



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Departament d'Arquitectura de Computadors

Tarjetas Gráficas y Aceleradores

OpenCL

Agustín Fernández

Departament d'Arquitectura de Computadors

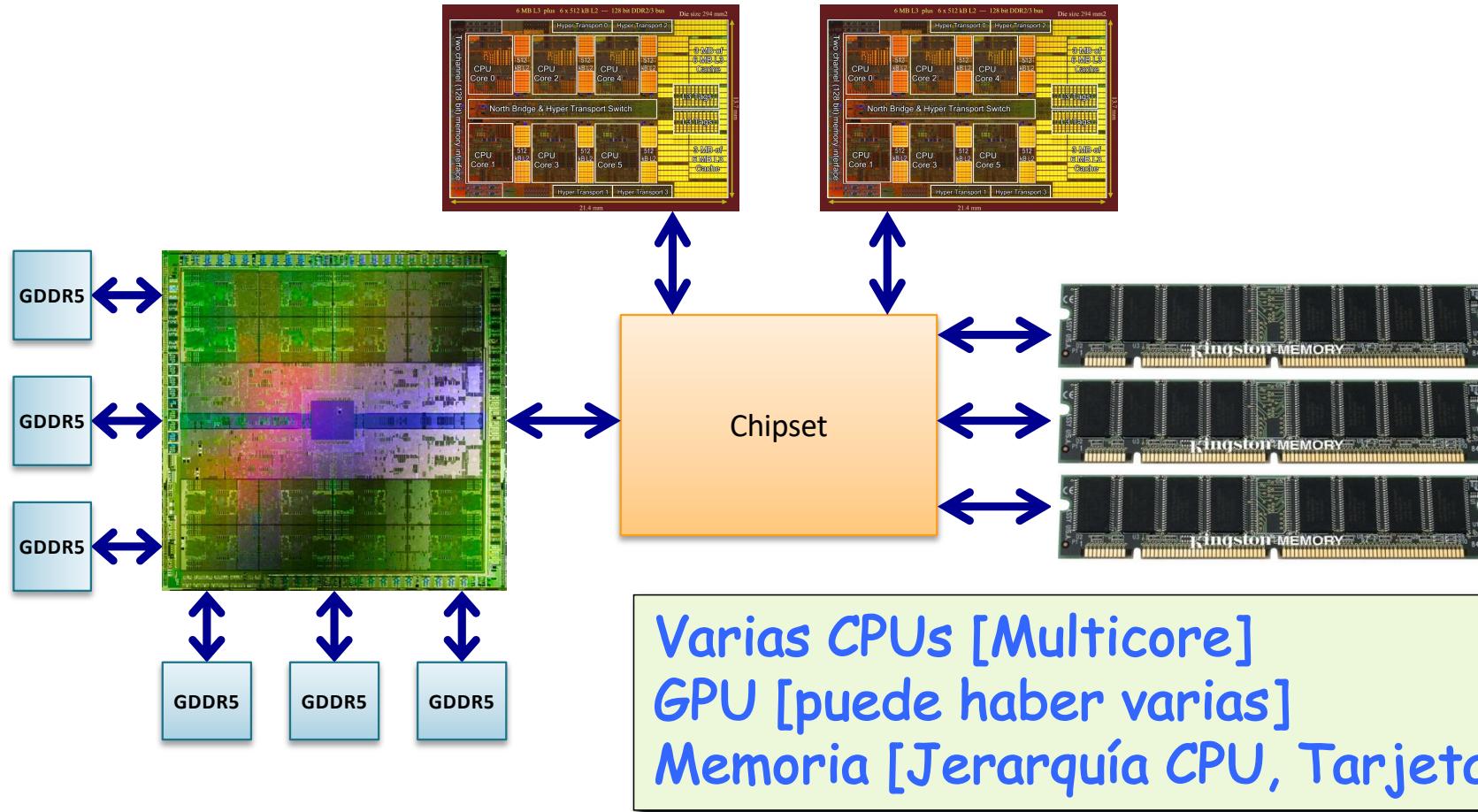
Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya



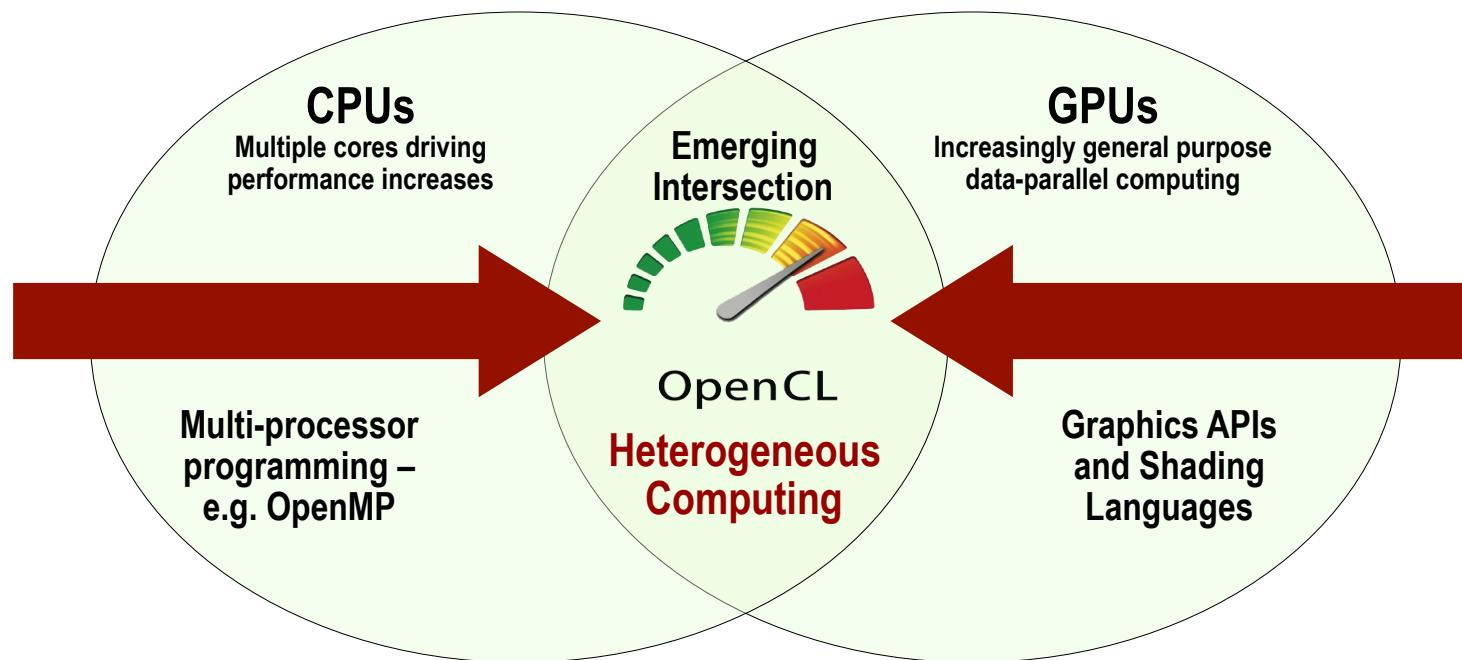
Introducción

El mundo de la computación es heterogéneo



Introducción

- OpenCL: Open Computing Language
- **Objetivo OpenCL:** permitir a los programadores escribir un único programa portable que use todos los recursos de una plataforma heterogénea.



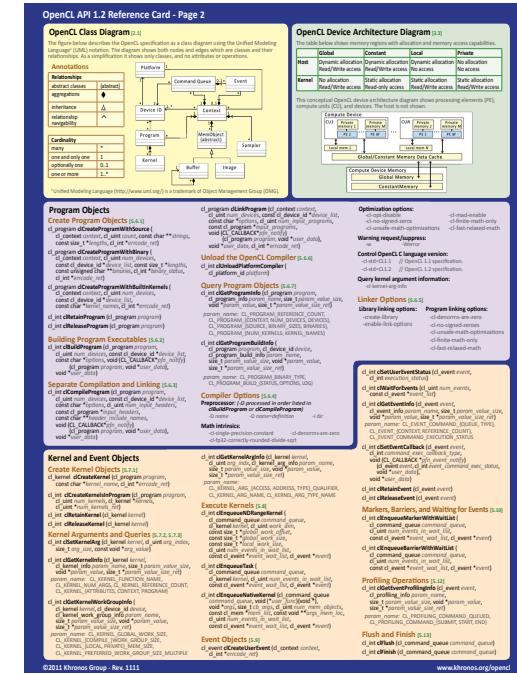
Introducción

- Iniciado por **Apple** con la colaboración técnica de **AMD/ATI, IBM, Intel** y **nVIDIA**.
- Apple lo propuso al grupo Khronos para convertirlo en un estándar abierto y libre de derechos.
- Todas las versiones de Mac OS disponen de OpenCL a través de Xcode.
- La primera definición de OpenCL apareció en Diciembre de 2008.
- En la definición de OpenCL hay una gran participación industrial



Introducción

- Especificación OpenCL 1.0 en diciembre 2008
 - Inmediatamente, AMD, nVIDIA, IBM anunciaron su intención de soportar completamente este estándar.
 - **Reference card:** www.khronos.org/files/opencl-quick-reference-card.pdf
- OpenCL 1.1 ratificada por Khronos en junio de 2010
 - Nuevos tipos de datos, p.e. vectores de 3 componentes
 - Zonas de datos 1D, 2D, 3D rectangulares
 - **Reference card:** www.khronos.org/files/opencl-1-1-quick-reference-card.pdf
- OpenCL 1.2 anunciada en noviembre 2011
 - **Reference card:** www.khronos.org/files/opencl-1-2-quick-reference-card.pdf
- OpenCL 2.0 anunciada en julio 2013
 - **Reference card:** www.khronos.org/files/opencl20-quick-reference-card.pdf
- OpenCL 2.1 anunciada en noviembre 2015
 - **Reference card:** www.khronos.org/files/opencl21-reference-guide.pdf
- OpenCL 2.2 especificación provisional (abril/2016)
 - **Reference card:** www.khronos.org/files/opencl22-reference-guide.pdf
- OpenCL 3.0, anunciada en abril de 2020 (publicada en noviembre 2020)
 - **Reference card:** www.khronos.org/files/opencl30-reference-guide.pdf



CUDA 1.0
apareció en
junio de 2007.

Khronos Group

- Consorcio sin ánimo de lucro encargado de la creación y desarrollo de estándares abiertos para computación paralela, gráficos, ...
- Todos los miembros pueden contribuir en el desarrollo de las APIs, decidir cómo han de ser y acelerar la llegada de las mismas a sus plataformas dado que disponen de toda la información antes de que se publique el estándar.
- Entre otros el grupo Khronos controla: OpenGL, OpenGL ES, OpenCL, Vulkan, WebGL, ...
- Participan en el consorcio:
 - Entidades académicas, acceso completo a la información, sin derecho a voto
 - Contribuyentes, todos los derechos y beneficios
 - Promotores, actúan como el consejo de administración con derecho de voto en la ratificación final de las especificaciones.



Khronos Group



accenture

ACRODEA



Adobe



AXELL CORPORATION



AMD



ARM

ERICSSON



P Powerhouse



COREAVI

codeplay



NOKIA
CONNECTING PEOPLE



GE Imagination

QUALCOMM

SAMSUNG
ELECTRONICS



TEXAS
INSTRUMENTS



dts

OMP

draw
Elements

TROPIC



FIXSTARS
Speed up your Business

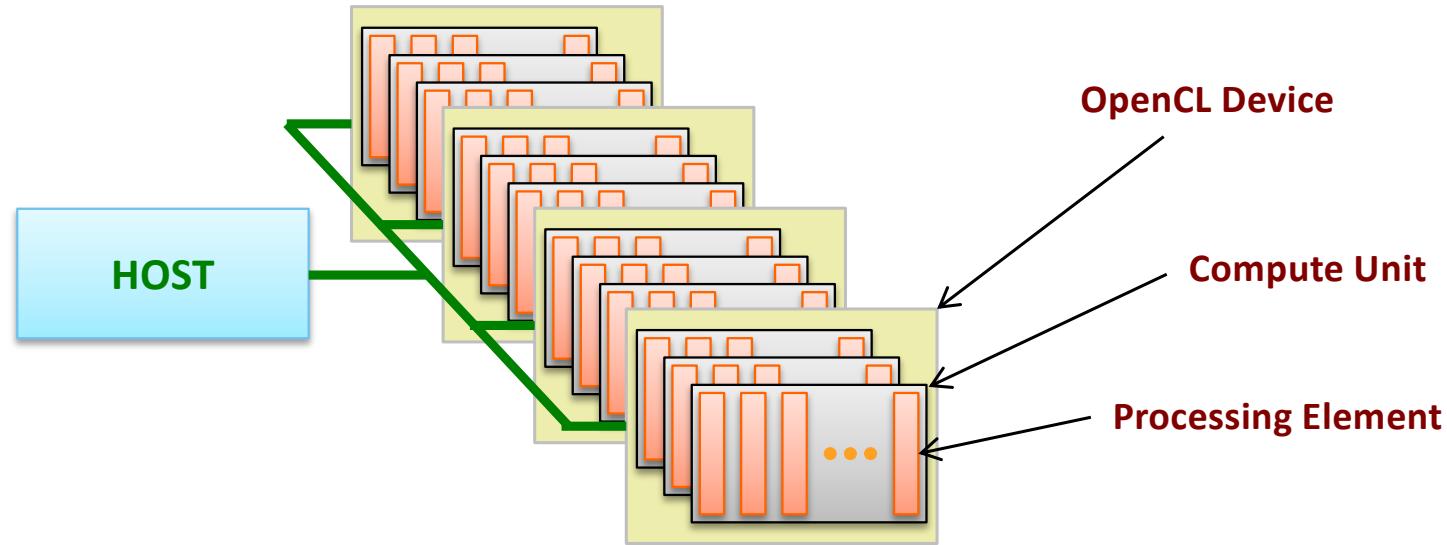
I REMARK

HISILICON

Google



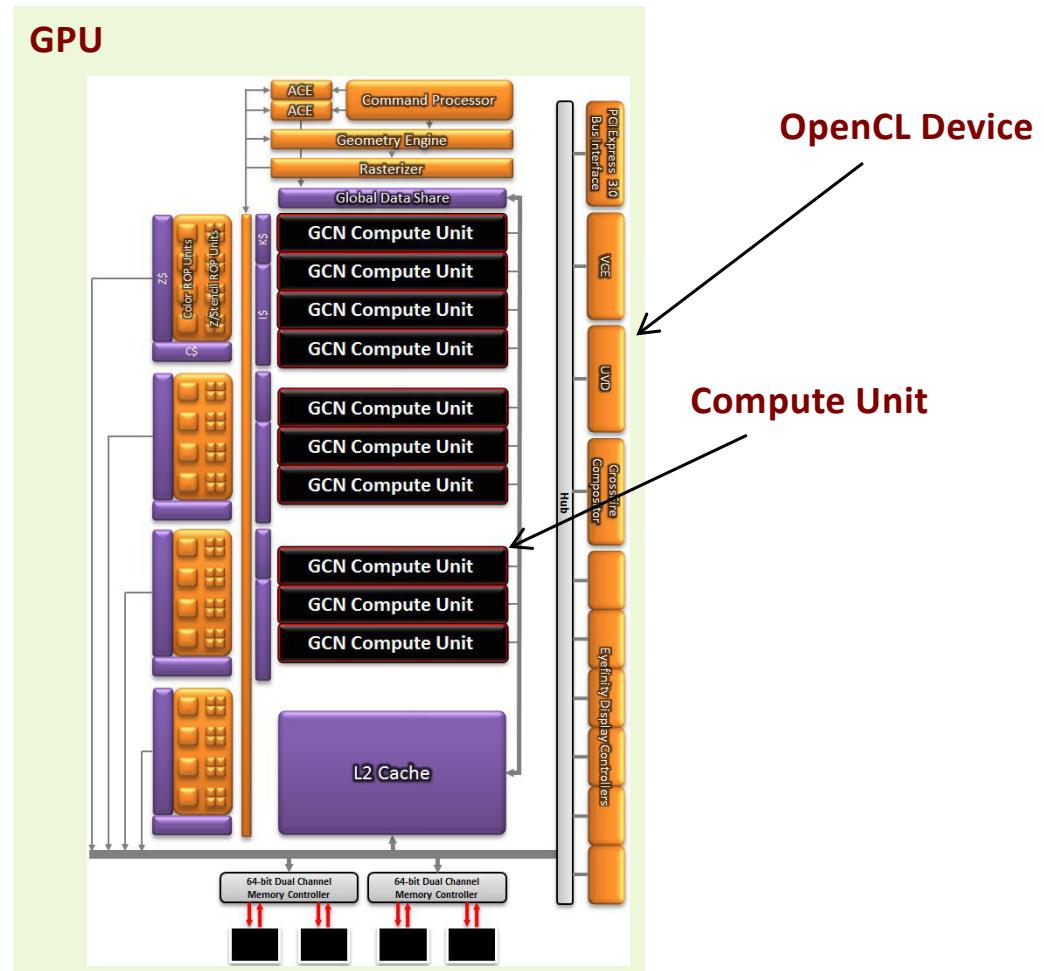
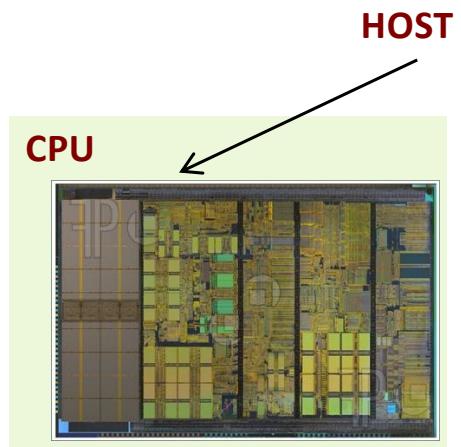
OpenCL Platform Model



