene su propio Program Counter y Conjunto de Registros, por lo que pueden tener distintas bifurcaciones y divergir, generando una modalidad MIMD.
- Cada bifurcacion dentro del Work Item se ejecuta en forma seriada. Los elementos del Work Item que no tomaron una bifurcación especifica se deshabilitan durante la ejecución de esa bifurcación



Ejecución


CPU
GPU
Context
Programs
Kernels
Memory Objects
Command Queues
__kernel void
dp_mul(global const float *a,
global const float *b,
global float *c)
{
int id = get_global_id(0);
c[id] = a[id] * b[id];
}
dp_mul
CPU program binary
dp_mul
GPU program binary
dp_mul
arg[0] value
arg[1] value
arg[2] value
Buffers
Images
In Order Queue
Out of Order Queue
Compute Device
Compile code
Create data & arguments
Send to execution

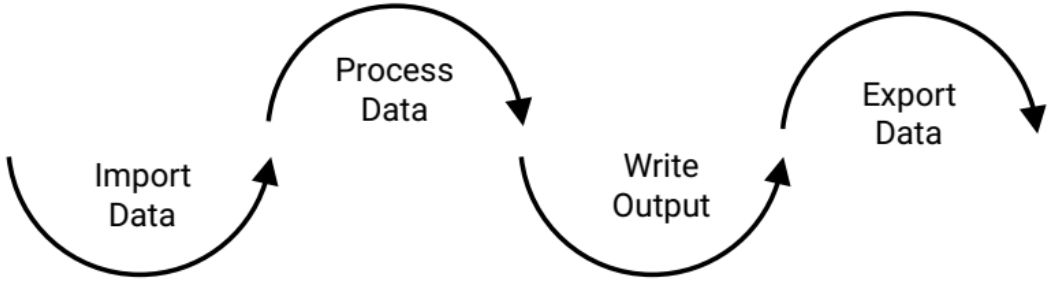
Temario

- Introducción
- GPU
  - CUDA
  - OpenCL
- **FPGA – High Level Synthesis (HLS)**
  - Vivado HLS – IP integrable (Orientado a Hardware)
  - Vitis HLS – Generación de Kernels en lógica programable (Orientado a Software)

