

# Taller de programación paralela en GPUs con oneAPI

X Semana Informática 2024

Carlos García Sánchez

UCM

21 de febrero de 2024

- “Especificaciones de SYCL 2020”,  
<https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html>
- “Data Parallel C++”, <https://link.springer.com/book/>



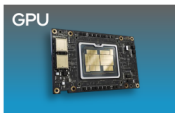
# Outline

- 1 Tendencias actuales
- 2 Programación
- 3 SYCL
- 4 Dispositivos
- 5 Modelo memoria
- 6 Niveles de paralelismo



# Introducción

- Alto rendimiento (HPC) solía ser un cuestión **exclusiva** en la gran ciencia
- ... pero está siendo una característica fundamental en otros ámbitos
  - IA, análisis de datos, creación de contenido o gráfico
- Proliferación de arquitecturas : GPUs, FPGAs, ASICs...
- **Programadores lidian con la complejidad de los diferentes modelos de programación en cada tipo de acelerador**



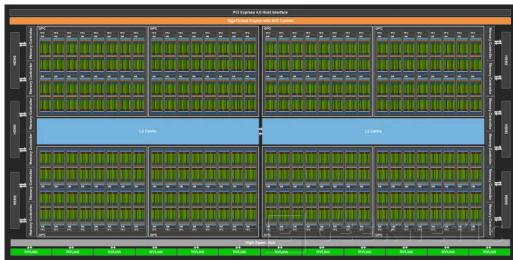
# Paralelismo (CPU)

- Búsqueda de mejorar rendimiento del sistema (a nivel de nodo):
  - ILP: hw extrae instrucciones independientes
  - DLP: pequeños vectores en x86 i.e: SSE128b, AVX256b, AVX512
    - Autovectorización en compiladores GNU-GCC o Intel ICX
  - TLP: explotación de multiples cores con hilos de ejecución
  - Heterogeneo: GPU vistas como coprocesador
    - GPUs como procesadores altamente paralelos

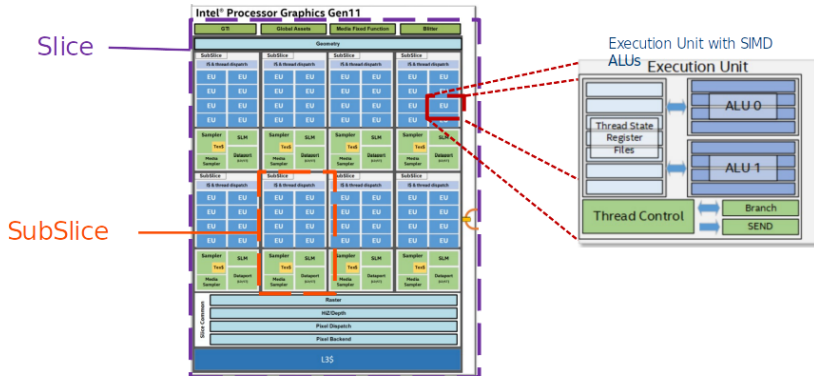


# NVIDIA- Ampere

- 8 GPCs (GPU Processing Clusters) x 8 TPCs (Texture Processing Clusters) x 2SMs = 128 SMs
- SM: 64 FP32 CUDA cores, 8 FP64 CUDA Cores, 4 Tensor Cores

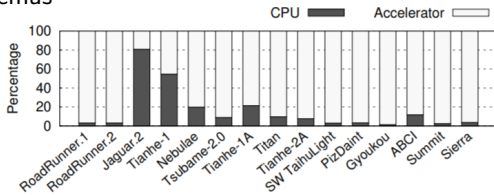


# Arquitectura Intel



# Motivación

- Heterogeneous-Computing (CPU+Accel) es una de las claves para mejorar rendimiento en **top500**<sup>1</sup>
  - Proporción de aceleradores en el rendimiento general del sistema para las 16 supercomputadoras heterogéneas más potentes
  - **Aceleradores** contribuyen con un 84 % del  $R_{peak}$  de los sistemas



(a) Flops ratio.