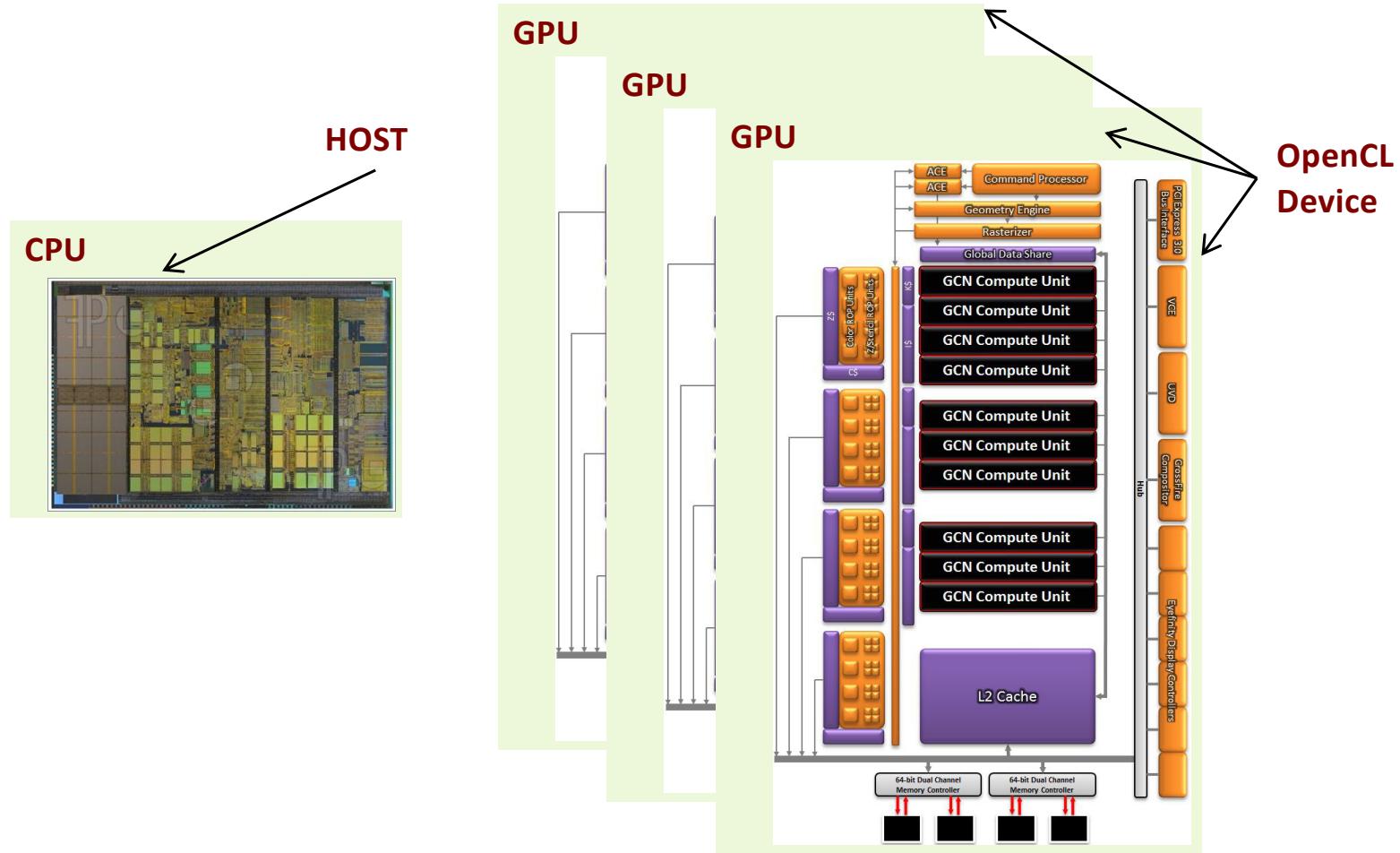
1 Host + 1 (o más) OpenCL Devices.
Cada OpenCL Device tiene 1 o más Compute Units.
Cada Compute Unit tiene 1 o más Processing Elements.
La Memoria está distribuida entre el host y los devices



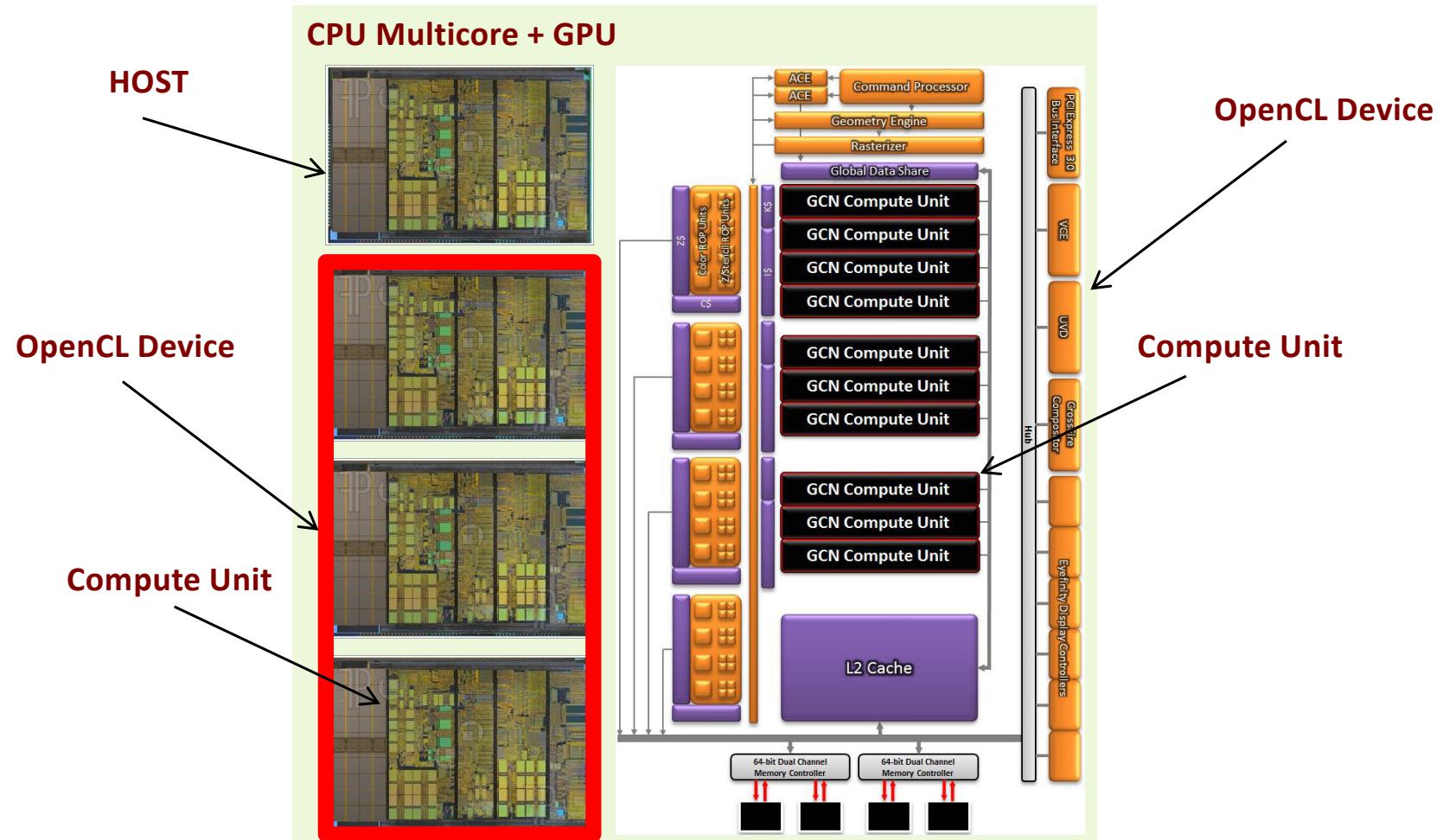
OpenCL Platform Model



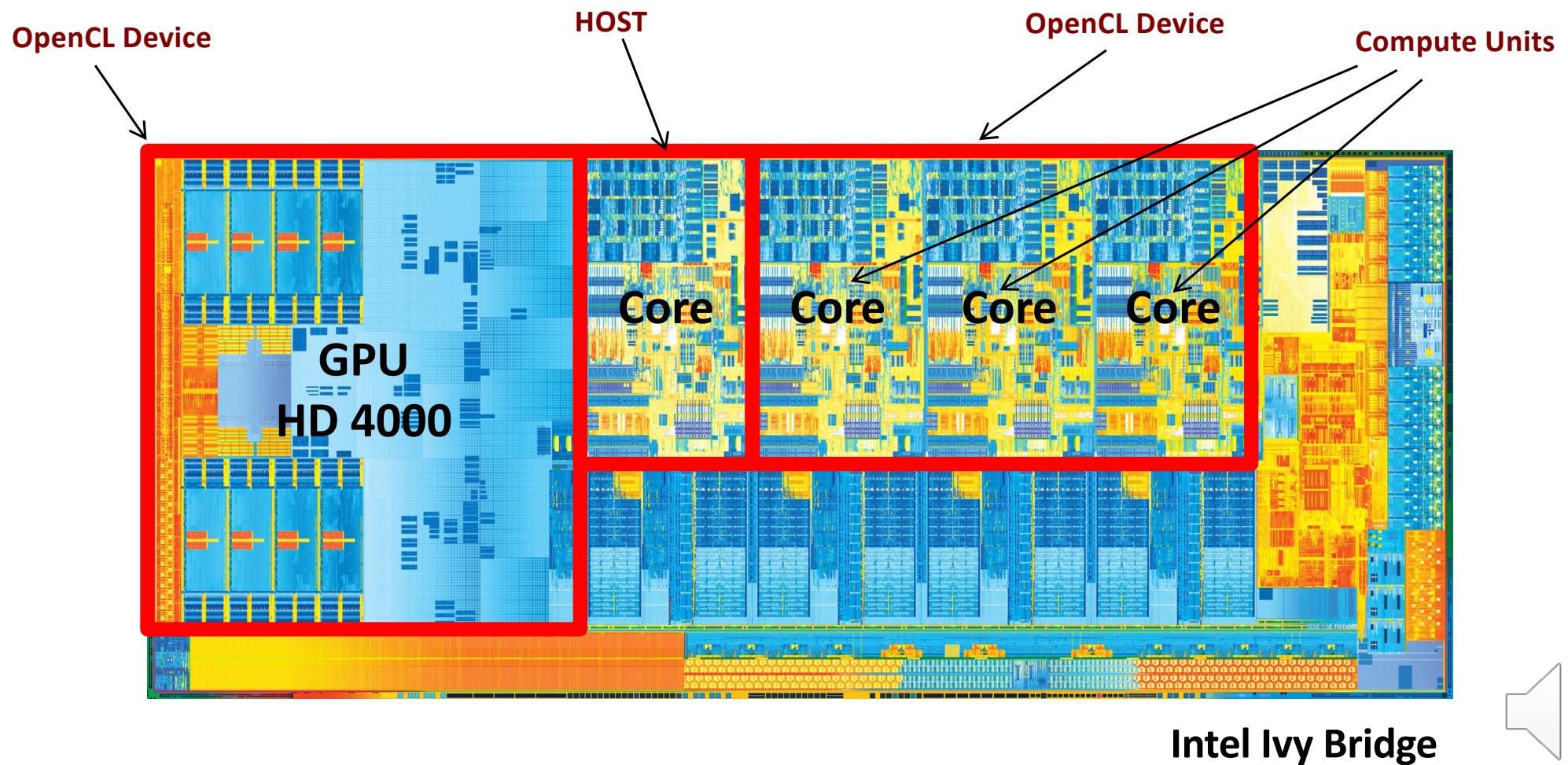
OpenCL Platform Model



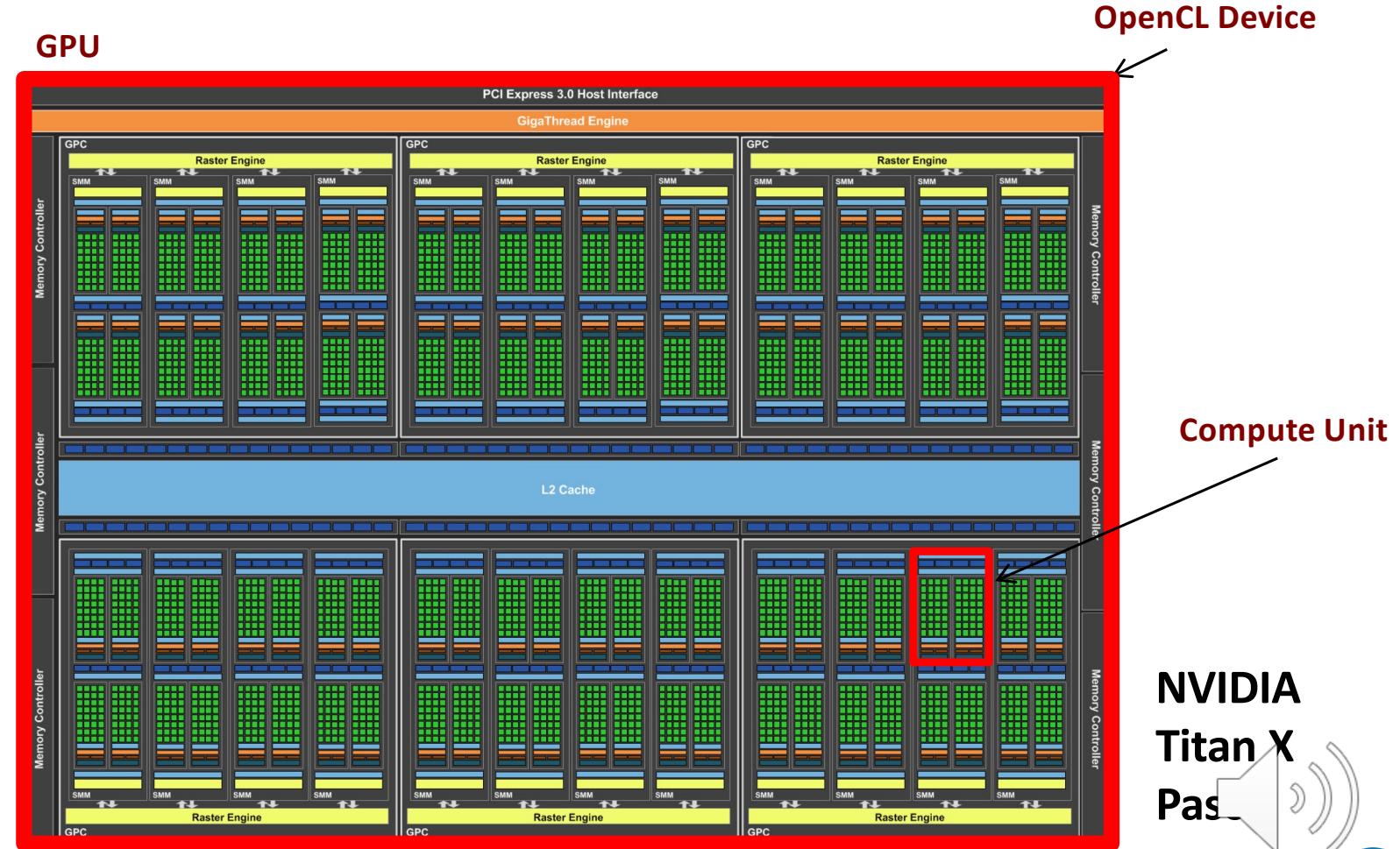
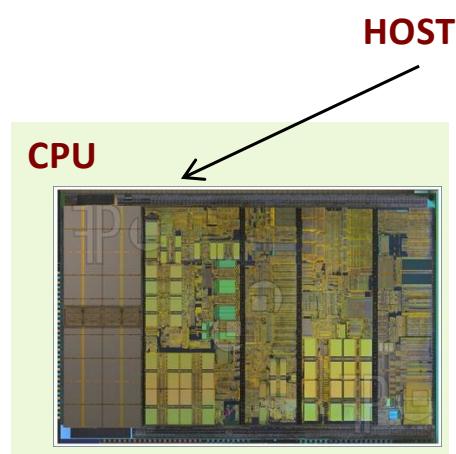
OpenCL Platform Model