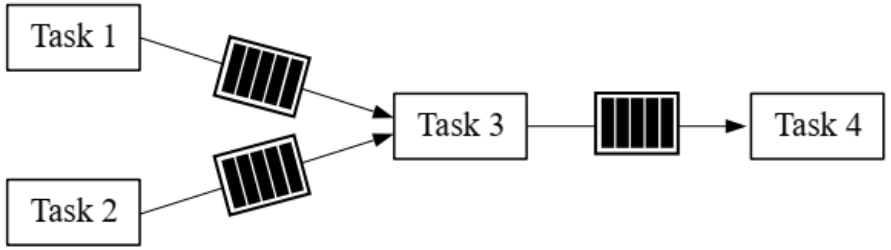
FPGA

Las FPGA pueden realizar procesamiento mediante tres paradigmas

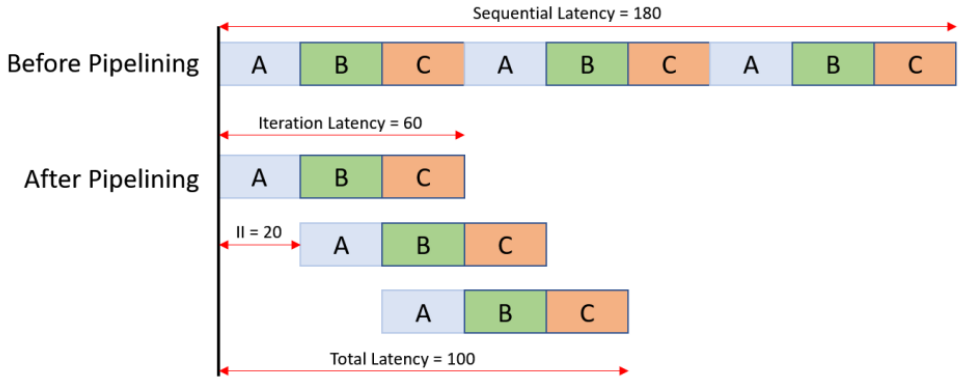
- Productor – Consumidor



- Streaming de Datos

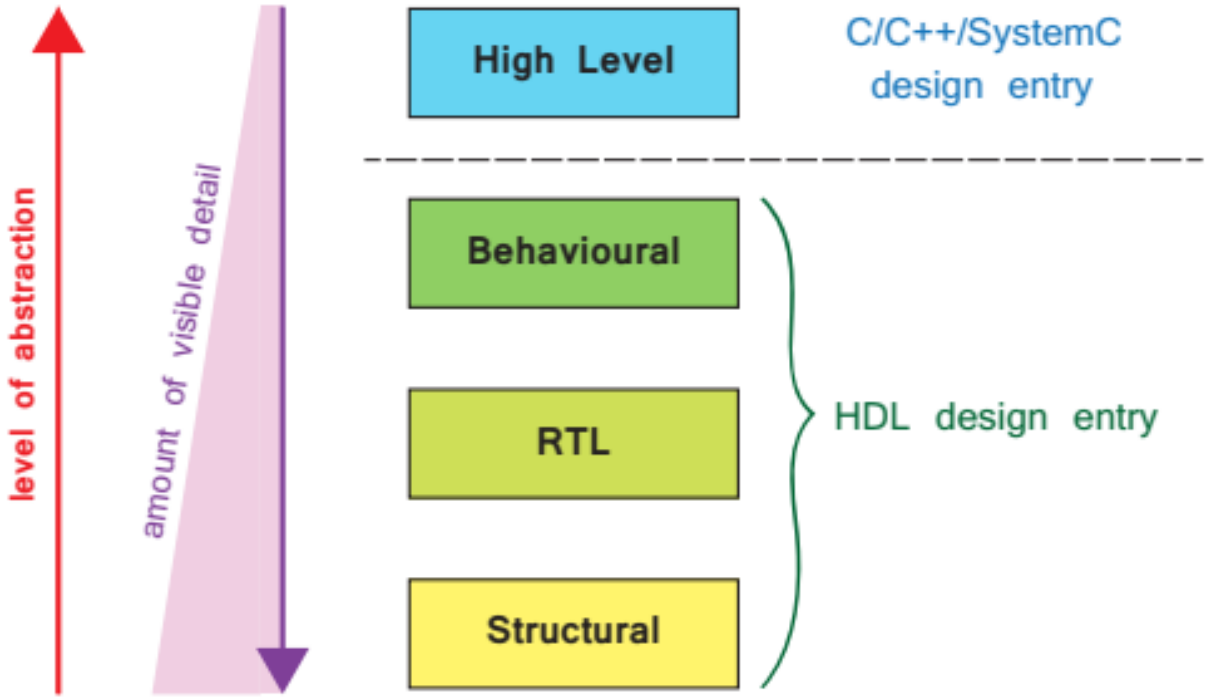


- Pipelining



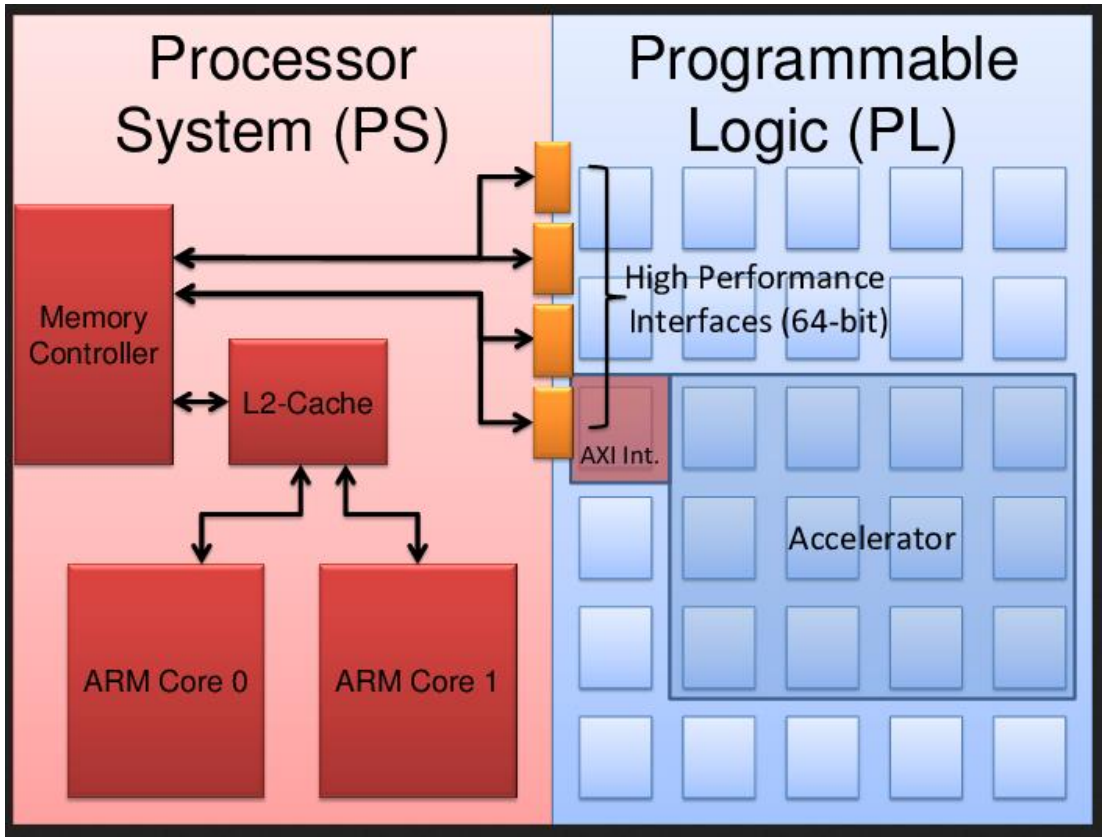
FPGA - HLS

- El uso de HLS es un nivel de abstracción mayor en el diseño