<sup>1</sup>Extraído de “An Analysis of System Balance and Architectural Trends Based on Top500 Supercomputers”, HPC Asia 2021



# Motivation

- Desafíos: muchos modelos de programación
  - Más abstracción vs Más Rendimiento

+++ Abstraction					+++ Performance
python	C/C++ Fortran	OpenMP (Cores)	OpenMP target OpenACC	OpenCL CUDA	Vector Intrs. GPU Intrinsics

## Retos

- 1 (Variedad): Muchos lenguajes con sus toolchains, versión a mantener e integrar
- 2 (Performance): Desarrollo de app con alto rendimiento habitualmente conlleva desarrolladores especializados
- 3 (Porting): Algunas abstracciones ofrecen soluciones para alto-rendimiento en diferentes arquitecturas





# Algunas comparaciones

- Algunos modelos de programación para aceleradores
  - CUDA moelo propietario
  - Directivas como OpenMP permite aplicar técnicas de programación incremental para sistemas heterogeneos (ej: accelerator offloading, tasks...)
    - Lista de compiladores que soportan OpenMP-offload: <https://www.openmp.org/resources/openmp-compilers-tools>
  - SYCL: multi-vendor, estandard definido por grupo Khronos

	CUDA	OpenACC	OpenMP (5.0)	SYCL
Language	C/C++	C/C++ Fortran	C/C++ Fortran	C/C++
Prog. Style		pragmas	pragmas	C++11 lambdas
Parallelism	SIMT	SIMD, Fork/join CUDA	SPMD, SIMD Tasks, Fork/join, CUDA	OpenCL
Licensing	Proprietary	Few comp.	Open-source	Open-source
Abstraction	Low	High	High	Medium



# Ejemplo: Suma vectores (C)

## vectorAdd.c

```
// Compute vector sum C = A+B
void vecAdd(float* A, float* B, float* C,
            int n)
{
    for (i = 0, i < n, i++)
        C[i] = A[i] + B[i];
}

int main()
{
    // Memory allocation for A_h, B_h, C_h
    // I/O to read A_h and B_h, N elements
    ...
    vecAdd(A_h, B_h, C_h, N);
}
```

... portalo para aceleradores

- Reescritura para cada kernel



# Ejemplo: Suma vectores (CUDA)

vectorAdd.cu

```
// Compute vector sum C = A+B
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}

int main()
{
    float* A_d, B_d, C_d;
    int size = n* sizeof(float);

    // A, B and C malloc and init
    ...

    // Get device memory for A, B, C
    // copy A and B to device memory
    cudaMalloc((void **) &A_d, size);
    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &B_d, size);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    // Kernel execution in device
    // (vector add in device)
    dim3 DimBlock(256, 1, 1);
    dim3 DimGrid(ceil(n/256.0), 1, 1);
    vecAddKernel<<DimGrid,DimBlock>>>(A_d, B_d, C_d, n);

    // copy C from device memory
    // free A, B, C
    cudaFree(A_d); cudaFree(B_d); cudaFree (C_d);
}
```



## Ejemplo: Suma vectores (OpenCL)

### add\_vector\_kernel.cl

```
// many instances of the kernel,  
// called work-items, execute in parallel  
__kernel void add_vector_kernel(__global const float *a,  
    __global const float *b, __global float *c)  
{  
    int id = get_global_id(0);  
    c[id] = a[id] + b[id];  
}
```



# Ejemplo: Suma vectores (OpenCL)

**CREAR PROGRAMA**

```
// create the OpenCL context on a GPU device
cl_context ctx = clCreateContext(0, 1, &CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);

cl_device_id[] devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices, NULL);
```

**PREPARAR MEMORIA**

```
// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0], 0, NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA, NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcB, NULL);

memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
    sizeof(cl_float)*n, NULL, NULL);
```

**CREAR PROGRAMA**

```
program = clCreateProgramWithSource(context, 1,
    &program_source, NULL, NULL);
```

**CREAR PROGRAMA**

```
// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
```

**CREAR KERNEL**

```
// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);
```

**Y ENVÍO DATOS**

```
// set the args values
err = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
    sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *) &memobjs[1],
    sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *) &memobjs[2],
    sizeof(cl_mem));
```

**EXECUTION KERNEL**

```
// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL,
    global_work_size, NULL, 0, NULL, NULL);
```

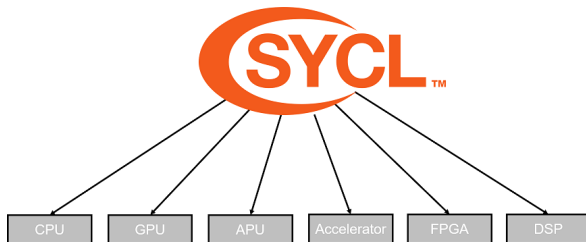
**ENVÍO RESULTADOS**

```
// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],
    CL_TRUE, 0, n * sizeof(cl_float), dst,
    0, NULL, NULL);
```



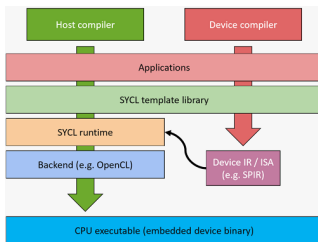
# ¿Que es SYCL?