OpenCL no implica AMD



OpenCL no implica AMD



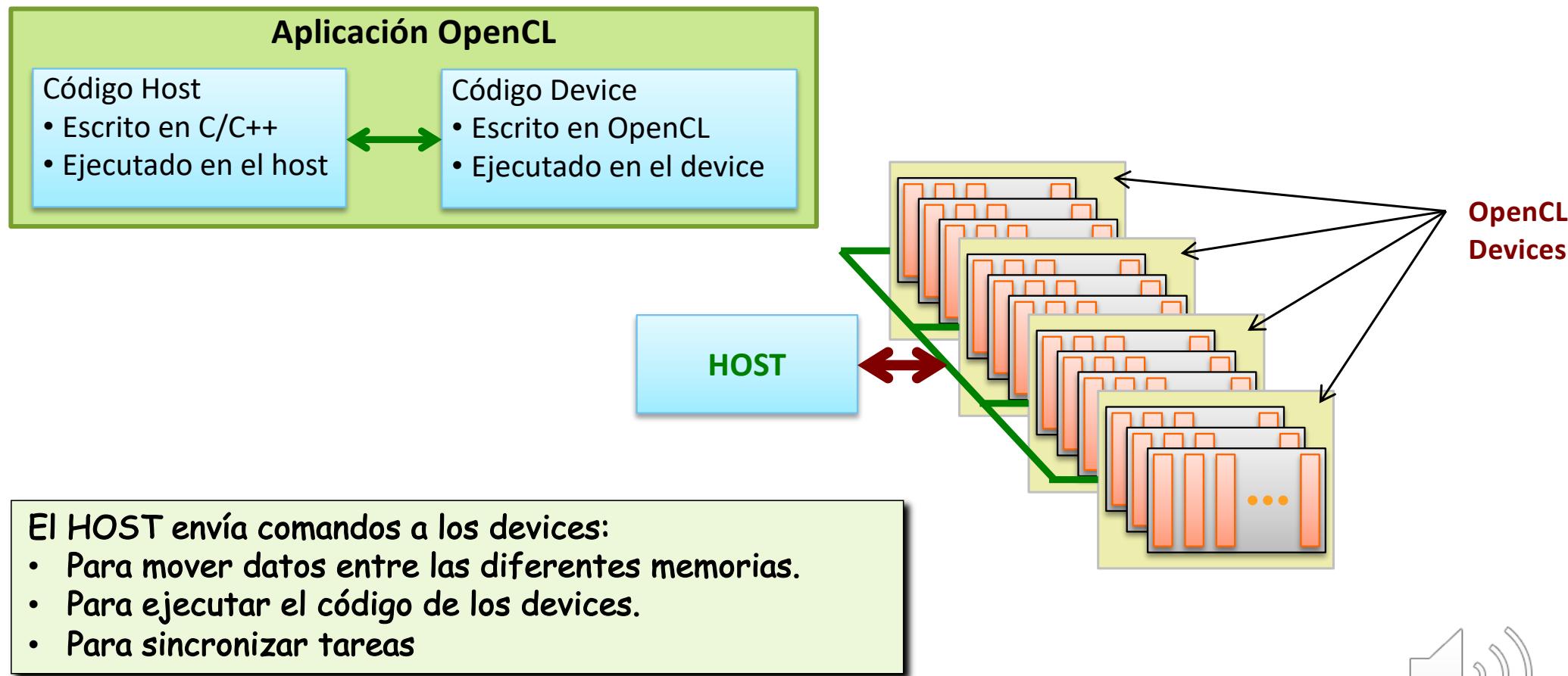
NVIDIA
Titan X
Pas
Speaker icon

Terminología OpenCL vs CUDA

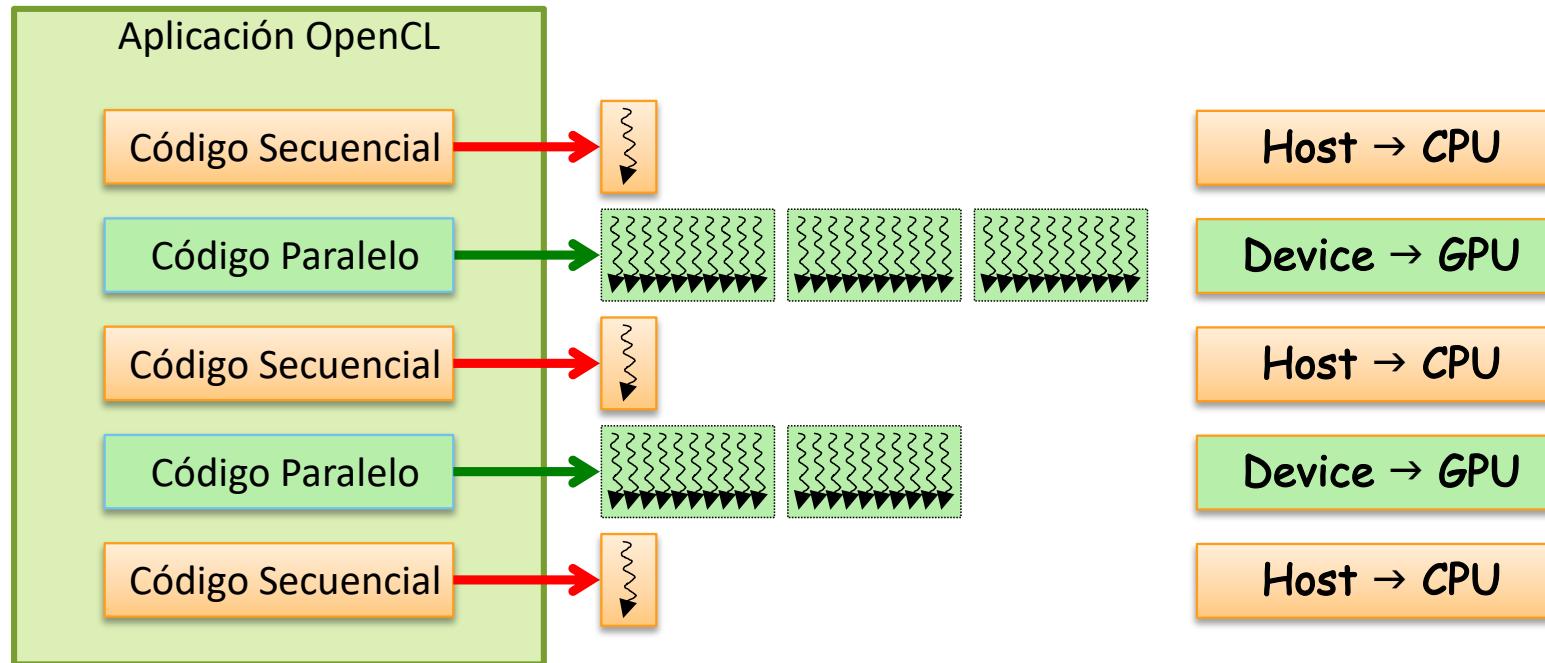
CUDA	OpenCL
GPU	Device (CPU, GPU, etc)
Multiprocessor	Compute Unit or CU
Scalar or CUDA core	Processing Element or PE
Global or Device Memory	Global Memory
Shared Memory (per block)	Local Memory (per workgroup)
Local Memory (registers)	Private Memory
Thread Block	WorkGroup
Thread	Work-item
Warp	-
Grid	NDRange



Anatomía de una aplicación OpenCL



Anatomía de una aplicación OpenCL



- El código secuencial se ejecuta en un thread del HOST (CPU).
- El código paralelo se ejecuta en un conjunto de threads que se ejecutan en los OpenCL devices en sus múltiples Processing Elements.



Anatomía de una aplicación OpenCL

La GRAN IDEA que hay detrás.

- Descomponer el trabajo en *work-items*
 - Definir un dominio computacional N-dimensional
 - Ejecutar un kernel en cada punto del dominio

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



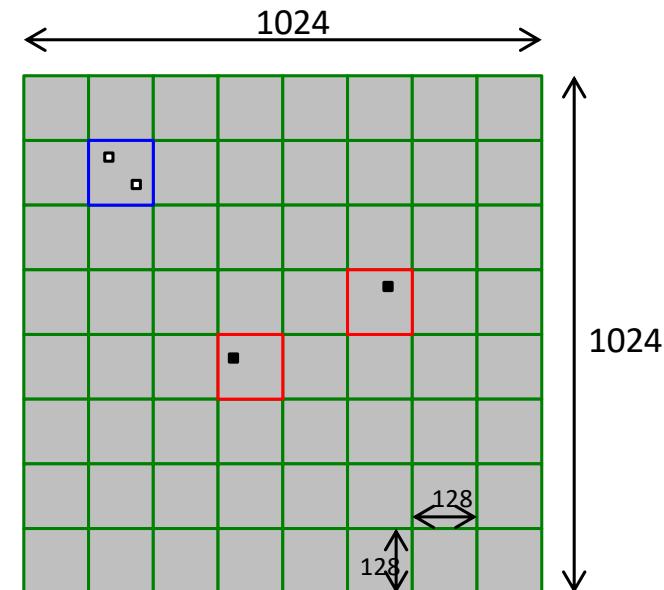
```
kernel void MUL(int n,
                 global const float *a,
                 global const float *b,
                 global float *c) {
    int id = get_global_id(0);
    if (id < n)
        c[id] = a[id] * b[id];
}
```

- El bucle es reemplazado por una función (kernel) que se ejecuta en cada punto del dominio.
- Por ejemplo, si definimos una matriz de 1024·1024 elementos y una invocación por cada elemento, tendremos 1.048.576 ejecuciones independientes del kernel



Dominio N-dimensional de *work-items*

- Decisión MUY IMPORTANTE:
 - definir el mejor dominio N-dimensional para nuestro algoritmo.
- Los kernels se ejecutan en un dominio global de *work-items*.
- Los *work-items* se agrupan en *work-groups*.
 - Dimensión global: $1024 \cdot 1024$
 - Dimensión local: $128 \cdot 128$ (se ejecutan juntos)
- La sincronización entre *work-items* sólo es posible dentro de un *work-group* (p.e. con barriers)
- No es posible sincronizar *work-items* de *work-groups* diferentes.
- Influye en la definición del dominio



Definición del problema

- Cualquier problema que queramos resolver con openCL ha de tener dimensiones
 - Por ejemplo, ejecutar un kernel para todos los elementos de una matriz de 2 dimensiones.
- Se asocia cada punto del espacio de iteraciones con un *work-item*
- Los *work-items* se agrupan en *work-groups*
 - Comparten memoria y pueden sincronizarse entre ellos
- Podemos especificar un determinado número de *work-items* en cada *work-group*
 - *Local (work-group) size*
- Cuando ejecutamos un kernel, podemos especificar hasta 3 dimensiones.
- Hay que especificar el tamaño del problema completo en todas sus dimensiones
 - *Global size*
- Esta descomposición / definición podemos dejar que la haga openCL en tiempo de ejecución (el rendimiento no suele ser el mejor).



Modelo de Memoria de OpenCL

□ Memoria Privada

- *Work-item*

□ Memoria Local

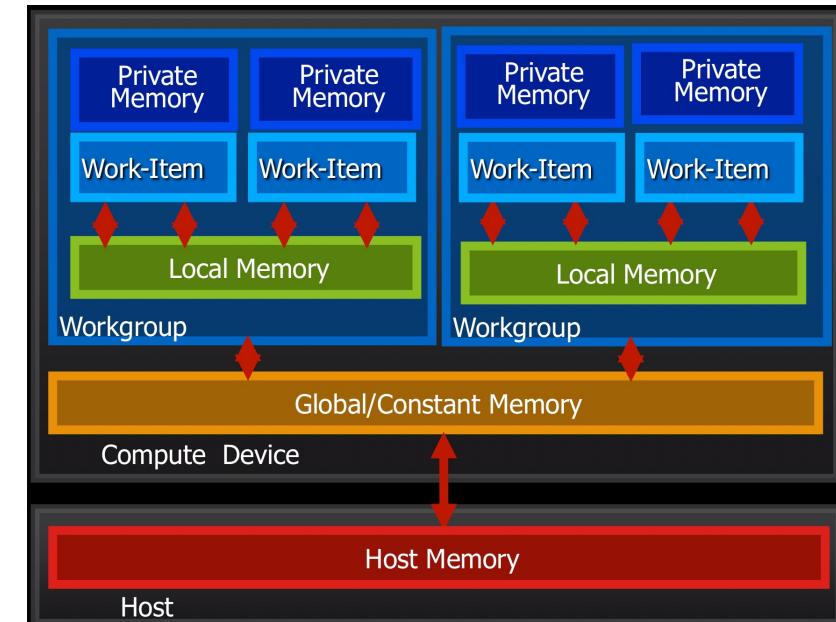
- Compartida por un *work-group*

□ Memoria Global/Constantes

- Visible por todos los *work-groups*

□ Memoria Host

- En la CPU



© Krhinos Group

- La gestión de memoria es explícita
- El programador es el encargado de mover la información entre host, global y local
- Una adecuada gestión de la memoria es fundamental para obtener buenos rendimientos.



Modelo de Memoria de OpenCL

- Es una jerarquía de memoria, con gestión explícita

Tipo	Uso	Tamaño	Ancho Banda	Coste de 1 acceso
Memoria Privada	Work-item	10's palabras	Para un w-i 2-3-4 registros / ciclo	1 ciclo
Memoria Local	Work-group	10-100's KBytes	Para un w-g 10's words / ciclo	Pocos ciclos
Memoria Global	Kernel	GBytes	100's GB/s	100's ciclos
Memoria Host	Programa	GBytes	10's GB/s	-



Modelo de Memoria de OpenCL

□ Memoria Privada

- Recurso muy escaso
- Unas pocas decenas de palabras de 32bits
- Equivalen a registros del procesador
- El uso excesivo de este recurso limitará el número y eficiencia de los work-items.

□ Memoria Local

- Recurso escaso.
- Varios work-groups pueden compartir la memoria local de una misma compute unit.
- IMPORTANTE: almacenar en esta memoria datos que comparten varios work-items.
- Los patrones de acceso influyen en el rendimiento: conflicto en los bancos, ...

□ Memoria Global

- OpenCL usa un modelo de consistencia relajado
- No está garantizado que la memoria visible por un work-group sea consistente para todos los work-groups en cualquier momento.
- Sólo está garantizada la consistencia para los work-items de un work-group en un barrier.
- La consistencia entre comandos se consigue de forma explícita mediante sincronización.



Terminología OpenCL vs CUDA

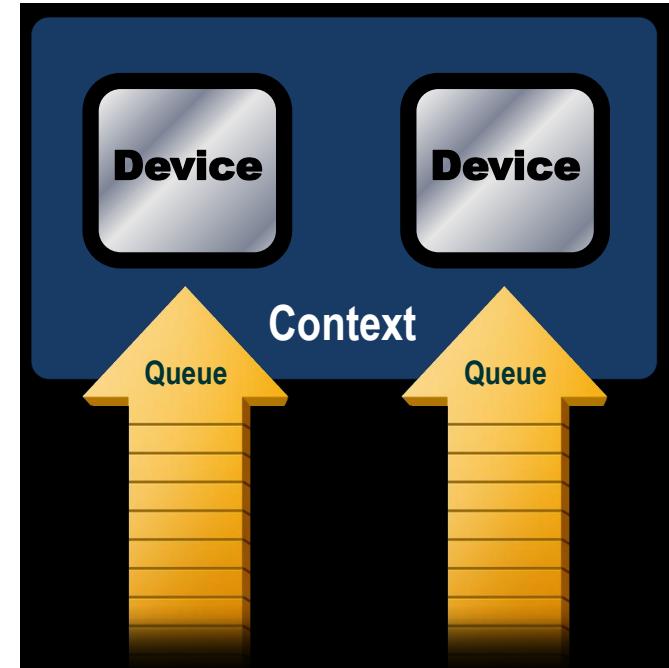
CUDA	OpenCL
GPU	Device (CPU, GPU, etc)
Multiprocessor	Compute Unit or CU
Scalar or CUDA core	Processing Element or PE
Global or Device Memory	Global Memory
Shared Memory (per block)	Local Memory (per workgroup)
Local Memory (registers)	Private Memory
Thread Block	WorkGroup
Thread	Work-item
Warp	-
Grid	NDRange



Modelo Ejecución OpenCL

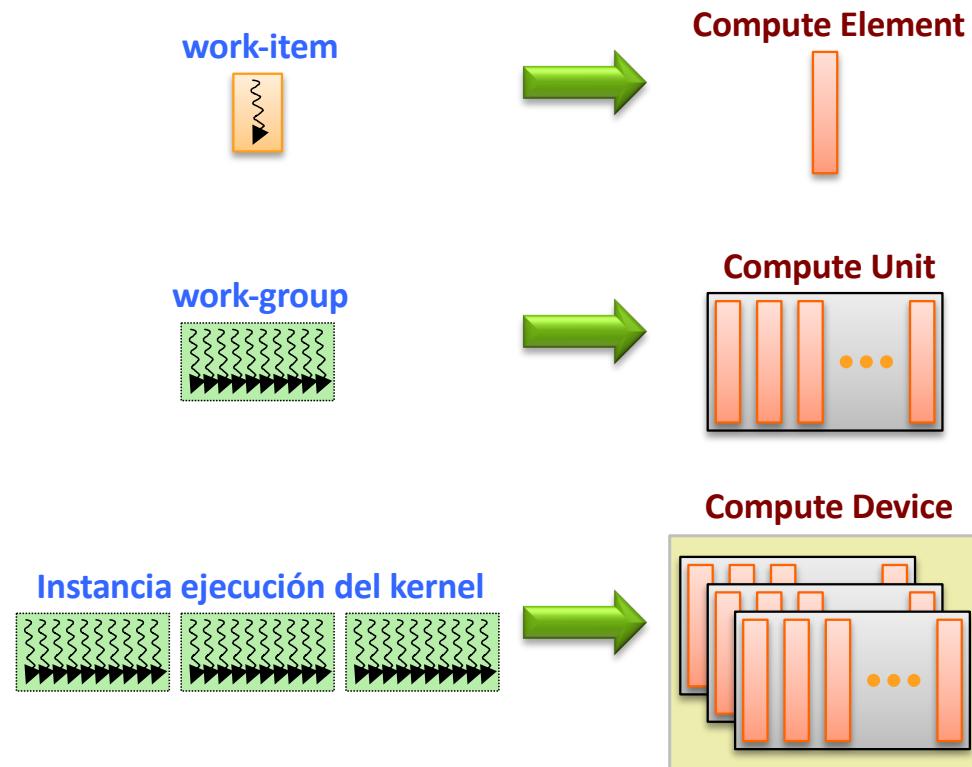
La aplicación corre en un Host que envía “trabajo” a los Devices

- Context**. Es el entorno en el que se ejecutan los *work-items*, se gestiona la memoria y la sincronización. Incluye los devices, su memorias y las colas de comandos (*command queues*).
- Command Queues**. Una cola que utiliza la aplicación que corre en el Host para enviar “trabajos” a un Device (p.e. Las instancias de un kernel).
 - Las tareas se encolan en orden
 - Una cola por device
 - Las tareas se pueden ejecutar en orden o en desorden.
 - Todas las tareas que se ejecutan en un device se reciben a través de la command-queue (ejecución de un kernel, sincronización y operaciones con memoria).



