

Introducción a la computación en Paralelo -OpenCL-

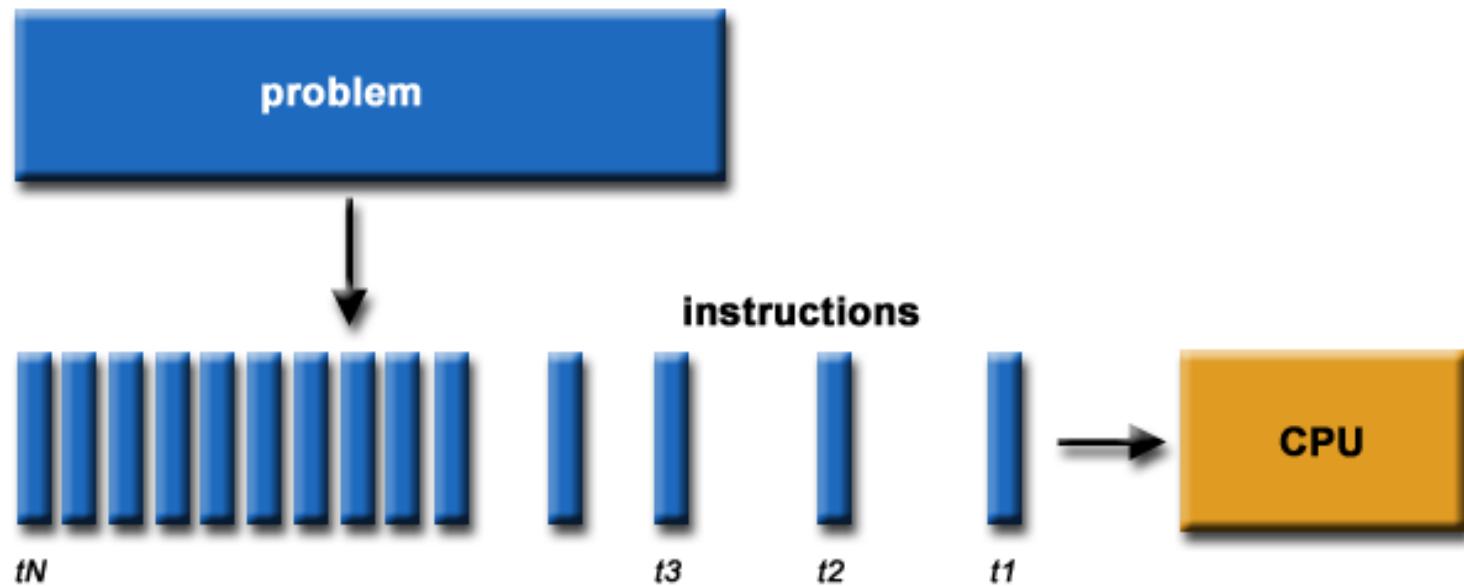
Arq. De Comp. – FAMAF
J. Fraire

Introducción

» Hacia la computación en paralelo

Herencia

- ▶ Tradicionalmente, software es escrito para ejecución serial
 - Solo una instrucción por ciclo



Lo que nos rodea

► Pero el universo, es paralelo...