- SYCL es un modelo de programación C++ estándar de alto nivel
- Código fuente único que puede apuntar a una **variedad de plataformas heterogéneas**



# ¿Que es SYCL?

- SYCL le permite escribir tanto la CPU del host como el código del dispositivo en el mismo archivo fuente de C++
- Esto requiere dos pases de compilación; uno para el código del host y otro para el código del dispositivo



# ¿Que es SYCL?

- SYCL proporciona abstracciones de alto nivel sobre el código repetitivo común
  - 1 Selección de plataforma/dispositivo
  - 2 Creación de búfer y movimiento de datos (USM)
  - 3 Compilación de funciones del kernel
  - 4 Gestión y programación de dependencias





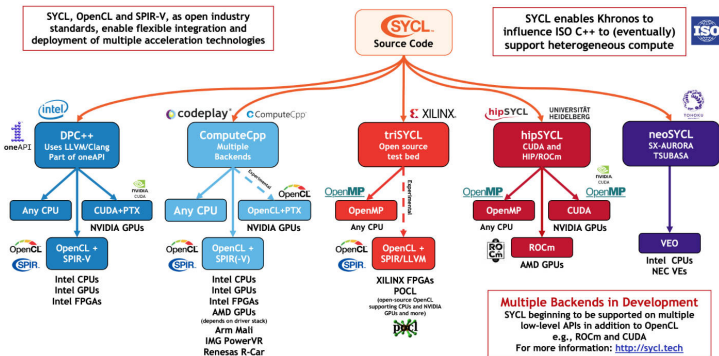
# SYCL

- SYCL le permite escribir C++ estándar
  - SYCL 2020 está basado en C++17
- A diferencia de las otras implementaciones que se muestran a la izquierda, hay:
  - Sin extensiones de idioma
  - Sin pragmas
  - Sin atributos

```
array<float> a, b, c;  
std::vector<float> a, b, c;  
  
//<2> idx) __attribute__((amp)) {  
#pragma parallel_for  
for(int i = 0; i < a.size(); i++) {  
    c[i]  
}  
  
__global__ vec_add(float *a, float *b, float *c) {  
    return c[i] = a[i] + b[i];  
}  
  
float *a, *b, *c;  
vec_add<<range>>>(a, b, c);  
  
cgh.parallel_for(range, [=](cl::sycl::id<2> idx) {  
    c[idx] = a[idx] + b[idx];  
});
```



# Implementaciones SYCL



# ¿Qué es Data parallel C++?

- Data Parallel C++  $\Leftrightarrow$  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

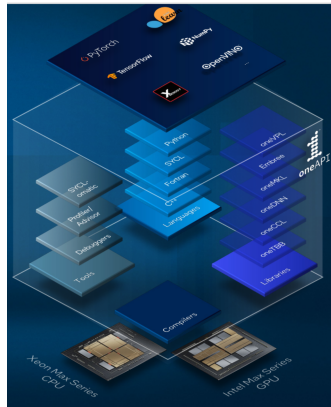


# Intel oneAPI

- Un lenguaje basado en estándares: C++ y SYCL
- Potentes API para acelerar funciones de dominio específico

## Soluciones a proveedor único

- Estándar abierto para promover el apoyo de la comunidad y la industria
- Permite la reutilización de código en diferentes arquitecturas y proveedores



# Tipos colas

- En SYCL, todo el trabajo se envía mediante los comandos a una cola
- La **cola** tiene un **dispositivo asociado** al que apuntarán todos los comandos en cola
  - En la **cola** se ejecutan una serie de comandos que se representan en la *command group*

type	Device
default_selector_v	Selects any device or host device if no device can be found.
gpu_selector_v	Select a GPU
accelerator_selector_v	Select an accelerator
cpu_selector_v	Select a CPU device
my_device_selector	Custom selector