© Krhinos Group

Ejecución de un kernel



- Cada work-item se ejecuta en un Compute Element
- Cada work-group se ejecuta en una Compute Unit.
- Puede haber varios workgroups concurrentes, dependiendo de los requerimientos de memoria y los recursos disponibles
- Cada kernel se ejecuta en un Compute Device



Construir el Program Object

- Un program object encapsula:

- Un contexto
 - El código fuente o binario
 - Lista de devices y opciones

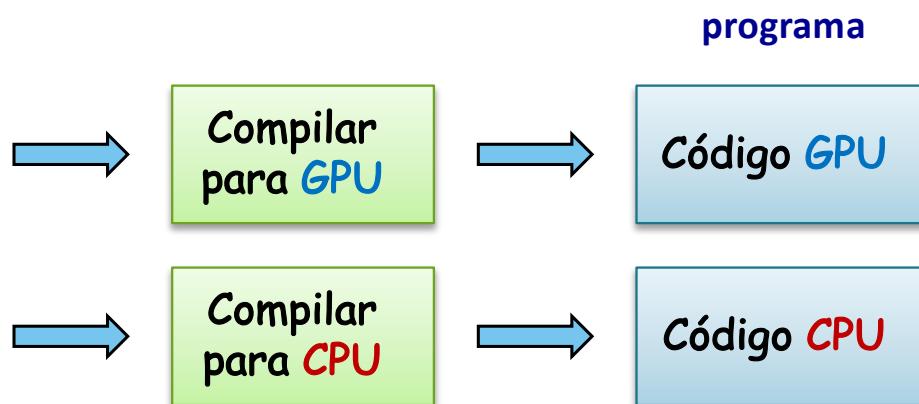
OpenCL usa runtime compilation

- Para crear el program object

- clCreateProgramWithSource ()
 - clCreateProgramWithBinary ()

```
kernel void mul(int n,
    global const float *a,
    global const float *b,
    global float *c) {
    int id = get_global_id(0);
    if (id <n)
        c[id] = a[id] * b[id];
}
```

kernel code



Ejemplo OpenCL

- Ejemplo MUY SENCILLO: producto de dos vectores elemento a elemento

```
float a[N], b[N], c[N];
for (int i=0; i<N; i++)
    c[i] = a[i] * b[i];
```

- Hay que definir:

- Código del KERNEL
 - Código del HOST



Ejemplo OpenCL

```
float a[N], b[N], c[N];
for (int i=0; i<N; i++)
    c[i] = a[i] * b[i];
```

Código Secuencial

Describe **TODO** lo que hay que hacer.

```
kernel void mul(global const float *a,
                global const float *b,
                global float *c
                global const uint N) {
```

```
    int id = get_global_id(0);
    if (id < N)
        c[id] = a[id] * b[id];
}
```

Kernel OpenCL

Indica **quién** soy.

Describe cómo se calcula **1 elemento** del vector: $c[id]$. Es lo que se ejecuta en un work-item

Kernel OpenCL vs Kernel CUDA

```
__global__
void mul(float *a, float *b, float *c, int N) {
    int id = blockIdx.x*blockDim.x + threadIdx.x;
    if (i<N) c[id] = a[id] * b[id];
}
```

Kernel CUDA

Describe cómo se calcula **1 elemento** del vector: $c[id]$. Es lo que se ejecuta en un thread.

```
kernel void mul(global const float *a,
                global const float *b,
                global float *c
                global const uint N) {

    int id = get_global_id(0);
    if (id < N)
        c[id] = a[id] * b[id];
}
```

Kernel OpenCL

Indica **quién soy**.

Describe cómo se calcula **1 elemento** del vector: $c[id]$. Es lo que se ejecuta en un work-item.

Ejemplo OpenCL

□ El programa que corre en el host ha de:

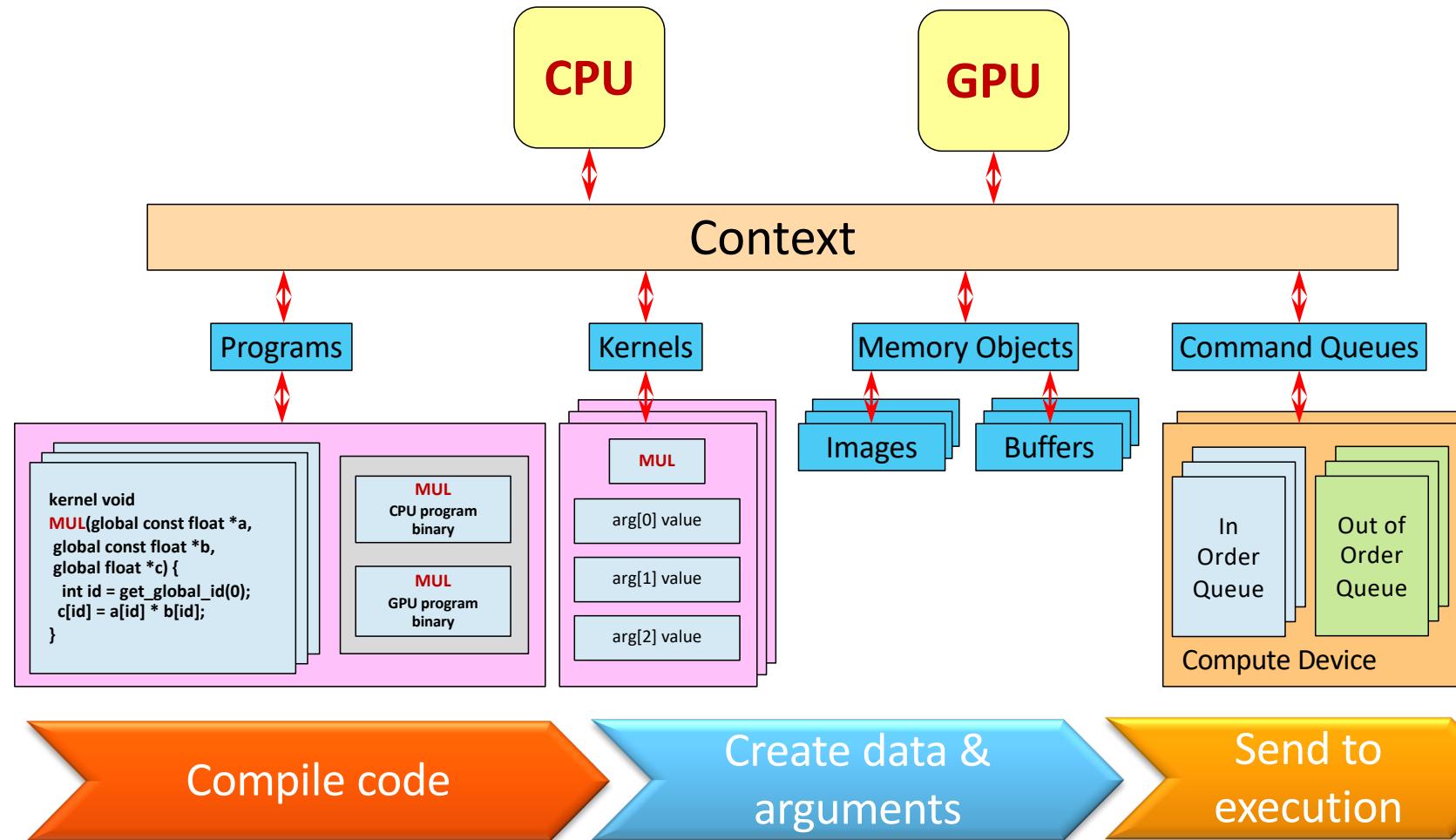
- Inicializar el entorno para un programa OpenCL
- Crear y gestionar los kernels

□ **Programa básico del host en 5 pasos:**

1. Definir la plataforma (plataforma = devices + context + queues)
2. Crear y Construir el programa (biblioteca dinámica con los kernels)
3. Inicializar los memory objects
4. Definir el kernel (asociar argumentos a la función kernel)
5. Enviar los comandos, transferir los memory objects y ejecutar los kernels.

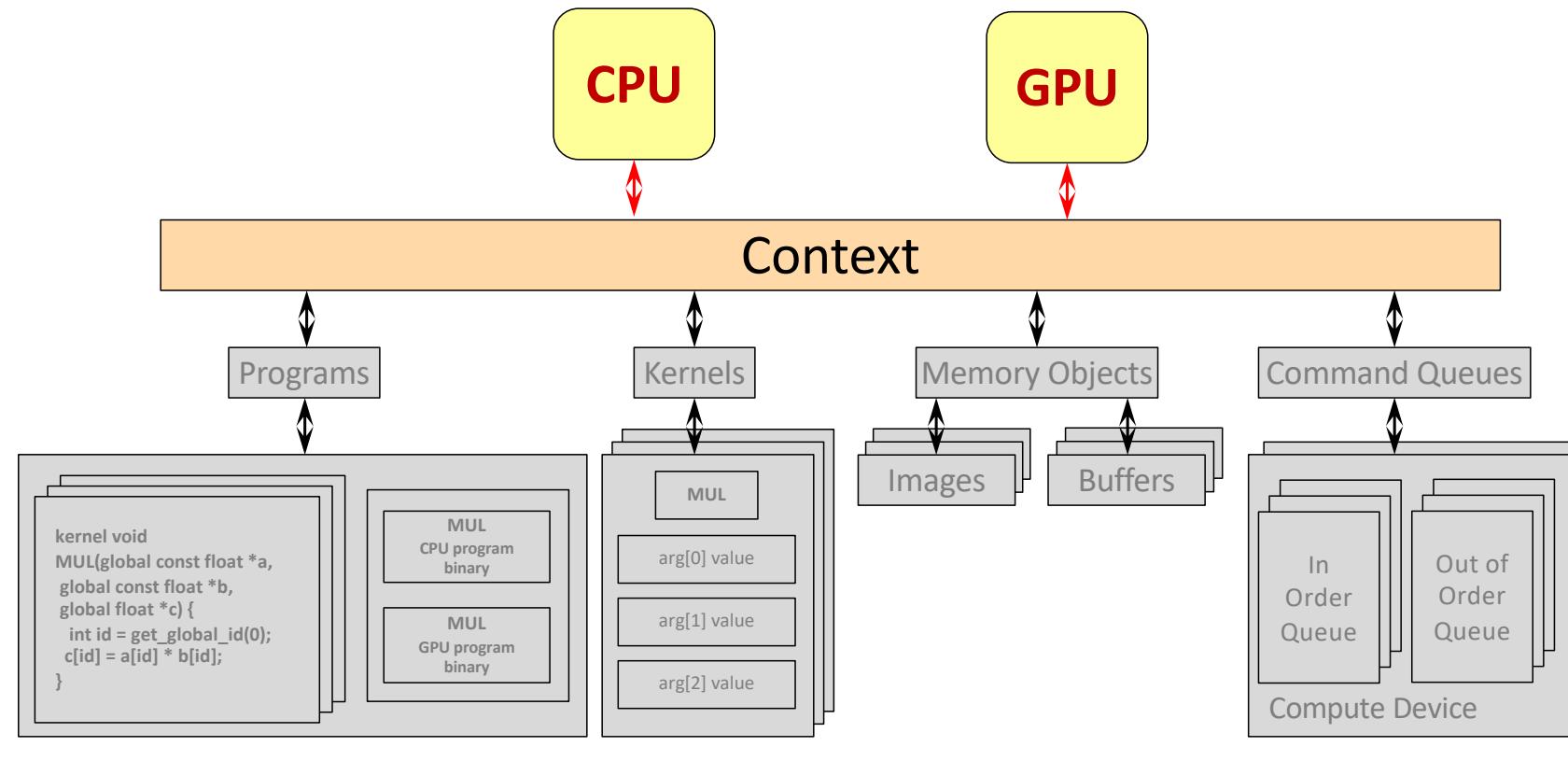


OpenCL Framework



© Khronos Group

OpenCL Framework: Platform Layer



Compile code

Create data & arguments

Send to execution



© Khronos Group

Ejemplo OpenCL

1. Definir la plataforma

- Obtener la primera plataforma disponible

```
err = clGetPlatformIDs(1, &cpPlatf, NULL);
```

```
CL_DEVICE_TYPE_CPU  
CL_DEVICE_TYPE_GPU  
CL_DEVICE_TYPE_ACCELERATOR  
. . .
```

- Obtener un device apropiado

```
err = clGetDeviceIDs(cpPlatf, devType, 1, &device_id, NULL);
```

- Crear un contexto apropiado

```
context = clCreateContext(cpPlatf, 1, &device_id, NULL, NULL, &err);
```

- Crear command queue

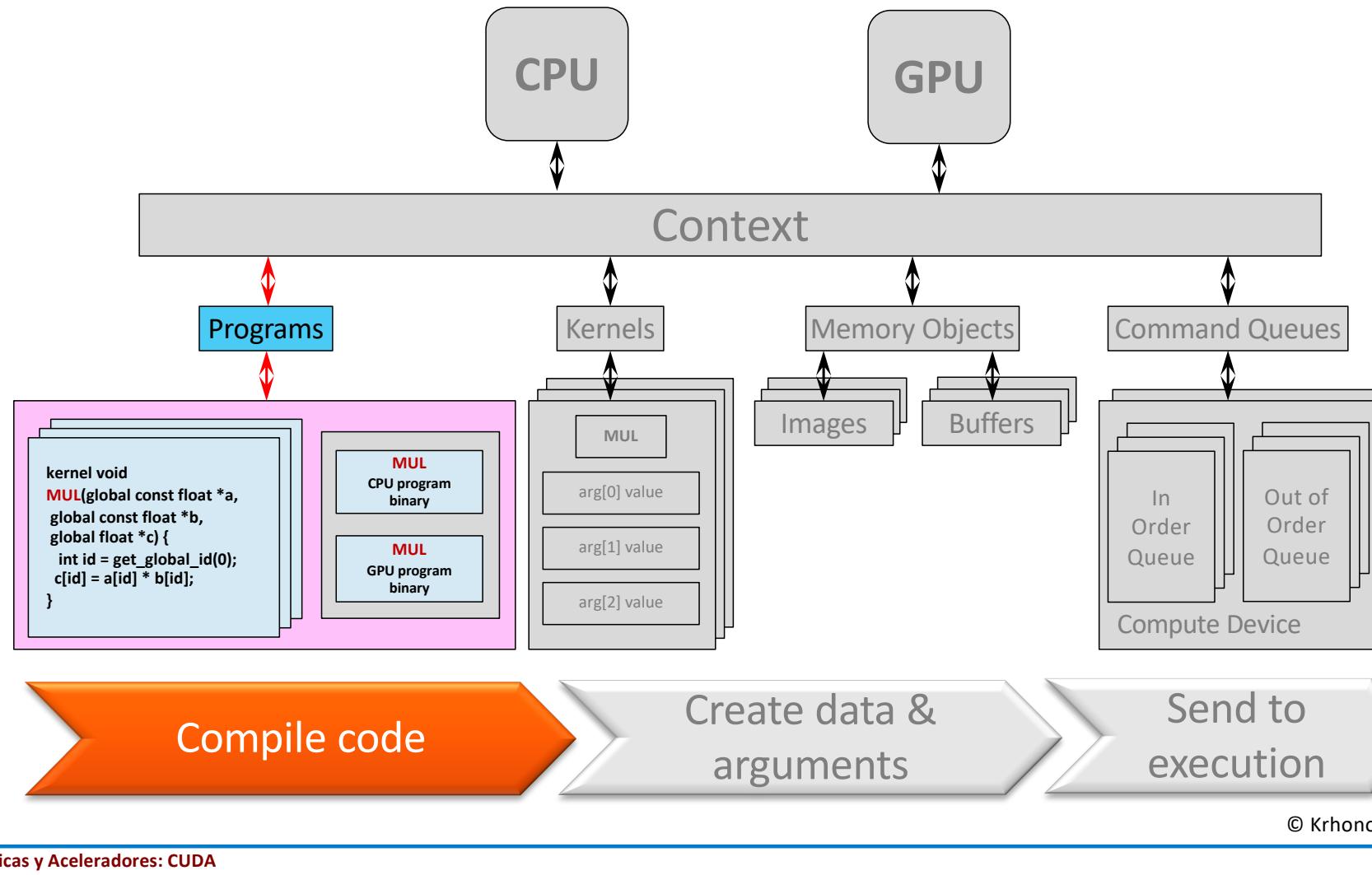
```
commands = clCreateCommandQueue(context, device_id, 0, &err);
```

```
err = clGetPlatformIDs(1, &cpPlatform, NULL);
if (err != CL_SUCCESS) {
    cerr << "Error: Failed to find a platform!" << endl;
    return EXIT_FAILURE;
}
```

El control de errores es imprescindible



OpenCL Framework: Program Compile



Ejemplo OpenCL

2. Crear y definir el programa

- Definir el programa fuente para el kernel, ya sea a partir de un string de caracteres (programas muy simples) o bien leído de un fichero (aplicaciones reales).