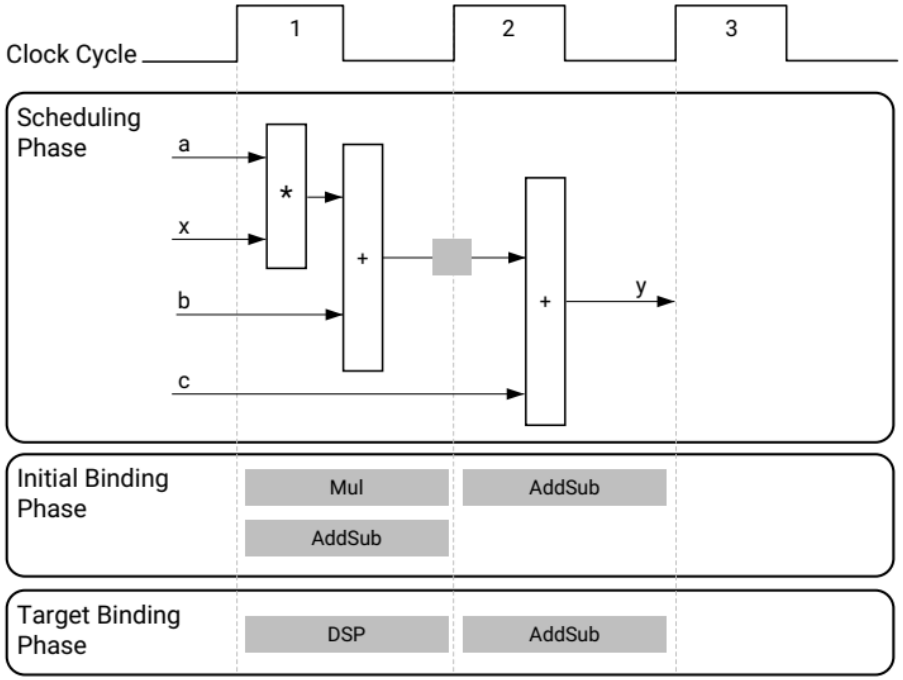
FPGA

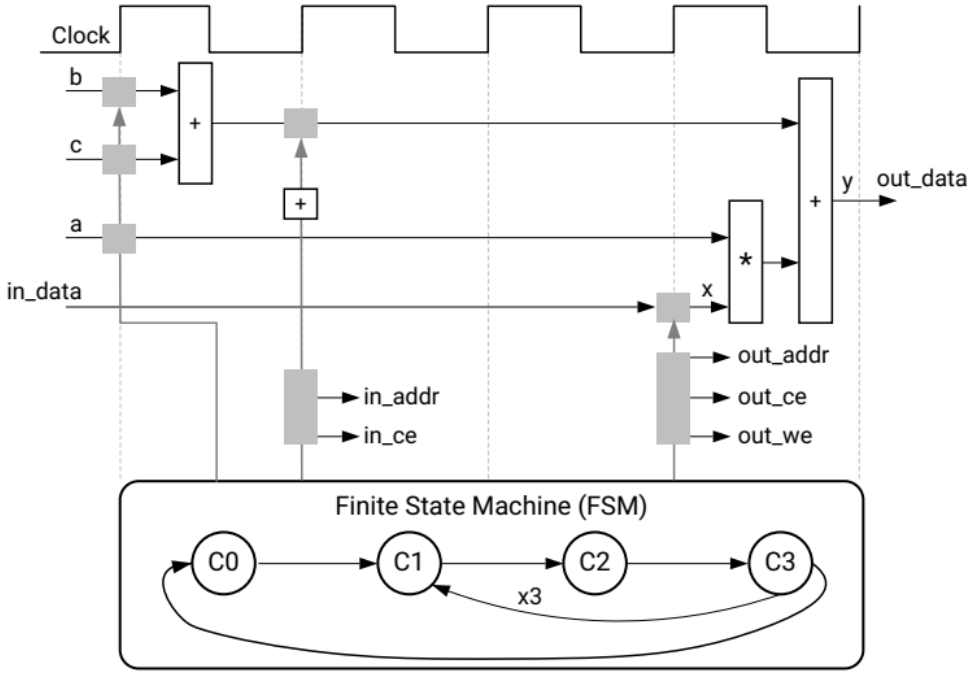
- En sistemas mixtos Procesador – Lógica programable se pueden implementar aceleradores en la lógica programable, los cuales son gestionados desde el software ejecutándose en el procesador.
- Estos aceleradores pueden generarse como bloques IP o como Kernels de OpenCL