# Ejemplo cola

## queue\_device.cpp

```
int main() {
    sycl::device d;

    d = sycl::device(sycl::gpu_selector());
    std::cout << "Using " << d.get_info<sycl::info::device::name>();

    sycl::queue Q(d);

    Q.submit([&](sycl::handler &cgh) {
        // Create a output stream
        sycl::stream sout(1024, 256, cgh);
        // Submit a unique task, using a lambda
        cgh.single_task( [=]() {
            sout << "Hello, World!" << sycl::endl;
        }); // End of the kernel function
    }); // End of the queue commands. The kernel is now submitted

    // wait for all queue submissions to complete
    Q.wait();
}
```



# Hands-on

- 1 Conéctate al Intel Developer Cloud
- 2 En el apartado de **Training and Workshops**
- 3 ... pero vamos a trabajar en **Essentials of SYCL**
  - <https://console.cloud.intel.com/training/detail/9b8933a6-b466-4c69-8217-9a09d084a55a>
- 4 Dentro del IDC, abrir el cuaderno de la ruta  
**Training/HPC/oneapi-essentials-training/01\_oneAPI\_Intro/oneAPI\_Intro.ipynb**



# Clases importantes en SYCL

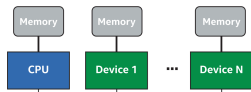
Clase	Funcionalidad
<code>sycl::device</code>	Representa el dispositivo: CPU, GPU, FPGA... donde se ejecuta el kernel SYCL
<code>sycl::queue</code>	Representa una cola donde el kernel se envía (encolar) Varias colas pueden mapear al mismo <code>sycl::device</code>
<code>sycl::buffer</code>	Encapsula una asignación de memoria que el tiempo de ejecución puede transferirse entre host y el dispositivo
<code>sycl::handler</code>	Manejador. Se utiliza para definir un ámbito del grupo de comandos. Conecta conceptos de buffers y kernels
<code>sycl::accessor</code>	Se utiliza para definir los requisitos de acceso de kernels (ie, lectura, escritura, lectura-escritura)
<code>sycl::range</code> , <code>sycl::nd_range</code> <code>sycl::id</code> , <code>sycl::item</code> , <code>sycl::nd_item</code>	Representa rangos de ejecución y paralelismo en ejecución



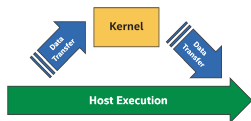


# Memoria

- El modelo simple de memoria indica que las memorias de los dispositivos y del host son separadas



- Los datos suelen ser transferidos a los memorias locales de los dispositivos para que sean accesibles desde los kernels



# Manejo de datos

- En SYCL existen dos modelos de memoria:

- 1 Buffer & Accessors
- 2 USM



# USM

- Es un mecanismo para manejo de la memoria gestionado por punteros como C/C++
- No se requieren *accessors* para el acceso de memoria
- Teniendo en cuenta el mecanismo de acceso a memoria virtual: esto significa que cada puntero devuelto en un *alloc* a USM puede ser utilizado desde el *host* o en el *device*
  - USM permite describir tres tipos de memoria alojada en: *host*, *device* o *shared*



# USM

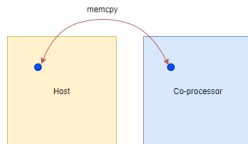
- USM permite describir tres tipos de memoria alojada en: *host*, *device* o *shared*

Tipo Asignación	Descripción	¿Accesible desde el host?	¿Accesible desde el dispositivo?	Localización
device	Asignada en memoria del dispositivo	NO	SI	device
host	Asignada en memoria del host	SI	SI	host
shared	Asignada entre mem. del host y device	SI	SI	puede migrar



# Manejo Explícito

- Manejo Explícito de memoria
  - Los datos son transferidos entre host-device de forma explícita
  - El runtime de SYCL no realiza ningún análisis de dependencias y el programador debe hacerlas explícitamente