- Crear el programa almacenado en un buffer

```
program = clCreateProgramWithSource(context, 1,
                                     (const char **) &KernelSource, NULL, &err);
```

- Compilar el programa. Se crea una “dynamic library” de dónde se obtendrá el kernel

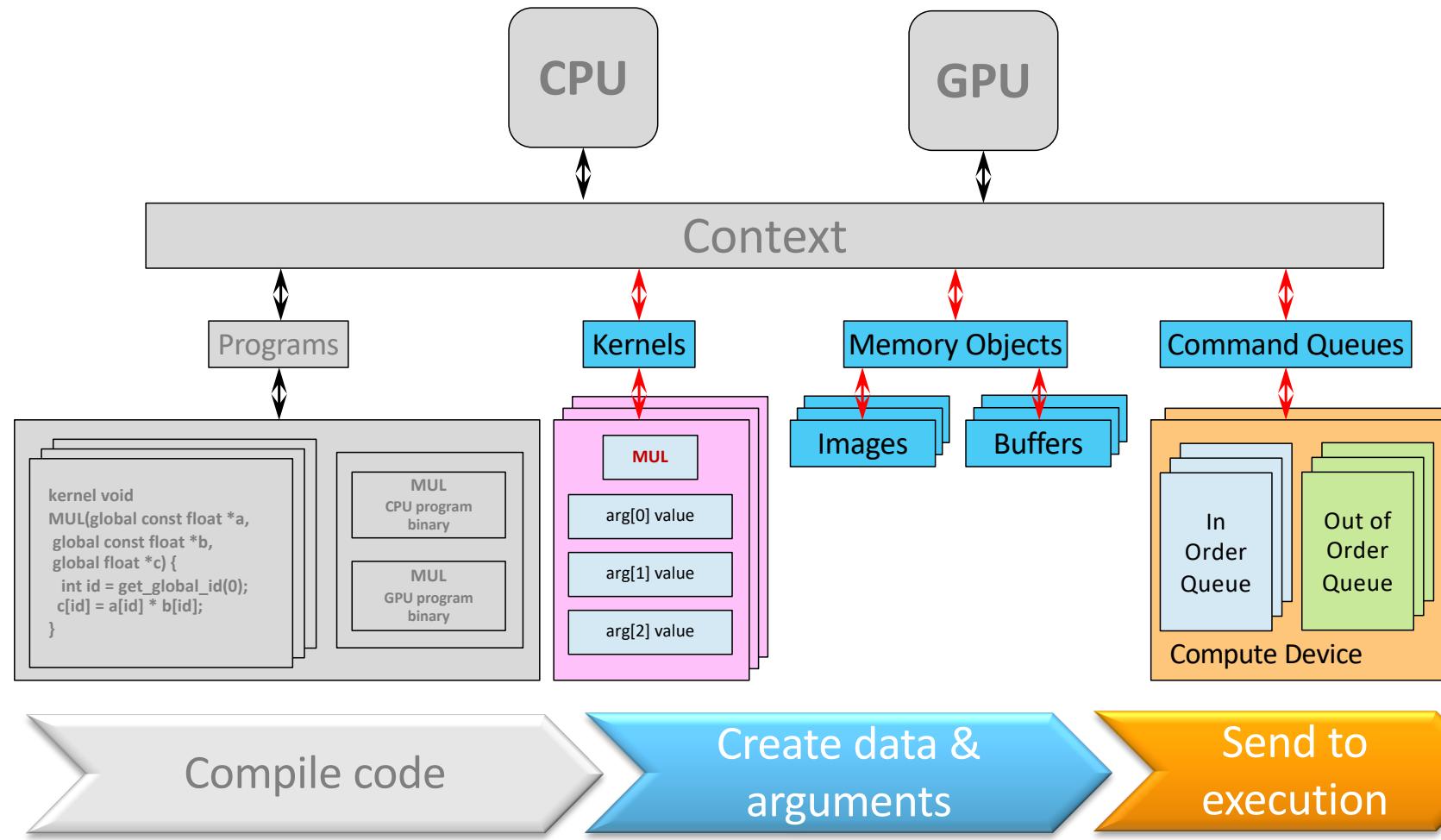
```
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
```

- Si hay errores de compilación, hay que volcar los errores

```
if (err != CL_SUCCESS) {
    size_t len; char buffer[2048];
    clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG,
    sizeof(buffer), buffer, &len);
    cerr << buffer << endl;
    exit(1);}
```



OpenCL Framework: Runtime



© Khronos Group

Ejemplo OpenCL

3. Inicializar los memory objects

- Los datos de entrada se han de leer de un fichero o inicializar por programa

```
for(int i=0; i<count; i++) {  
    h_a[i] = rand() / (float)RAND_MAX;  
    h_b[i] = rand() / (float)RAND_MAX;  
}
```

Memoria del Host

- Definir los memory objects de OpenCL

```
tam = sizeof(float) * count;  
d_a = clCreateBuffer(context, CL_MEM_READ_ONLY, tam, NULL, NULL);  
d_b = clCreateBuffer(context, CL_MEM_READ_ONLY, tam, NULL, NULL);  
d_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY, tam, NULL, NULL);
```

Memoria global del device



¿Cómo usamos la memoria?

□ Usamos Buffer Objects

○ Buffer Object

✓ Se pueden ver como vectores de bytes. Accesibles por los kernels usando punteros / vectores

○ Image Objects (Son la interfaz con OpenGL [No veremos nada])

□ Declaración de un Buffer Object

```
cl_mem d_a, d_c; // datos en el device
```

□ Declaración de los datos originales en el host (tamaño y tipo)

```
float h_a[N], h_c[N];
```

CL_MEM_WRITE_ONLY
CL_MEM_READ_WRITE

□ Crear el buffer_object (d_a), asignarle los datos del host (h_a) y copiarlos en la memoria del device

```
cl_mem d_a = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,  
                           sizeof(float)*N, h_a, NULL);
```

□ Envío del comando para recuperar el resultado desde d_c

```
clEnqueueReadBuffer(commands, d_c, CL_TRUE, 0, sizeof(float)*N, h_c,  
                     NULL, NULL, NULL);
```

Blocking



Ejemplo OpenCL

4. Definir el kernel

- Crear el objeto kernel a partir de la función kernel mul.

```
kernel = clCreateKernel(program, "mul", &err);
```

- Asociar los argumentos de la función a los memory objects

```
tam = count;  
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);  
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);  
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);  
err |= clSetKernelArg(kernel, 3, sizeof(uint), &tam);
```

```
kernel void mul(global const float *a,  
                global const float *b,  
                global float *c  
                global const uint N) {  
  
    int id = get_global_id(0);  
    if (id < N)  
        c[id] = a[id] * b[id]; }
```



Ejemplo OpenCL

5. Enviar los comandos, transferir los memory objects y ejecutar los kernels.

- Escribir los memory buffers en la memoria global (operaciones non-blocking)

```
err = clEnqueueWriteBuffer(commands, d_a, CL_FALSE, tam, h_a, 0, NULL, NULL);  
err = clEnqueueWriteBuffer(commands, d_b, CL_FALSE, tam, h_b, 0, NULL, NULL);
```

- Encolar el kernel, para su ejecución (cola en orden)

```
err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL, &global, &local,  
0, NULL, NULL);
```

- Obtener el resultado (blocking). Como la cola es en orden sabemos que los comandos previos han acabado.

```
err = clEnqueueReadBuffer(commands, d_c, CL_TRUE, 0, tam, h_c, 0, NULL, NULL );
```



Ejemplo OpenCL

```
clEnqueueNDRangeKernel(-, -, work_dim, *global_work_offset,  
                      *global_work_size, *local_work_size, -, -, -);
```

- En esta rutina se define el tamaño del problema, la dimensionalidad del mismo y el tamaño de cada dimensión. Equivale a la invocación de CUDA.
- En definitiva, se define el tamaño del workgroup.
- Con estos datos se calculará el ID de cada work-item, para acceder a los datos.
- **work_dim**: dimensionalidad (1, 2 o 3)
- ***global_work_offset**: vector de desplazamientos
- ***global_work_size**: vector con las dimensiones globales
- ***local_work_size**: vector con las dimensiones locales (workgroup)

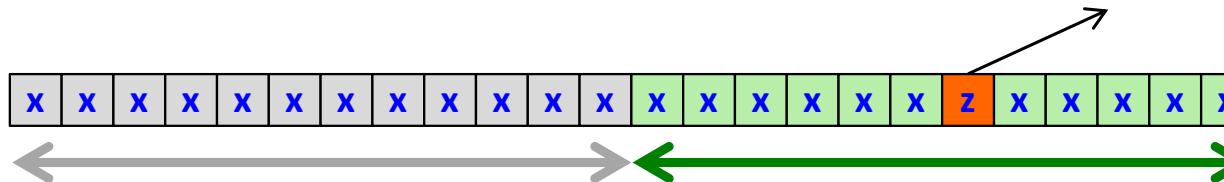


Dimensiones

```
get_work_dim() = 1  
get_global_size(0) = 24  
get_num_groups(0) = 2
```

work-item

```
get_local_id(0) = 6  
get_global_id(0) = 18
```



```
get_group_id(0) = 0  
get_local_size(0) = 12  
get_group_id(0) = 1  
get_local_size(0) = 12
```

work-group



Ejemplo OpenCL

// 1. DEFINIR LA PLATAFORMA Y LAS COLAS

```
err = clGetPlatformIDs(1, &cpPltf, NULL);
err = clGetDeviceIDs(cpPltf, devType, 1, &device_id, NULL);
context = clCreateContext(cpPltf, 1, &device_id, NULL, NULL, &err);
commands = clCreateCommandQueue(context, device_id, 0, &err);
```

// 2. CREAR Y DEFINIR EL PROGRAMA

```
program = clCreateProgramWithSource(context, 1, (const char **)KernelSource, NULL, &err);
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if (err != CL_SUCCESS) {
    size_t len; char buffer[2048];
    clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer,
    cerr << buffer << endl;
    exit(1);}
```

// 3. INICIALIZAR LOS MEMORY OBJECTS

```
tam = sizeof(float) * count;
d_a = clCreateBuffer(context, CL_MEM_READ_ONLY, tam, NULL, NULL);
d_b = clCreateBuffer(context, CL_MEM_READ_ONLY, tam, NULL, NULL);
d_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY, tam, NULL, NULL);
```

// 4. DEFINIR EL KERNEL

```
kernel = clCreateKernel(program, "mul", &err);
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);
err |= clSetKernelArg(kernel, 1, sizeof(uint), &tam);
```

// 5. ENVIAR COMANDOS, EJECUTAR KERNEL, OBTENER RESULTADO

```
err = clEnqueueWriteBuffer(commands, d_a, CL_FALSE, tam, h_a, 0, NULL, NULL);
err = clEnqueueWriteBuffer(commands, d_b, CL_FALSE, tam, h_b, 0, NULL, NULL);
err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL, &global, &local, 0, NULL, NULL);
err = clEnqueueReadBuffer(commands, d_c, CL_TRUE, 0, tam, h_c, 0, NULL, NULL );
```

¡Fácil! ¿No?

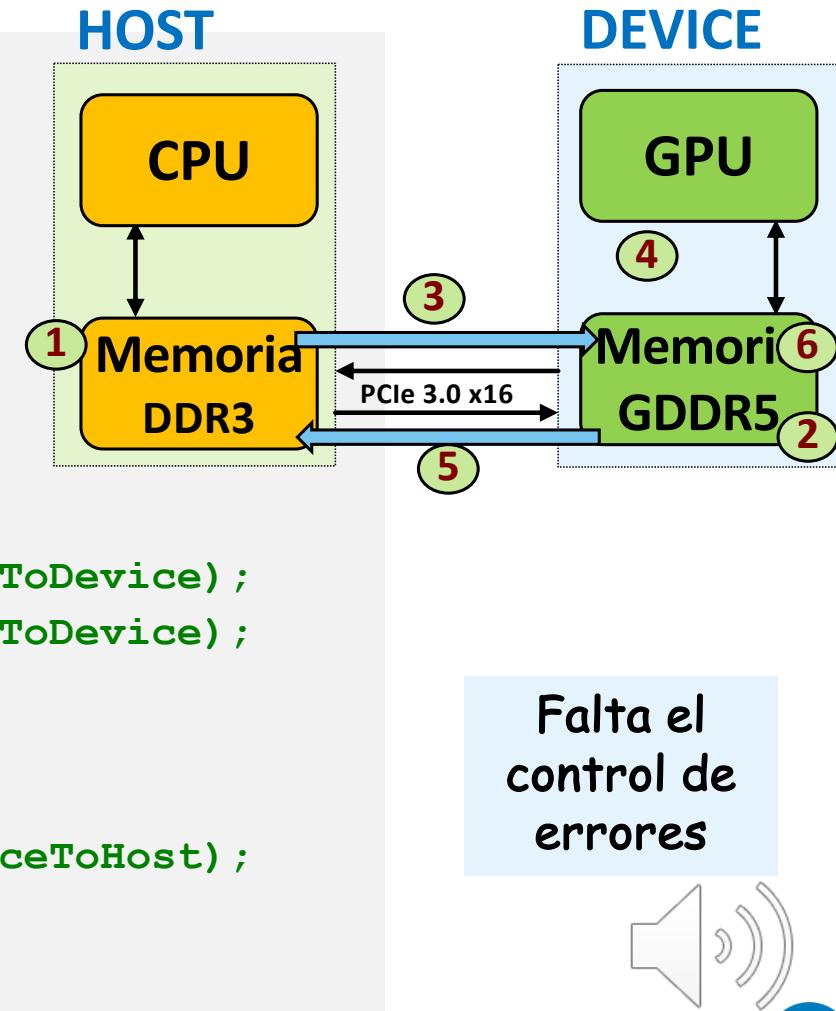
Falta el control de errores



El código equivalente en CUDA

```
1 // Obtener Memoria en el host  
2 // Obtener Memoria en el device  
3 // Copiar datos desde el host en el device  
4 // Ejecutar el kernel  
5 // Obtener el resultado desde el host  
6 // Liberar Memoria del device
```

```
unsigned int numBytes = N * sizeof(float);  
float* h_x = (float*) malloc(numBytes);  
float* h_y = (float*) malloc(numBytes);  
  
float* d_x, d_y;  
cudaMalloc((void**)&d_x, numBytes);  
cudaMalloc((void**)&d_y, numBytes);  
  
cudaMemcpy(d_x, h_x, numBytes, cudaMemcpyHostToDevice);  
cudaMemcpy(d_y, h_y, numBytes, cudaMemcpyHostToDevice);  
  
saxpyP<<<(N+63)/64, 64>>>(N, 3.5, d_x, d_y);  
  
cudaMemcpy(h_y, d_y, numBytes, cudaMemcpyDeviceToHost);  
  
cudaFree(d_x); cudaFree(d_y);
```



OpenCL

- Uno de los objetivos de OpenCL es que sea muy portable.
 - Eso hace que sea complejo y farragoso definir el código del Host.
 - Pero es fácilmente REUSABLE de una aplicación a otra.
-
- Khronos ha definido una interfaz C++ con OpenCL (cl.hpp)
 - Es mucho más fácil de trabajar con ella.
 - Todavía está en desarrollo pero se puede utilizar.