Ejemplo

```
int foo(char x, char a, char b, char c) {
 char y;
 y = x*a+b+c;
 return y;
}
```



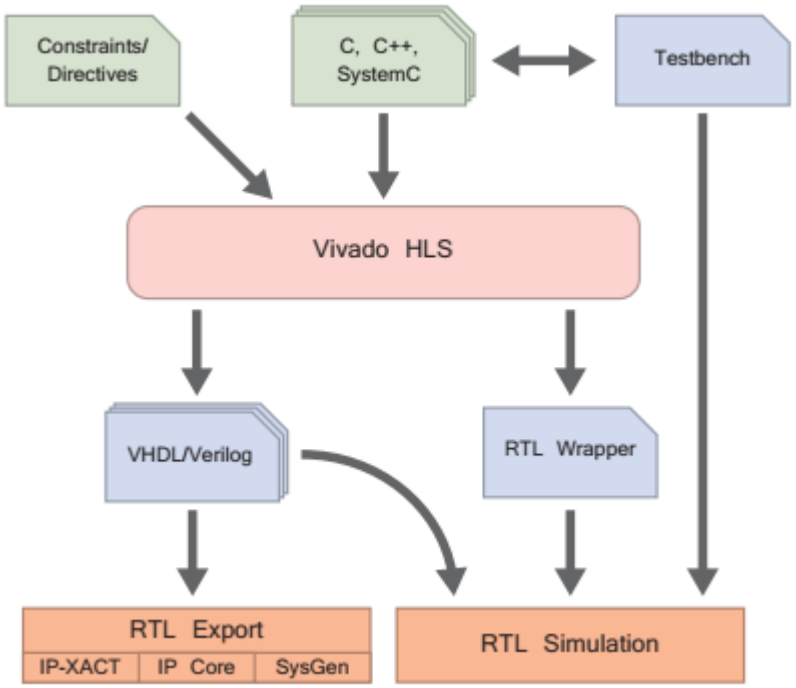


Temario

- Introducción
- GPU
  - CUDA
  - OpenCL
- FPGA – High Level Synthesis (HLS)
  - **Vivado HLS – IP integrable (Orientado a Hardware)**
  - Vitis HLS – Generación de Kernels en lógica programable (Orientado a Software)