Galaxy Formation



Planetary Movements



Climate Change



Rush Hour Traffic



Plate Tectonics



Weather



Drive-thru Lunch



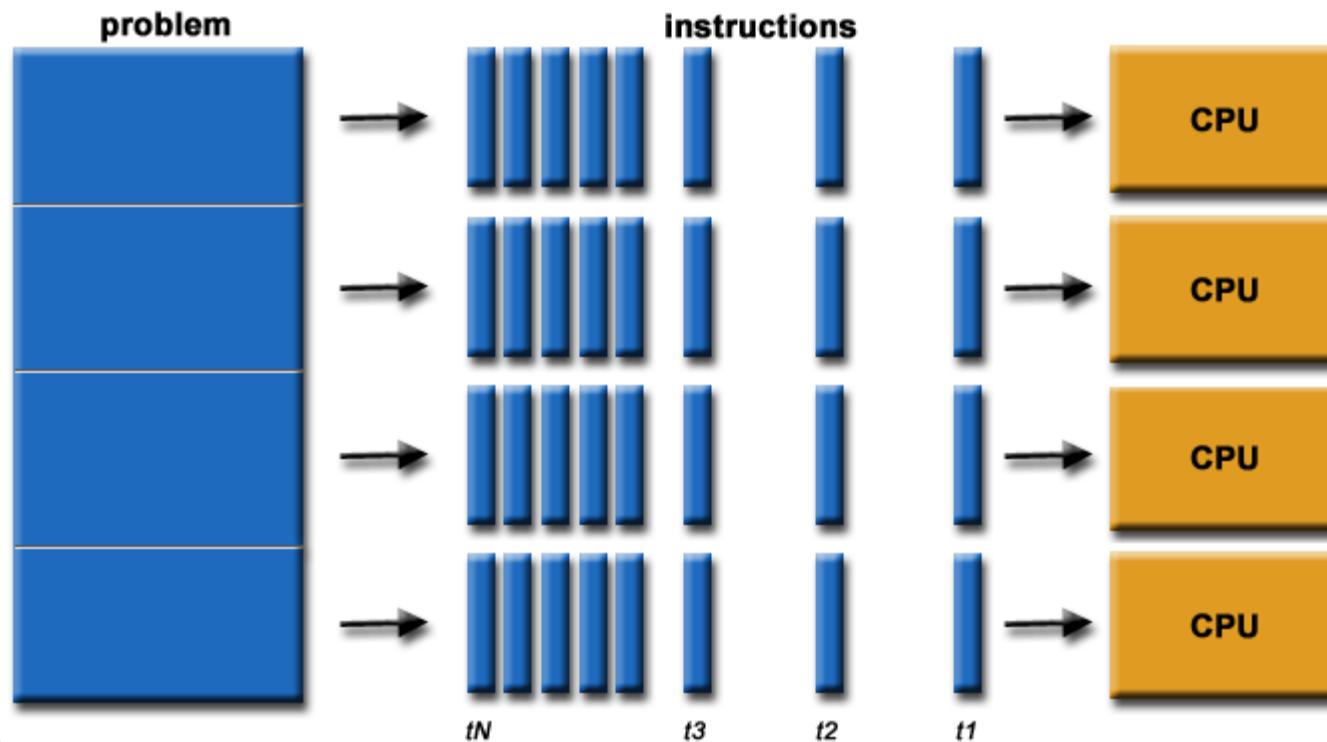
Auto Assembly



Jet Construction

Hacia lo paralelo

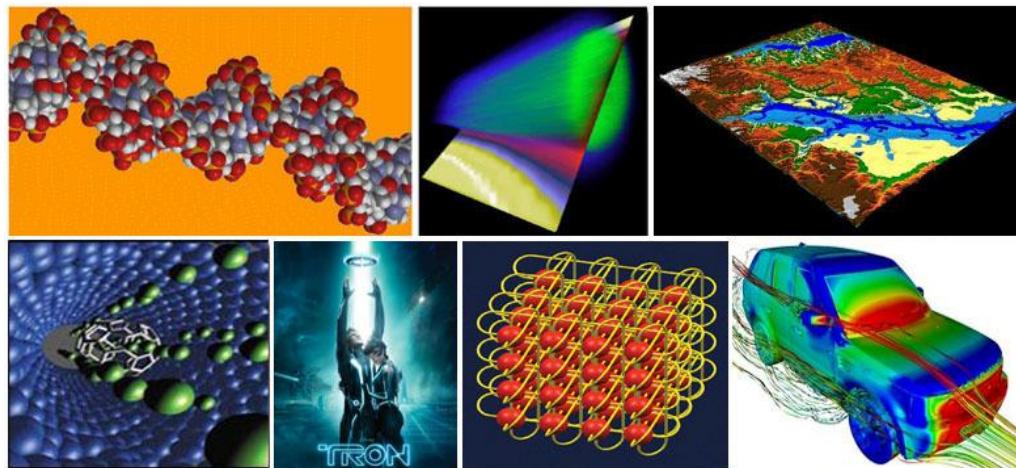
- ▶ La computación en paralelo propone
 - Múltiple CPUs / Múltiple instrucciones concurrentes