OpenCL

- Derivado de C99 (con restricciones)
- Además incluye
 - Work-items & work-groups
 - Vectores
 - Sincronización
 - Calificadores de variables
 - Una gran colección de funciones (muy parecida a la de GLSL)
- Restricciones
 - No permite punteros a funciones
 - Dispone de punteros a punteros, pero no como argumento de una función
 - No permite vectores y structs de tamaño variable
 - No permite recursividad
 - No permite campos de bits



OpenCL

Tipos de datos

□ Escalares

- char, uchar, short, ushort, int, uint, long, ulong
- float, double
- bool
- intptr_t, ptrdiff_t, size_t, uintptr_t

□ Vectores

- Tamaños: 2, 3, 4, 8 y 16
- Tipos: char, uchar, short, ushort, int, uint, long, ulong, float, double
- Ejemplos: char3, long8, float4
- Alineados a su tamaño

□ Matrices

- Todavía no están definidos.
- Está reservado: floatnxm y doublenxm



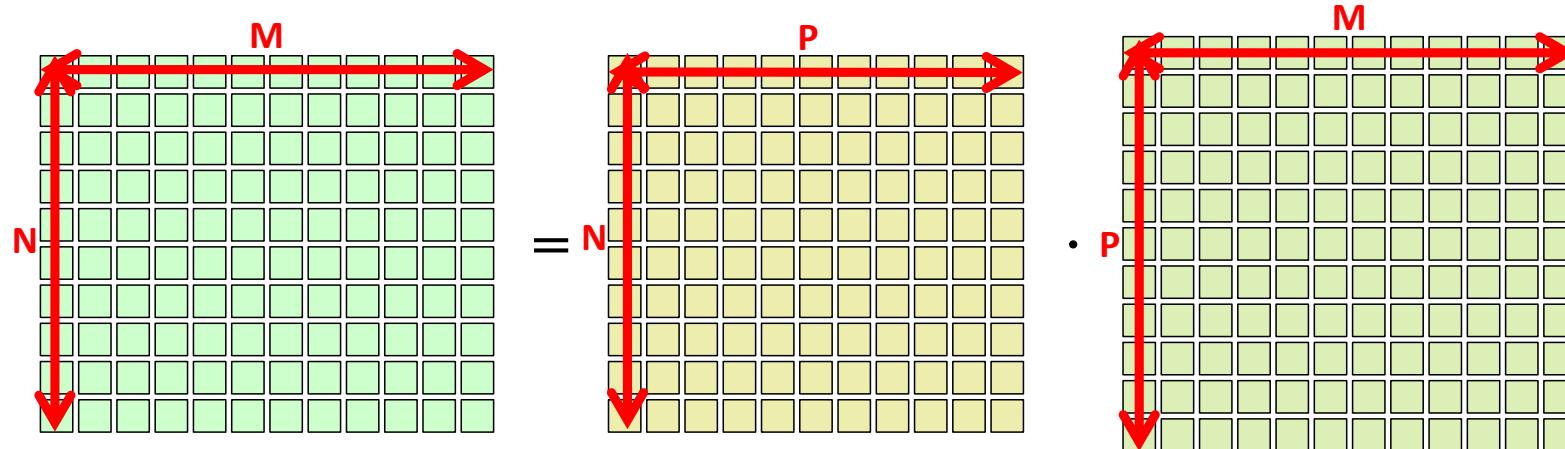
OpenCL

Sincronización

- **barrier(CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE)**
 - Todos los work-items de un work-group han de ejecutar esta función antes de poder continuar.
- **mem_fence(CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE)**
 - Espera hasta que todas las operaciones previas hechas con memoria por el work-item son visibles por todos los threads del work-group.
 - ✓ `read_mem_fence();`
 - ✓ `write_mem_fence();`
- Se pueden llamar desde un kernel.
- También se puede poner un barrier en una cola:
 - **clEnqueueBarrier()**
 - No se ejecutará nada más en la cola, hasta que todos los comandos previos en la cola hayan terminado su ejecución (non blocking)



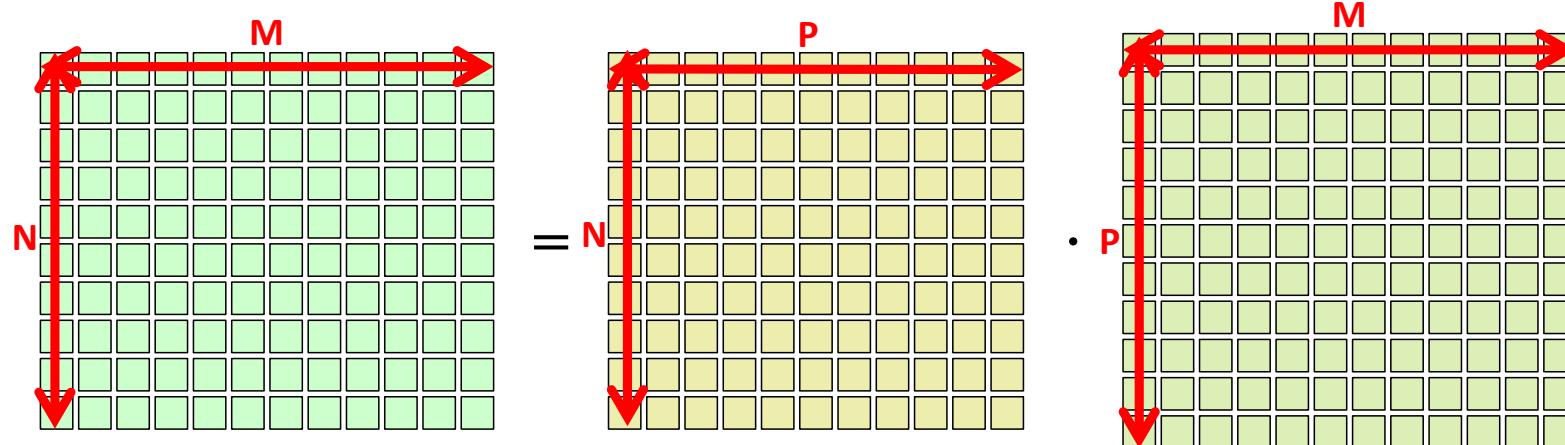
Producto de Matrices



```
void MM(float C[N][M], float A[N][P], float B[P][M]) {  
    int i, j, k;  
    for (i=0; i<N; i++)  
        for (j=0; j<M; j++) {  
            C[i][j] = 0.0;  
            for (k=0; k<P; k++)  
                C[i][j] += A[i][k] * B[k][j];  
        }  
}
```



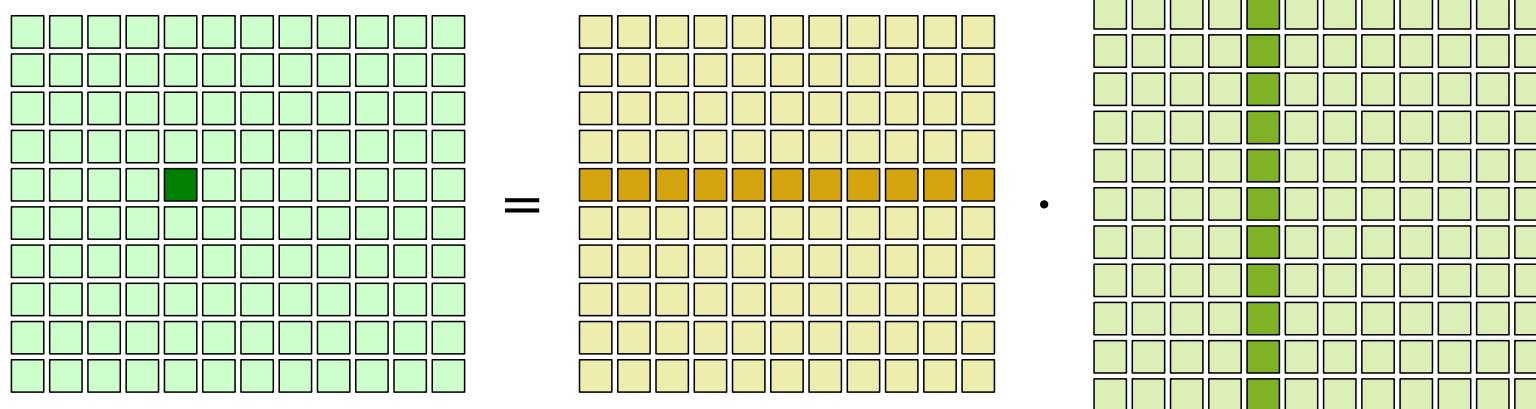
Producto de Matrices



```
void MM(int N, int M, int P, float *C, float *A, float *B){  
    int i, j, k; float tmp;  
    for (i=0; i<N; i++)  
        for (j=0; j<M; j++) {  
            tmp = 0.0;  
            for (k=0; k<P; k++)  
                tmp += A[i*N+k] * B[k*M+j];  
            C[i*M+j] = tmp;  
        }  
}
```



Producto de Matrices



```
void MM(int N, int M, int P, float *C, float *A, float *B){  
    int i, j, k; float tmp;  
    for (i=0; i<N; i++)  
        for (j=0; j<M; j++) {  
            tmp = 0.0;  
            for (k=0; k<P; k++)  
                tmp += A[i*N+k] * B[k*M+j];  
            C[i*M+j] = tmp;  
        }  
}
```



Producto de Matrices

```
void MM(int N, int M, int P, float *C, float *A, float *B){  
    int i, j, k; float tmp;  
    for (i=0; i<N; i++)  
        for (j=0; j<M; j++) {  
            tmp = 0.0;  
            for (k=0; k<P; k++)  
                tmp += A[i*P+k] * B[k*M+j];  
            C[i*M+j] = tmp;  
        }  
}  
  
kernel void K-MM(const int N, const int M, const int P,  
    global float *C, global float *A, global float *B){  
    int i, j, k; float tmp;  
    for (i=0; i<N; i++)  
        for (j=0; j<M; j++) {  
            tmp = 0.0;  
            for (k=0; k<P; k++)  
                tmp += A[i*P+k] * B[k*M+j];  
            C[i*M+j] = tmp;  
        }  
}
```



Producto de Matrices

```
kernel void K-MM(const int N, const int M, const int P,
    global float *C, global float *A, global float *B){
    int i, j, k; float tmp;
    for (i=0; i<N; i++)
        for (j=0; j<M; j++) {
            tmp = 0.0;
            for (k=0; k<P; k++)
                tmp += A[i*P+k] * B[k*M+j];
            C[i*M+j] = tmp;
        }
}
```

```
kernel void K-MM(const int N, const int M, const int P,
    global float *C, global float *A, global float *B){
    int i, j, k; float tmp;
    i = get_global_id(0);
    j = get_global_id(1);
    tmp = 0.0;
    for (k=0; k<P; k++)
        tmp += A[i*P+k] * B[k*M+j];
    C[i*M+j] = tmp;
}
```



Producto de Matrices

```
// 1. DEFINIR LA PLATAFORMA Y LAS COLAS  
.  
. .  
// 2. CREAR Y DEFINIR EL PROGRAMA  
.  
. .  
// 3. INICIALIZAR LOS MEMORY OBJECTS  
.  
. .  
// 4. DEFINIR EL KERNEL  
.  
. .  
// 5. ENVIAR COMANDOS, EJECUTAR KERNEL, OBTENER RESULTADO  
err = clEnqueueWriteBuffer(commands, d_A, CL_FALSE, tamNP, h_A, 0, NULL, NULL);  
err = clEnqueueWriteBuffer(commands, d_B, CL_FALSE, tamPM, h_B, 0, NULL, NULL);  
global[0] = 1024; global[1] = 1024;  
local[0] = 128; local[1] = 128;  
err = clEnqueueNDRangeKernel(commands, kernel, 2, NULL, global, local, 0, NULL, NULL);  
err = clEnqueueReadBuffer(commands, d_C, CL_TRUE, 0, tamNM, h_C, 0, NULL, NULL );
```

Hemos supuesto $N = M = P = 1024$



Producto de Matrices

Optimizando el Producto de Matrices (supondremos N=M=P)

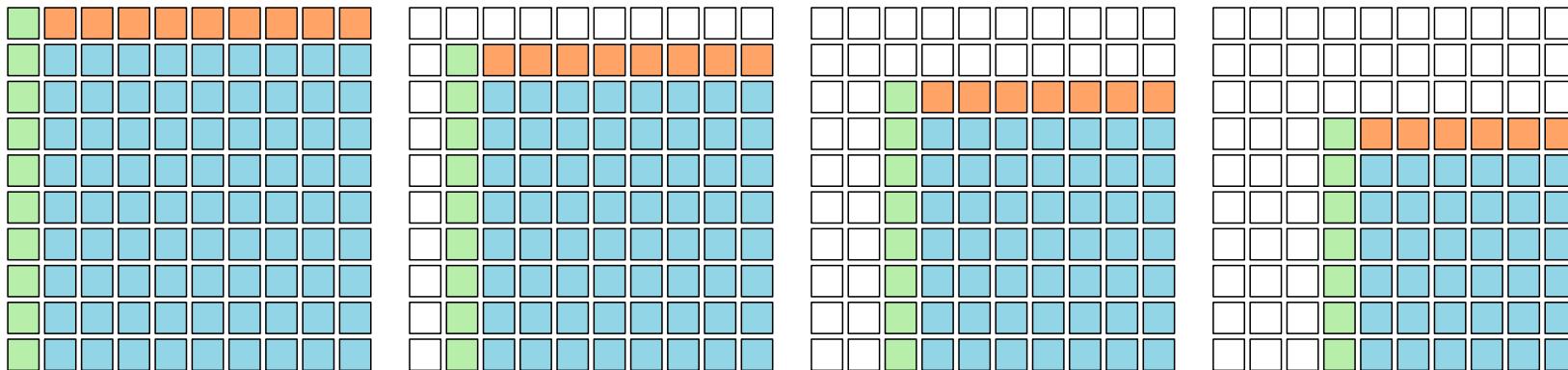
- Cálculos: $2 \cdot N^3$ operaciones en coma flotante
- Datos: $3 \cdot N^2$ números
- Accesos a Memoria: $2 \cdot N^3 + N^2$
- Repetidos accesos a los mismos datos: a cada elemento de A y B se accede N veces.
- La distribución de los cálculos es uniforme entre los work-items:
 - Todos los work-items tienen los mismos cálculos que hacer.
 - Todos los work-items se pueden ejecutar en paralelo
- Para optimizar es necesario optimizar el uso de la memoria local.
- Hay que mover datos de la memoria global a la local.
 - Cuando estén en la memoria local, hay que hacer el máximo número posible de operaciones.
 - Ideal: Llevar los datos 1 sola vez a la memoria local.



Descomposición LU

```
void LU(float A[N][N]) {  
    int i, j, k;  
    for (k=0; k<N; k++)  
        for (j=k+1; j<N; j++)  
            A[k][j] = A[k][j] / A[k][k];  
        for (i=k+1; i<N; i++) {  
            for (j=k+1; j<N; j++)  
                A[i][j] = A[i][j] - A[i][k] * A[k][j];  
        }  
}
```

La distribución de los cálculos no es uniforme: ni en el tiempo, ni en el espacio.



Descomposición LU

□ Aplicación típica de álgebra lineal: resolución sistemas de ecuaciones

$$\begin{array}{rcl} 4x - 2y + z & = & 3 \\ 20x - 7y + 12z & = & -1 \\ -8x + 13y + 17z & = & 1 \end{array}$$

$$\begin{bmatrix} 4 & -2 & 1 \\ 20 & -7 & 12 \\ -8 & 13 & 17 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 3 \\ -1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 4 & -2 & 1 \\ 20 & -7 & 12 \\ -8 & 13 & 17 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 5 & 1 & 0 \\ -2 & 3 & 1 \end{bmatrix} \cdot \begin{bmatrix} 4 & -2 & 1 \\ 0 & 3 & 7 \\ 0 & 0 & -2 \end{bmatrix}$$

Descomposición LU

$A = L \cdot U$

$A \cdot x = b$

Resolver $A \cdot x = b$

- 1) $L \cdot U \cdot x = b$
- 2) $U \cdot x = L^{-1} \cdot b$
- 3) $x = U^{-1} \cdot L^{-1} \cdot b$

Usamos Producto de matrices.
La descomposición LU también
usa el producto de matrices.