Vivado HLS

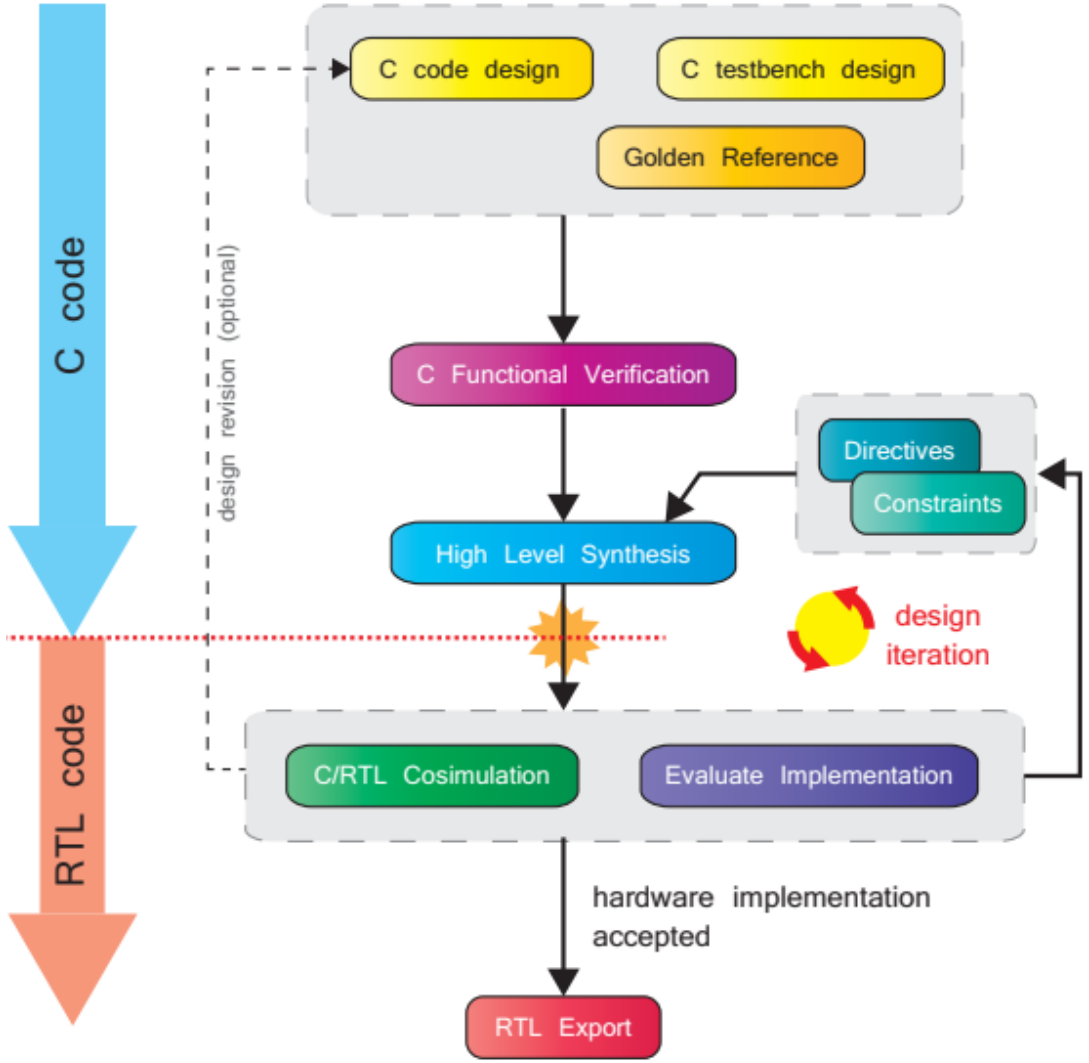
- Es una herramienta que convierte funciones de C/C++ a un diseño RTL en un HDL (VHDL o Verilog) implementable en FPGA



Vivado HLS – interfaces

- El acelerador puede implementar 3 interfaces
- IP-XACT: descripción de IP independiente de la herramienta que la generó. Se utiliza se exporta el acelerador (p.ej. a Mentor)
- IPCore: formato estándar para integrar en un proyecto utilizando las herramientas de Xilinx
- SysGen: formato que genera un bloque que puede utilizarse dentro de Simulink (MATLAB) para cosimulacion hardware-software