Características

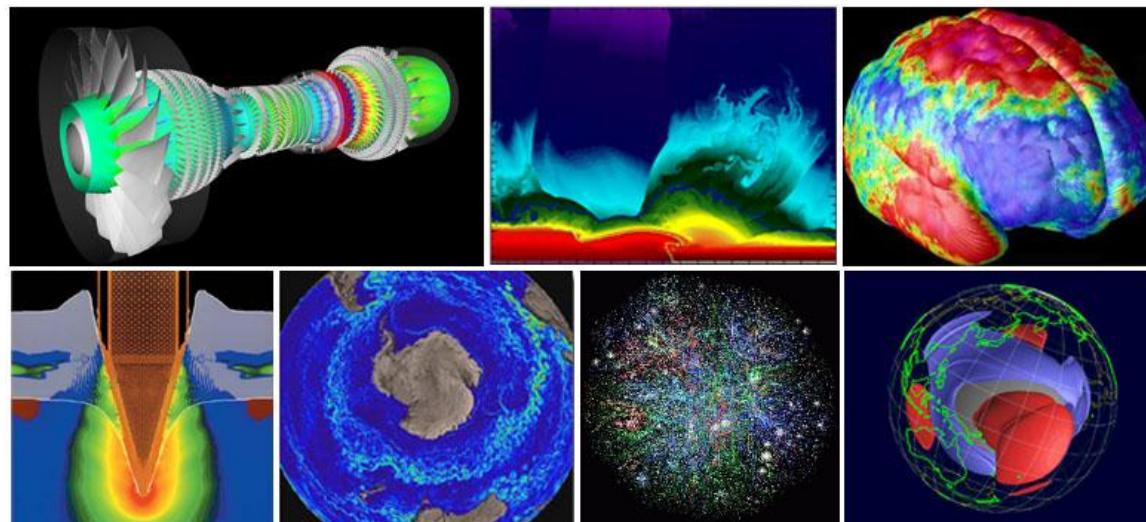
- ▶ Recursos:
 - Computadora con múltiples procesadores
 - Múltiples computadoras en red
 - Combinación de ambos
- ▶ El Problema a resolver:
 - Poder dividirse en porciones concurrentes
 - Ejecutar múltiple instrucciones simultaneas
 - Resolverse mas rápido en paralelo que en serie

Usos



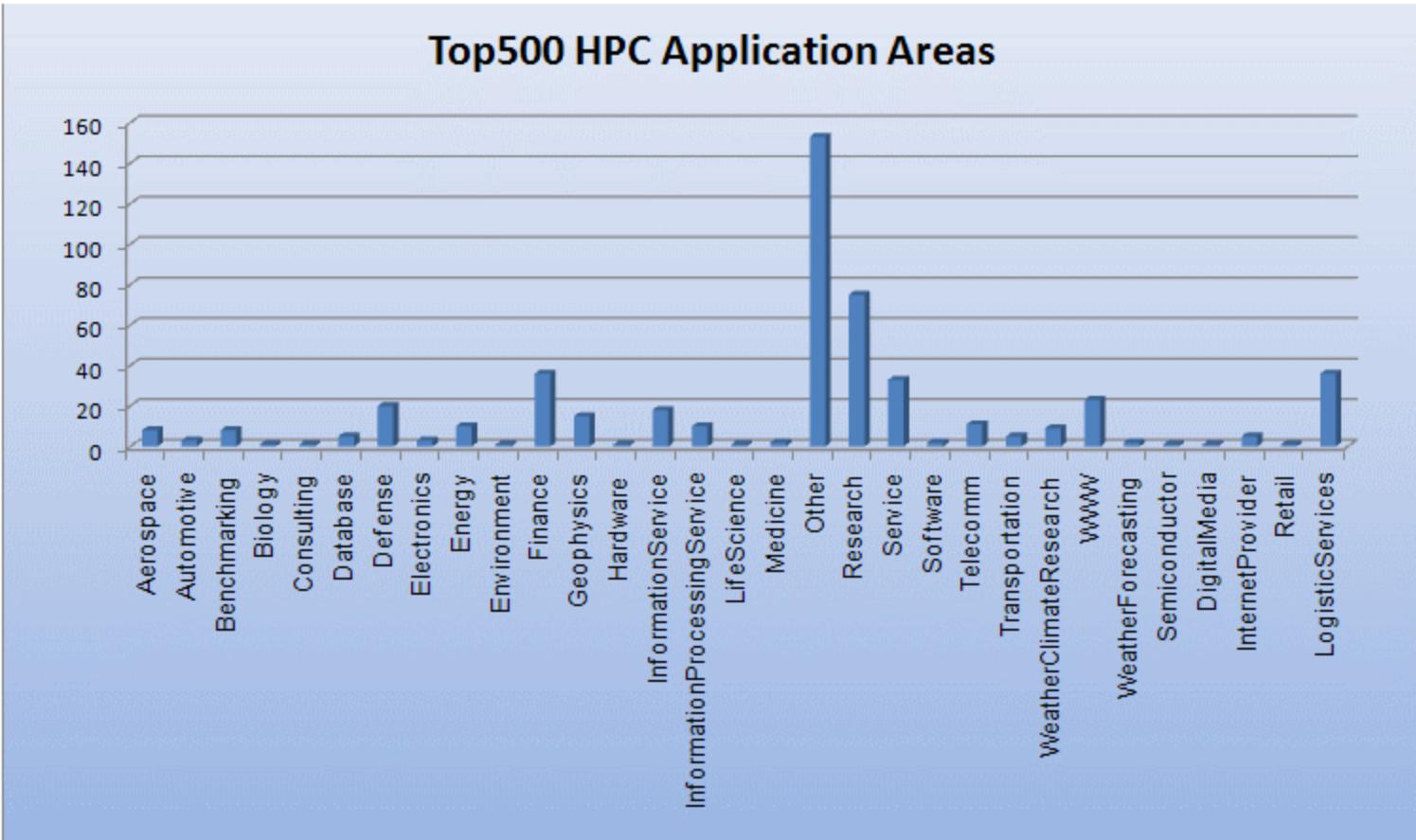
- ▶ Atmosphere, Earth, Environment
- ▶ Physics – applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
- ▶ Bioscience, Biotechnology, Genetics
- ▶ Chemistry, Molecular Sciences
- ▶ Geology, Seismology
- ▶ Mechanical Engineering – from prosthetics to spacecraft
- ▶ Electrical Engineering, Circuit Design, Microelectronics
- ▶ Computer Science, Mathematics

- ▶ Databases, data mining
- ▶ Oil exploration
- ▶ Web search engines, web based business services
- ▶ Medical imaging and diagnosis
- ▶ Financial and economic modeling
- ▶ Management of national and multi-national corporations
- ▶ Advanced graphics and virtual reality, particularly in the entertainment industry
- ▶ Networked video and multi-media technologies



Usos

▶ Estadísticas de Top500



Para que?