Modelo de Memoria de OpenCL

□ Memoria Privada

- *Work-item*

□ Memoria Local

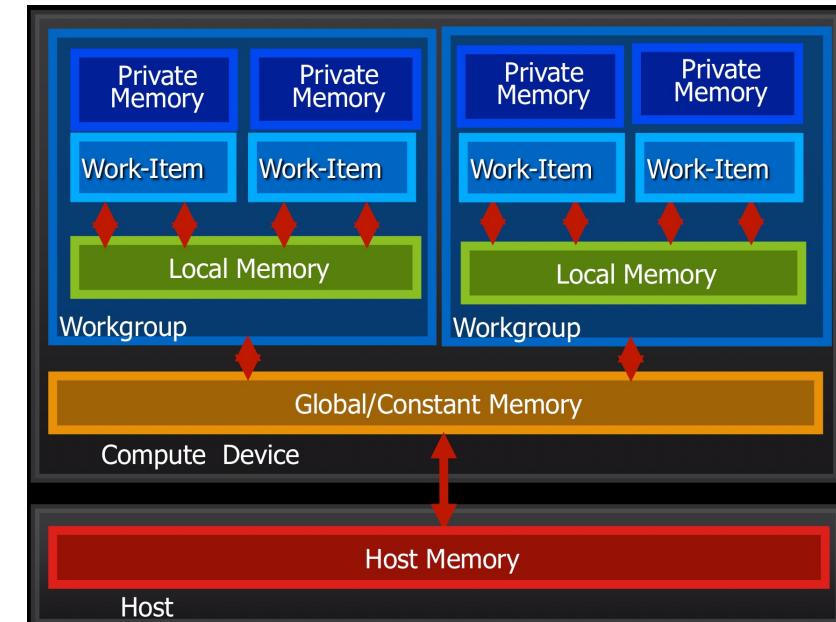
- Compartida por un *work-group*

□ Memoria Global/Constantes

- Visible por todos los *work-groups*

□ Memoria Host

- En la CPU



© Krhinos Group

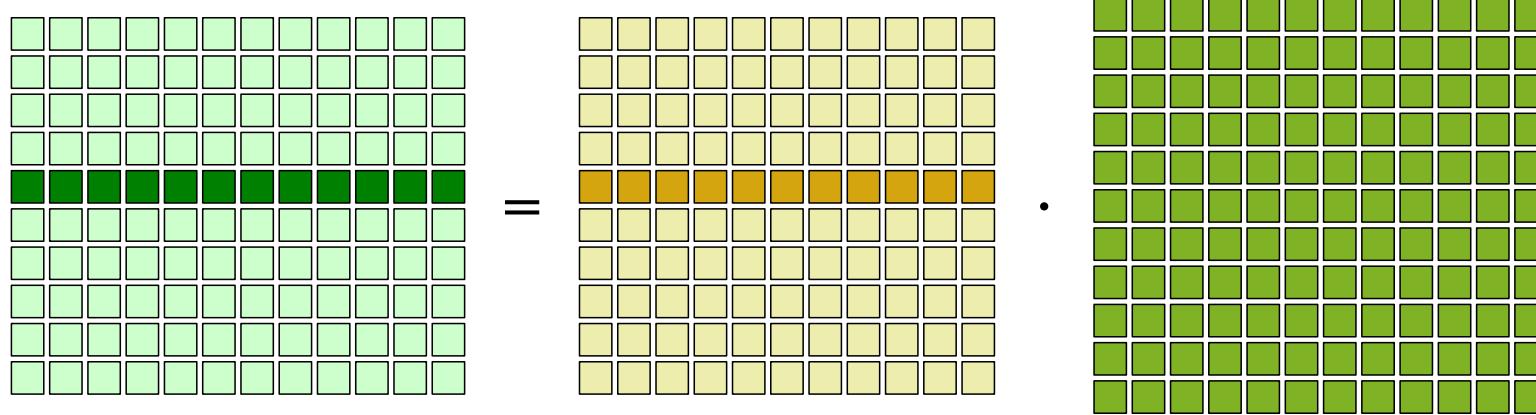
- La gestión de memoria es explícita
- El programador es el encargado de mover la información entre **host**, **global** y **local**
- Una adecuada gestión de la memoria es **fundamental** para obtener buenos rendimientos.

Modelo de Memoria de OpenCL

- Es una jerarquía de memoria, con gestión explícita

Tipo	Uso	Tamaño	Ancho Banda	Coste de 1 acceso
Memoria Privada	Work-item	10's palabras	Para un w-i 2-3-4 registros / ciclo	1 ciclo
Memoria Local	Work-group	10-100's KBytes	Para un w-g 10's words / ciclo	Pocos ciclos
Memoria Global	Kernel	GBytes	100's GB/s	100's ciclos
Memoria Host	Programa	GBytes	10's GB/s	-

Producto de Matrices



Optimización

- Un work-item calcula 1 fila de C
- Antes de empezar la ejecución del work-item se trae la fila correspondiente de A a su memoria local. [A se trae a memoria local 1 sola vez]
- Si se calcula la fila completa de C, la correspondiente fila de A no se vuelve a utilizar.
- Hay que asegurarse que la fila de A cabe en la memoria local.
- La matriz B se lee desde memoria global 1 vez para cada work-item

Producto de Matrices

```
// 1. DEFINIR LA PLATAFORMA Y LAS COLAS  
.  
. .  
// 2. CREAR Y DEFINIR EL PROGRAMA  
.  
. .  
// 3. INICIALIZAR LOS MEMORY OBJECTS  
.  
. .  
// 4. DEFINIR EL KERNEL  
.  
. .  
// 5. ENVIAR COMANDOS, EJECUTAR KERNEL, OBTENER RESULTADO  
err = clEnqueueWriteBuffer(commands, d_A, CL_FALSE, tamNP, h_A, 0, NULL, NULL);  
err = clEnqueueWriteBuffer(commands, d_A, CL_FALSE, tamPM, h_B, 0, NULL, NULL);  
global[0] = 1024;  
local[0] = 128;  
err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL, global, local, 0, NULL, NULL);  
err = clEnqueueReadBuffer(commands, d_C, CL_TRUE, 0, tamNM, h_C, 0, NULL, NULL );
```

Hemos supuesto $N = M = P = 1024$

Tendremos 8 work-groups, en cada uno de ellos se ejecutarán 128 work-items.

Producto de Matrices

```
kernel void K-MM(const int N, const int M, const int P,
    global float *C, global float *A, global float *B){
    int i, j, k; float tmp;
    i = get_global_id(0);
    j = get_global_id(1);
    tmp = 0.0;
    for (k=0; k<P; k++)
        tmp += A[i*P+k] * B[k*M+j];
    C[i*M+j] = tmp;
}
```

Kernel Original

```
kernel void K2-MM(const int N, const int M, const int P,
    global float *C, global float *A, global float *B){
    int i, j, k; float tmp;
    i = get_global_id(0);
    for (j=0; j<M; j++) {
        tmp = 0.0;
        for (k=0; k<P; k++)
            tmp += A[i*P+k] * B[k*M+j];
        C[i*M+j] = tmp;
    }
}
```

Nuevo Kernel, fila todavía A en memoria global.

Producto de Matrices

```
kernel void K2-MM(const int N, const int M, const int P,
    global float *C, global float *A, global float *B){
    int i, j, k; float tmp;
    local float pmA[maxP];

    i = get_global_id(0);
    for (k=0; k<P; k++)
        pmA[k] = A[i*P+k];

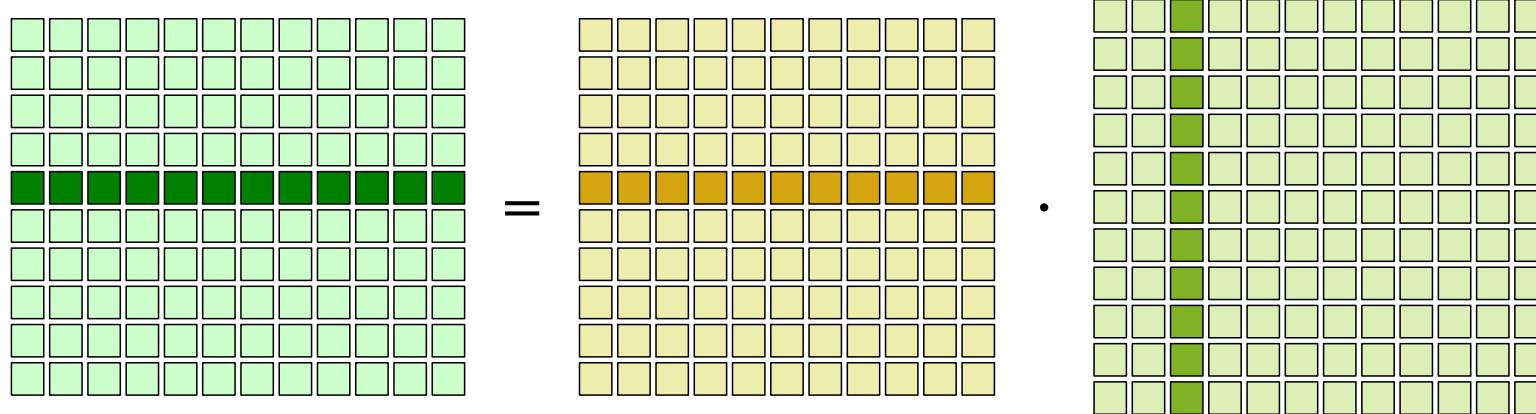
    for (j=0; j<M; j++) {
        tmp = 0.0;
        for (k=0; k<P; k++)
            tmp += pmA[k] * B[k*M+j];
        C[i*M+j] = tmp;
    }
}
```

Nuevo Kernel.

Antes de empezar los cálculos se copia la fila i de la matriz A en memoria local.

La matriz A SÓLO SE COPIA 1 VEZ en la memoria local.

Producto de Matrices



Optimización

- Un work-item calcula 1 fila de C
- Antes de empezar la ejecución del work-item se trae la fila correspondiente de A a su memoria local. [A sólo se trae a memoria local 1 vez]
- Se traen las columnas de B a la **memoria de constantes** y se comparten dentro del work-group
- La matriz B se trae a memoria de constantes **1 vez para cada work-group**.



Producto de Matrices

// 1. DEFINIR LA PLATAFORMA Y LAS COLAS

...

// 2. CREAR Y DEFINIR EL PROGRAMA

...

// 3. INICIALIZAR LOS MEMORY OBJECTS

...

// 4. DEFINIR EL KERNEL

...

// 5. ENVIAR COMANDOS, EJECUTAR KERNEL, OBTENER RESULTADO

```
err = clEnqueueWriteBuffer(commands, d_A, CL_FALSE, tamNP, h_A, 0, NULL, NULL);
err = clEnqueueWriteBuffer(commands, d_A, CL_FALSE, tamPM, h_B, 0, NULL, NULL);
global[0] = 1024;
local[0] = 128;
err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL, global, local, 0, NULL, NULL);
err = clEnqueueReadBuffer(commands, d_C, CL_TRUE, 0, tamNM, h_C, 0, NULL, NULL );
```

El acceso a la memoria de constantes se ha de asociar al kernel e inicializarla en el device.

Hemos supuesto $N = M = P = 1024$

Tendremos 8 work-groups, en cada uno de ellos se ejecutarán 128 work-items.



Producto de Matrices

```
kernel void K3-MM(const int N, const int M, const int P,
    global float *C, global float *A, global float *B,
    const float *localB){
    int i, j, k; float tmp;
    local float pmA[maxP];

    i = get_global_id(0);
    il = get_local_id(0);
    nl = get_local_size(0);
    for (k=0; k<P; k++)
        pmA[k] = A[i*P+k];

    for (j=0; j<M; j++) {
        for (k=il; k<P; k=k+nl)
            localB[k] = B[k*M+j];
        barrier(CLK_LOCAL_MEM_FENCE);
        tmp = 0.0;
        for (k=0; k<P; k++)
            tmp += pmA[k] * localB[k];
        C[i*M+j] = tmp;
    }
}
```

Nuevo Kernel.

Las filas de la matriz B se copian en la memoria de constantes. Cada work-item se encarga de una porción.

Antes de empezar el cálculo, es necesario asegurarse que todos los work-items han acabado.

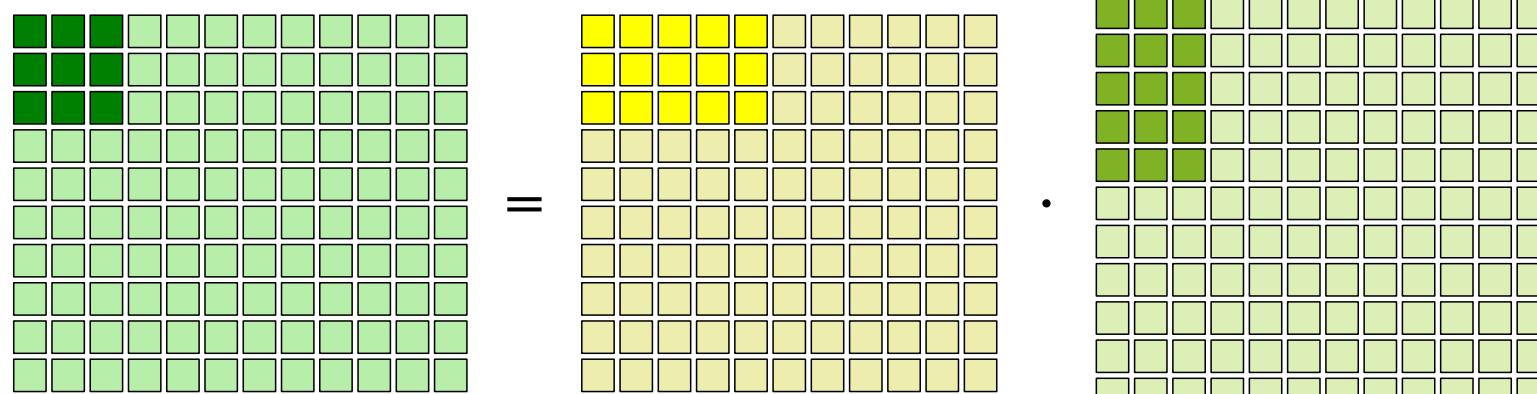
La matriz B SE COPIA 1 VEZ en la memoria de constantes por cada work-group.



Producto de Matrices

Optimizaciones finales

- Hay que hacer que el número de work-items se múltiplo del “width vector” hardware.
- Para obtener rendimientos importantes, hay que rehacer el algoritmo usando técnicas de “blocking”. A dos niveles:
 - Para aprovechar la memoria local
 - Para aprovechar los bancos de registros hardware
 - Intrucciones vectoriales



Reducción

- Esquema típico que aparece en múltiples aplicaciones
- Operaciones conmutativas: $[a \text{ op } (b \text{ op } c)] = [a \text{ op } b] \text{ op } c$

```
typeT REDUCT(int N, typeT *V) {
    int i;
    typeT tmp;
    tmp = InitValue;
    for (i=0; i<N; i++)
        tmp = tmp + V[i];
        tmp = tmp * V[i];
        tmp = max(tmp, V[i]);
        tmp = min(tmp, V[i]);
    return tmp;
}
```

```
double PI(int N) {
    int i;
    double x, s, step;
    s = 0.0;
    step = 1.0 / (double) N;
    for (i=0; i<N; i++) {
        x = (i + 0.5) * step;
        s = s + 4.0 / (1 + x * x);
    }
    return s * step;
}
```

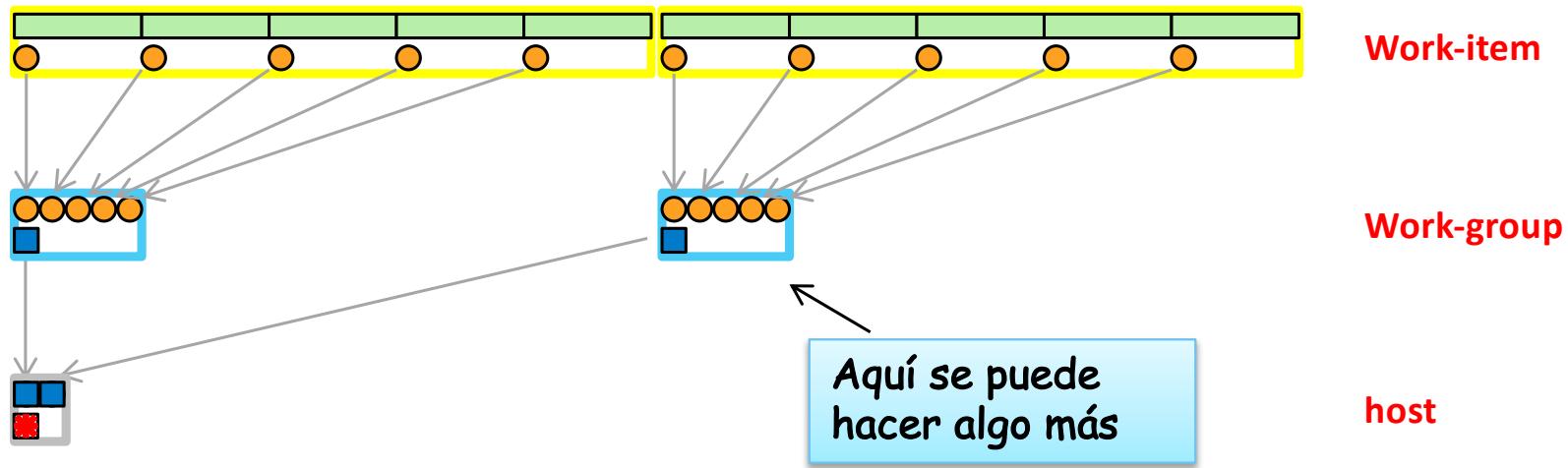
- La implementación en una GPU requiere sincronización