Vivado HLS – Flujo de Diseño



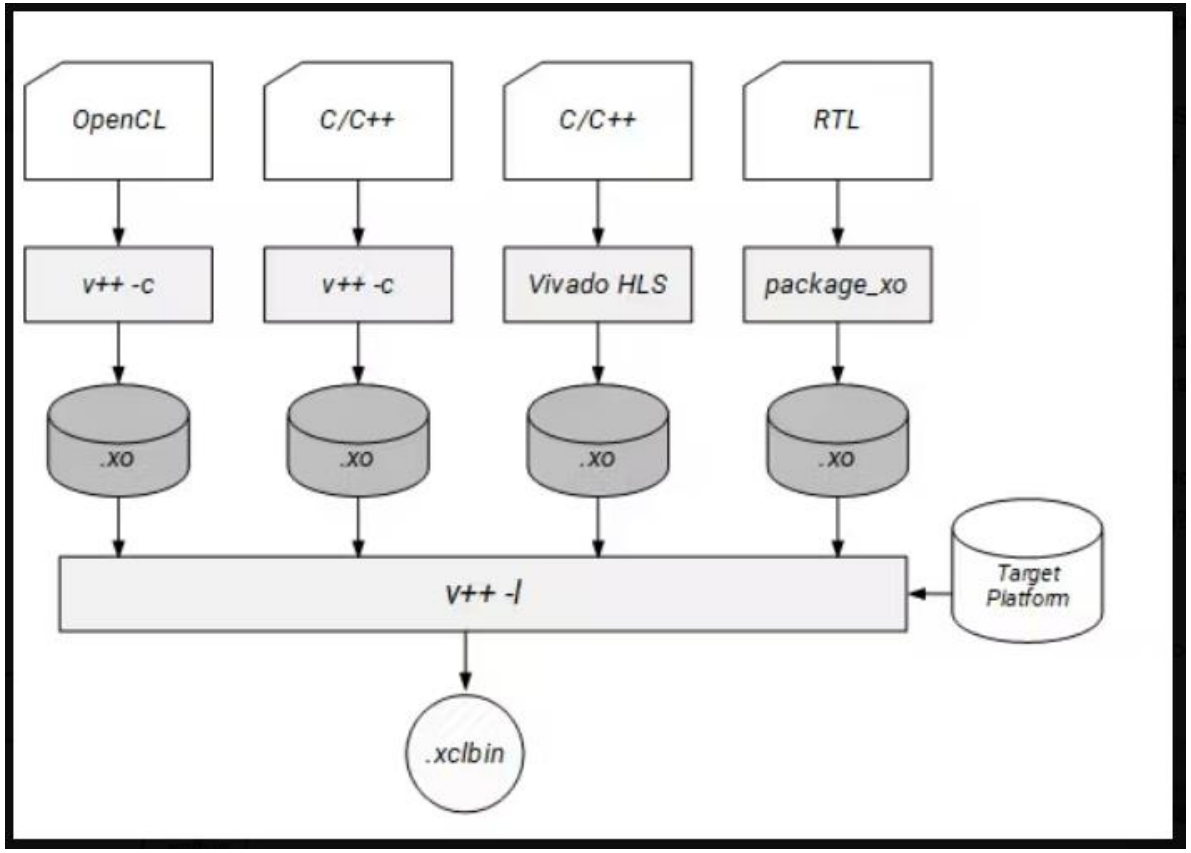
Temario

- Introducción
- GPU
  - CUDA
  - OpenCL
- FPGA – High Level Synthesis (HLS)
  - Vivado HLS – IP integrable (Orientado a Hardware)
  - **Vitis HLS – Generación de Kernels en lógica programable (Orientado a Software)**

Vitis HLS

- Es una herramienta que convierte funciones de C/C++/OpenCL a un diseño RTL implementable en lógica programable
- Ejecuta el siguiente flujo de diseño
  - Compilado/simulación/depuración código en C/C++
  - Optimización del código
  - Conversion (Síntesis) del código en un diseño RTL en HDL (Verilog)
  - Verificación de la implementación RTL
  - Generación de un archivo .xo con la implementación

Vitis HLS


OpenCL
C/C++
C/C++
RTL
v++ -c
v++ -c
Vivado HLS
package_xo
.xo
.xo
.xo
.xo
v++ -l
Target Platform
.xclbin

Vitis HLS - interfaces

- Se utiliza en un sistema basado en procesador (procesador, bus, periféricos, sistema operativo, etc)
- La interfaz al acelerador puede ser de tres tipos
- Memoria: se transfieren datos desde/hacia una memoria (DDR/BRAM/etc)
- Streaming: se transfieren datos desde/hacia otro acelerador o una fuente que genera/consume datos de forma continua (ej. Adquisidor de video, salida HDMI)
- Registros: el acelerador se mapea como un periférico y la transferencia de datos se realiza mediante operaciones de lectura/escritura
- El código ejecutado en el procesador utiliza OpenCL para gestionar la Aceleración, de esta manera el acelerador se abstrae a un Device y el código que lo gestiona se ejecuta en Host

Bibliografia

- Computer Organization and Design: The hardware/software interface, RiscV Edition – Patterson & Hennessy
- Nvidia Turing GPU Architecture Whitepaper – Nvidia
- RDNA3 Instruction Set Architecture – AMD
- Cuda C Programming Guide - Nvidia
- OpenCL Guide – Khronos Group
- Vitis High Level Synthesis User Guide – Xilinx