# Manejo Explícito de memoria (USM)

- Manejo explícito con **memcpy** haciendo uso de USM

## explicit\_memory.cpp

```
int main() {
    queue q;
    int *data = static_cast<int *>(malloc(N * sizeof(int)));
    for (int i = 0; i < N; i++) data[i] = i;

    // Explicit USM allocation using malloc_device
    int *data_device = malloc_device<int>(N, q);

    // copy mem from host to device
    auto e1 = q.memcpy(data_device, data, sizeof(int) * N);

    // update device memory
    auto e2 = q.submit([&](handler &h) {
        h.depends_on(e1);
        h.parallel_for(range<1>(N), [=](id<1> i) { data_device[i] *= 2; });
    });

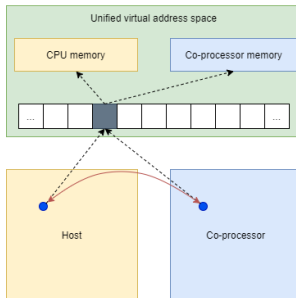
    // copy mem from device to host
    q.submit([&](handler &h) {
        h.depends_on(e2);
        h.memcpy(data, data_device, sizeof(int) * N);
    }).wait();

    //print output
    for (int i = 0; i < N; i++) std::cout << data[i] << std::endl;
    free(data_device, q);
    free(data);
    return 0;
}
```



# USM

- USM permite movimiento **implícito** de datos entre el *host* y *device* con asignación **shared**
  - **Simplifica portabilidad** en la codificación para aceleradores
- Proporciona al programador el nivel de control deseado
- Complementario al modelo con *buffers*



# USM

## usm\_memory.cpp

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

static const int N = 16;

int main() {
    queue q;

    // USM allocation using malloc_shared
    int *data = static_cast<int *>(malloc_shared(N * sizeof(int), q));

    // Initialize data array
    for (int i = 0; i < N; i++) data[i] = i;

    // Modify data array on device
    q.parallel_for({N}, [=](id<1> i) { data[i] *= 2; }).wait();

    // print output
    for (int i = 0; i < N; i++) std::cout << data[i] << std::endl;
    free(data, q);
    return 0;
}
```





# Hands-on

- 1 Conéctate al Intel Developer Cloud
- 2 En el apartado de **Training and Workshops**
- 3 ... pero vamos a trabajar en **Essentials of SYCL**
  - <https://console.cloud.intel.com/training/detail/9b8933a6-b466-4c69-8217-9a09d084a55a>
- 4 Dentro del IDC, abrir el cuaderno de la ruta  
**Training/HPC/oneapi-essentials-training/03\_SYCL\_Unified\_Shared\_Memory/Unified\_Shared\_Memory.ipynb**
- 5 Dentro del IDC, abrir el cuaderno de la ruta  
**Training/HPC/oneapi-essentials-training/09\_SYCL\_Buffers\_And\_Accessors\_In-depth/SYCL\_Buffers\_accessors.ipynb**



# Kernels

- Existen tres tipos de kernels en SYCL
  - **single\_task**: se ejecuta una única instancia del kernel
  - **parallel\_for**: se ejecuta tantas instancias como work-items (agrupados en work-groups)
  - **parallel\_for\_work\_group**: capacidad similar al NDRange de OpenCL (o *grid* en CUDA)
    - Puede ejecutarse el código de un work-group que se distribuye entre work-items
    - Está permitido el uso de memoria locales (equivalentes a *shared* de CUDA, o memorias *private*)



# 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(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* e *item*
  - *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

## matrix\_mult.cpp

```
h.parallel_for(range<2>(N, N), [=](id<2> item) {  
    auto i = item[0];  
    auto j = item[1];  
    // CODE THAT RUNS ON DEVICE  
    c[i*N+j] = 0.0f;  
    for(int k=0; k<N; k++)  
        c[i*N+j] += a[i*N+k]*b[k*N+j];  
    //    c[id] += a[id(j,k)]*b[id(k,i)]; // Equivalent  
}); // End of the kernel function
```



# Hands-on

- 1 Conéctate al Intel Developer Cloud
- 2 En el apartado de **Training and Workshops**
- 3 ... pero vamos a trabajar en **Essentials of SYCL**
  - <https://console.cloud.intel.com/training/detail/9b8933a6-b466-4c69-8217-9a09d084a55a>
- 4 Dentro del IDC, abrir el cuaderno de la ruta  
**Training/HPC/oneapi-essentials-training/02\_SYCL\_Program\_Structure/SYCL\_Program\_Structure.ipynb**



## Recursos disponibles

- Iniciativa oneAPI <https://www.oneapi.io/>
- Intel oneAPI Base & HPC Toolkit  
<https://www.intel.com/content/www/us/en/developer/tools/oneapi/commercial-base-hpc.html>
- **Nuevo** Libro 2023 *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL* disponible en [el link](#)
- Aprender SYCL en un hora, [webinar de James R Reinders](#)
- Instrucciones de [acceso del Intel Developer Cloud](#)