Reducción

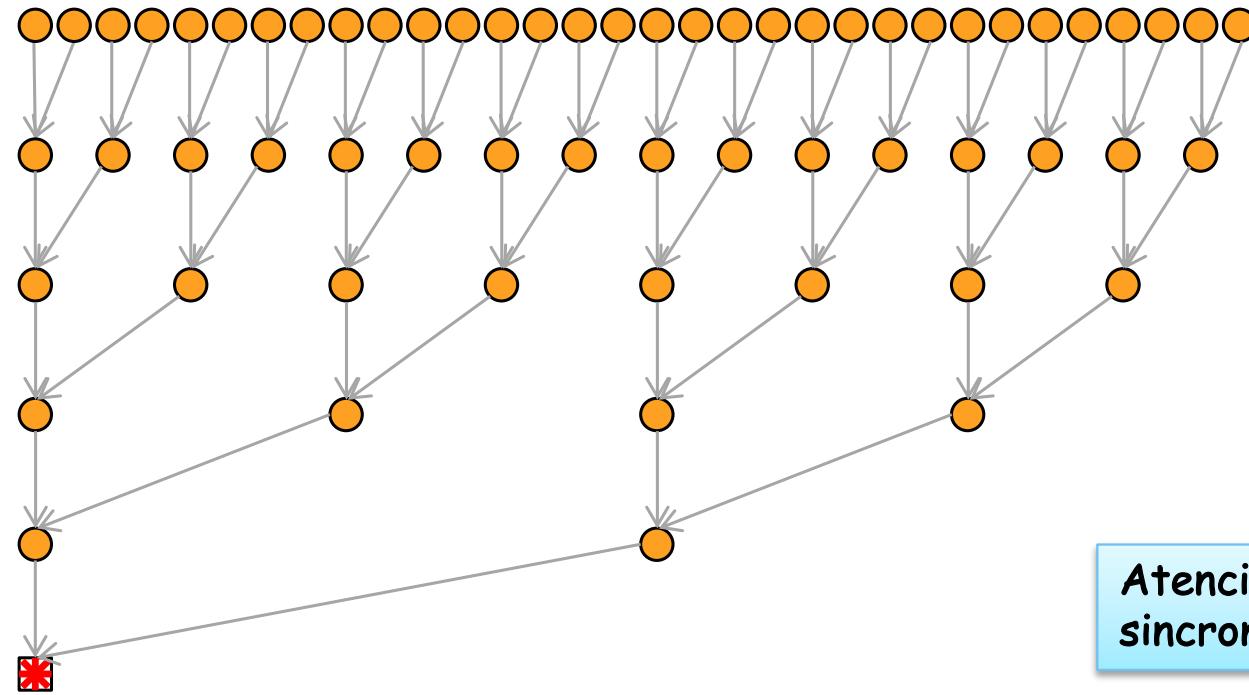
□ Se puede hacer una implementación relativamente simple:

1. Cada work item hace la reducción con la porción del vector que le corresponde y deja el resultado en una zona de memoria común al work group.
2. Cuando todos los work items han acabado, uno de ellos hace la reducción de todos los resultados parciales del work-group y deja el resultado en memoria global.
3. Cuando todos los work-groups han acabado, el host hace la reducción final.



Reducción

- Reducción paralela con 1 dato por work-item
- Sólo aplicable con un número de work-items potencia de 2



Atención a la
sincronización



Reducción

□ Reducción paralela con 1 dato por work-item [cálculo SUM]

```
kernel void reduce(global float* buffer, local float* scratch,
                  const int length, global float* result) {
    int global_index = get_global_id(0);
    int local_index = get_local_id(0);
    if (global_index < length) // Load data into local memory
        scratch[local_index] = buffer[global_index];
    else scratch[local_index] = 0.0; // Identity
    barrier(CLK_LOCAL_MEM_FENCE);

    for(int offset = 1; offset < get_local_size(0); offset <= 1) {
        int mask = (offset << 1) - 1;
        if ((local_index & mask) == 0)
            scratch[local_index] += scratch[local_index + offset];
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    if (local_index == 0) result[get_group_id(0)] = scratch[0];
}
```



Reducción

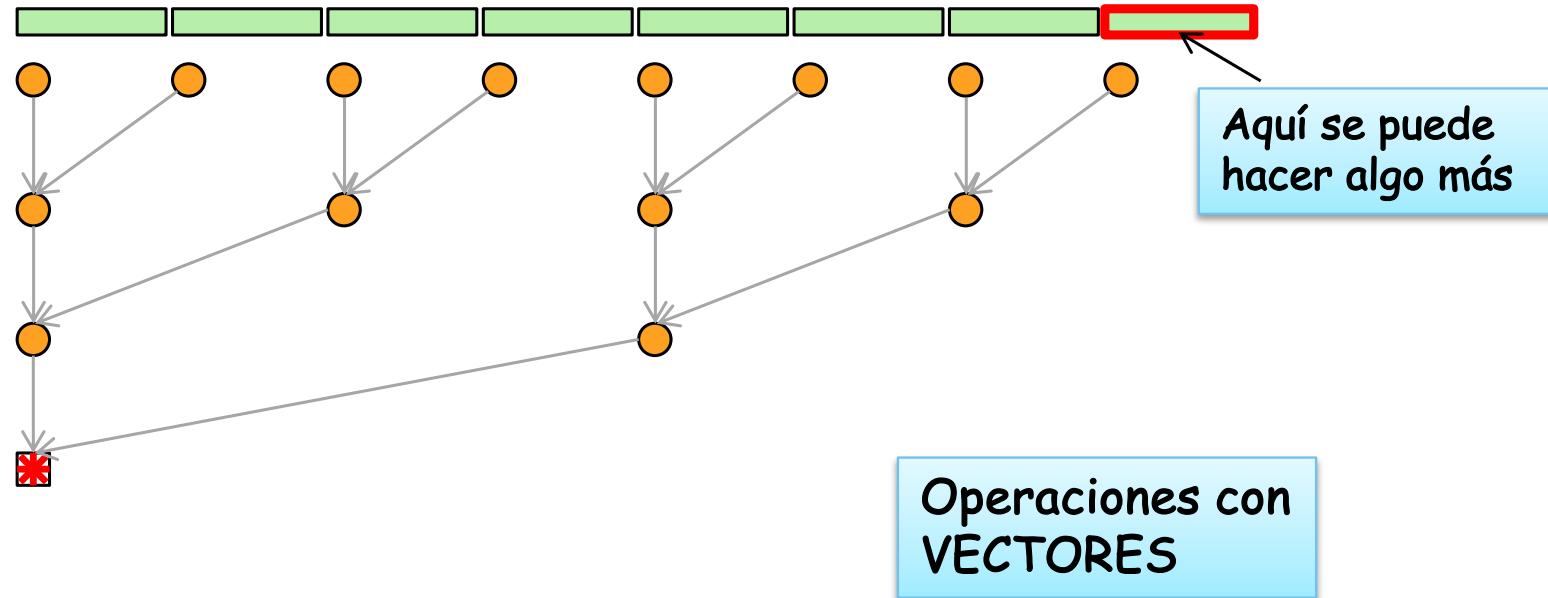
□ Reducción paralela con 1 dato por work-item [cálculo MIN]

```
kernel void reduce(global float* buffer, local float* scratch,
                  const int length, global float* result) {
    int global_index = get_global_id(0);
    int local_index = get_local_id(0);
    if (global_index < length) // Load data into local memory
        scratch[local_index] = buffer[global_index];
    else scratch[local_index] = INFINITY; // Identity
    barrier(CLK_LOCAL_MEM_FENCE);
    for(int offset = 1; offset < get_local_size(0); offset <= 1) {
        int mask = (offset << 1) - 1;
        if ((local_index & mask) == 0) {
            float other = scratch[local_index + offset];
            float mine = scratch[local_index];
            scratch[local_index] = (mine < other) ? mine : other;
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    if (local_index == 0) result[get_group_id(0)] = scratch[0];
}
```



Reducción

□ ¿Dónde se puede mejorar?



Operaciones con Vectores

- La mayoría de GPUs disponen de instrucciones SIMD.
- La mayoría de CPUs disponen de instrucciones SIMD:
 - SSE, MMX, AVX, ...
- Para obtener buenos rendimientos es muy importante aprovechar estos recursos
 - El compilador debería ser capaz de aprovecharlos.
 - La experiencia demuestra que no es así.
- Hay que programarlo explícitamente.
 - Código NO PORTABLE
 - Programación en ensamblador
 - Pocos programadores utilizan estas instrucciones
- Necesitaríamos un lenguaje de alto nivel, portable con un conjunto de instrucciones vectoriales:

OpenCL



Operaciones con Vectores

- OpenCL dispone de vectores de 2, 3, 4, 8 y 16 elementos de diferentes tipos:

- Enteros y naturales de 8, 16, 32 y 64 bits
 - Coma flotante de 32 y 64 bits.

- Dispone de operadores y funciones para manipularlos:

```
int8 v1 = (int8) (0, 1, 2, 3, 4, 5, 6, 7);  
int8 v2 = (int8) -3;  
v2 = v2+v1;  
v2 = 3-v1;  
v1 = abs(v2);
```

- La única restricción es que han de estar alineados en memoria al tamaño del vector.
- Programar en OpenCL con los vectores predefinidos por el lenguaje garantiza que el código sea portable entre diferentes CPUs y/o GPUs.



Operaciones con Vectores

- Programar con vectores es relativamente simple

```
float SUM(int N, float *v) {  
    int i;  
    float tmp;  
  
    tmp = 0.0;  
    for (i=0; i<N; i++)  
        tmp = tmp + v[i];  
    return tmp;  
}
```

```
float SUM(int N, float *v) {  
    int i;  
    float4 tmp = (float4) 0.0;  
  
    for (i=0; i<N; i=i+4){  
        tmp.x = tmp.x + v[i];  
        tmp.y = tmp.y + v[i+1];  
        tmp.z = tmp.z + v[i+2];  
        tmp.w = tmp.w + v[i+3];  
    }  
    for (; i<N; i++)  
        tmp.x = tmp.x + v[i];  
    return tmp.x + tmp.y + tmp.z + tmp.w;  
}
```



Operaciones con Vectores

- Programar con vectores es relativamente simple

```
float SUM(int N, float *V) {  
    int i;  
    float4 tmp = (float4) 0.0;  
  
    for (i=0; i<N; i=i+4){  
        tmp.x = tmp.x + V[i];  
        tmp.y = tmp.y + V[i+1];  
        tmp.z = tmp.z + V[i+2];  
        tmp.w = tmp.w + V[i+3];  
    }  
    for (; i<N; i++)  
        tmp.x = tmp.x + V[i];  
    return tmp.x + tmp.y +  
        tmp.z + tmp.w;  
}
```

```
float SUM(int N, float *V) {  
    int i;  
    float4 tmp = (float4) 0.0;  
    float4 vl;  
  
    for (i=0; i<N; i=i+4){  
        vl = vload4(0, &V[i]);  
        tmp = tmp + vl;  
    }  
    for (; i<N; i++)  
        tmp.x = tmp.x + V[i];  
    return tmp.x + tmp.y +  
        tmp.z + tmp.w;  
}
```

El Kernel puede correr tanto en una CPU como en una GPU



Operaciones con Vectores

- Programar con vectores es relativamente simple

```
double PI(int N) {  
    int i;  
    double x, s, step;  
    s = 0.0;  
    step = 1.0 / (double) N;  
    for (i=0; i<N; i++) {  
        x = (i + 0.5) * step;  
        s = s + 4.0 / (1 + x * x);  
    }  
    return s * step;  
}
```

```
double PI(int N) {  
    int i;  
    double4 x;  
    double4 tmp = (double4)(0, 1, 2, 3);  
    double4 s = (double4) 0.0;  
    double step = 1.0 / (double) N;  
  
    for (i=0; i<N; i=i+4) {  
        x = (tmp + 0.5) * step  
        s = s + 4.0 / (1 + x * x);  
        tmp = tmp + 4.0;  
    }  
    return (s.x + s.y + s.z + s.w) * step;  
}
```

El Kernel puede correr tanto en una CPU como en una GPU



Consideraciones finales

□ Cálculo heterogéneo

- Es factible correr varios kernels en varios devices simultáneamente
- Podemos utilizar varias GPUs y la CPU simultáneamente para cálculo

□ Hay que definir un context con múltiples platforms, devices y múltiples colas.

□ Podemos utilizar sincronización entre colas con eventos.

□ Nada impide definir varios contextos.

□ Rutinas a estudiar en detalle:

- `cl_int = clGetPlatformIDs(-, &cpPlatf, -);`
- `cl_int = clGetDeviceIDs(cpPlatf, -, -, &device_id, -);`
- `context = clCreateContext(cpPlatf, -, -, -, -, -);`
- `commands = clCreateCommandQueue(context, -, -, -);`



Consideraciones finales

Ideas generales a tener en cuenta

□ Acceso eficiente a memoria

- Memory coalescing: lo ideal es que el work-item i acceda a $v[i]$ y el work-item j acceda a $v[j]$
- Alineamiento de memoria. Es muy importante que todo esté alineado a 16, 32 o 64 bytes. Hay que usar padding.
- Ejemplo de cómo hay que pensar:
 - ✓ Vector de estructuras: $(x, y, z, w), (x, y, z, w), (x, y, z, w), (x, y, z, w), (x, y, z, w), \dots, (x, y, z, w)$
 - Bueno para aprovechar la jerarquía de memoria en una CPU
 - ✓ Estructura de vectores: $(x, x, x, x, \dots, x), (y, y, y, y, \dots, y), (z, z, z, z, \dots, z), (w, w, w, w, \dots, w)$
 - Bueno en una GPU (memory coalescing)

□ Número de work items y tamaño de los work groups.

- Trabajando con GPUs, idealmente necesitamos al menos 4 work-items por Processing Element en una Computing Unit.
- Aumentar este número es bueno, pero hay un límite que no conviene superar, porque los recursos de cada PE son finitos



Consideraciones finales

Ideas generales a tener en cuenta

□ Cómo sabemos que estamos haciendo bien las cosas

- Uso de profilers (ocupar los recursos >50% es una buena medida)
- Uso del ancho de banda. Es fácil calcular el ancho de banda que estamos utilizando. Es fácil saber el ancho de banda de pico de nuestra máquina.
- Registros por work-item. Es fácil averiguar cuantos registros hay disponibles en una GPU.
 - ✓ Si tenemos 32K registros y definimos 8 work-groups con 1024 work-items, nos da 4 registros por work-item. Normalmente un work-item necesita alrededor de 20 registros.

□ Portabilidad

- Si optimizamos nuestro código al máximo (número de work items, tamaños de memoria, ancho vector, ...) la portabilidad será un problema.
- La jerarquía de memoria puede variar mucho de un dispositivo a otro.
- La coma flotante en doble precisión puede ser un problema.
- Determinadas cosas se pueden calcular en tiempo de ejecución:
 - ✓ Número de work-items/ work-groups
 - ✓ Ancho de vector



OpenCL vs CUDA

Hardware

- CUDA sólo funciona en las GPUs de **Nvidia**
 - Existe un compilador de CUDA a x86 (**PGI**)
- OpenCL está disponible en:
 - GPUs de **Nvidia**
 - GPUs y CPUs de **AMD**
 - **Apple** (MacOS X) en GPUs y CPUs
 - CPUs, GPUs y MIC (Xeon Phi) de **Intel**
 - FPGAs de **Altera**
 - GPUs de **ARM**



OpenCL vs CUDA

Portabilidad del código

- Una aplicación CUDA debería funcionar, con buenos rendimientos, en cualquier GPU compatible de Nvidia. Sólo necesitaría recompilar.
- Aunque OpenCL puede trabajar con diferentes dispositivos. Nada nos asegura que un código OpenCL funcione en cualquier dispositivo sin un esfuerzo considerable.
El esfuerzo de reescritura se centra en el código de kernel.
El código de host no debería cambiar.

Rendimiento

- Con el mismo hardware los rendimientos deberían ser similares.
- Los resultados experimentales le dan ventaja a CUDA.
- Sólo se pueden hacer comparaciones con las GPUs de Nvidia.
- Las GPUs de Nvidia están diseñadas para la ejecución óptima de CUDA.



OpenCL vs CUDA

Capacidades

- OpenCL no dispone de pinned memory. Penaliza el ancho de banda en un factor de 2.
- OpenCL dispone de herramientas de sincronización más flexibles que CUDA.
- CUDA dispone de herramientas mucho más potentes que OpenCL: debugger, profiling, librerías.
- La API de CUDA es mucho más simple de utilizar (p.e.: para trabajar con varias GPUs)
- CUDA está más orientado a C que a C++.
- OpenCL permite trabajar en C++ de forma más natural que CUDA.
- OpenCL permite trabajar con vectores de forma explícita



Referencias

- Página de Khronos Group:
 - www.khronos.org/opencl/resources
 - El manual de OpenCL 2.2:
www.khronos.org/registry/cl/specs/opencl-2.2.pdf
 - El manual de OpenCL 2.2:
www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf
- Help de Xcode en los Mac de Apple
- Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry & Dana Schaa.
Heterogeneous Computing with OpenCL.
Morgan Kaufmann, 2011





UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Departament d'Arquitectura de Computadors

Tarjetas Gráficas y Aceleradores

OpenCL

Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya