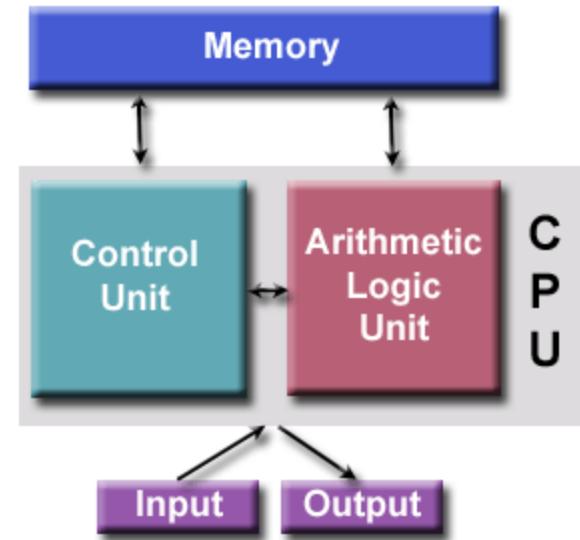
- ▶ Ahorrar dinero
- ▶ Resolver problemas mas complejos
- ▶ Proveer concurrencia
- ▶ Uso de recursos remotos
- ▶ Límites de computación serial
 - Velocidad en cobre: 9cm/nanosegundos
 - Cada vez es mas caro/complexo reducir superficie de silicio por procesador

Conceptos

» Flynn, Nodes, CPUs, Cores

Arq. Von-Neumann

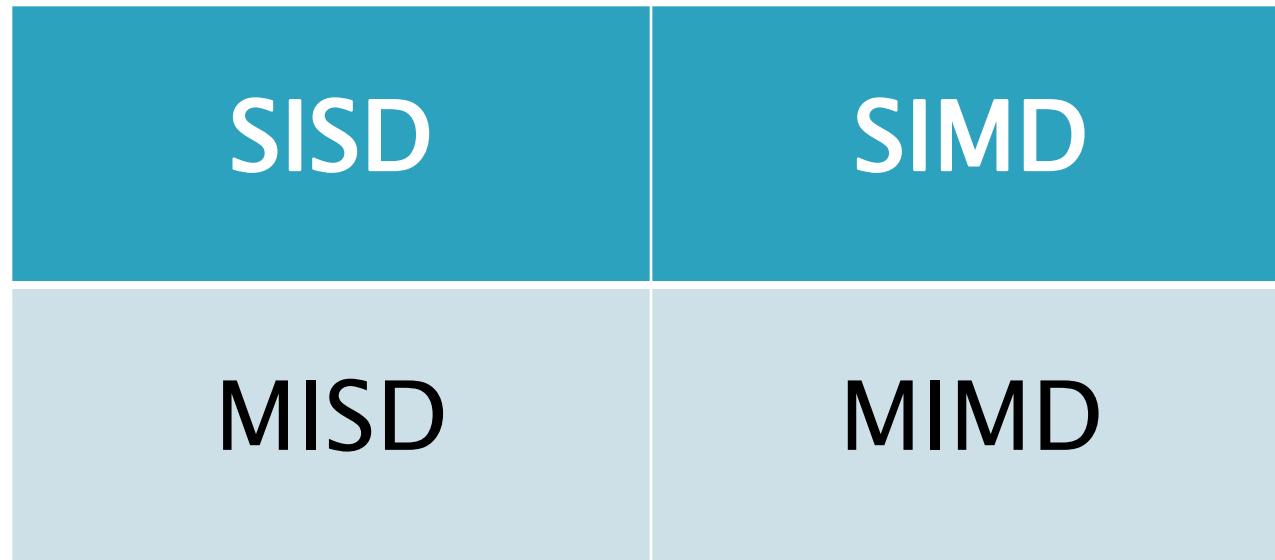
- Random Access Memory
 - Datos e Instrucciones
- Control Unit
 - Extrae Datos e Instrucciones
 - Coordina secuencialmente ALU
- Arithmetic Logic Unit
 - Procesos aritméticos



“Computación en paralelo sigue el mismo diseño multiplicado en unidades”

Taxonomía de Flynn

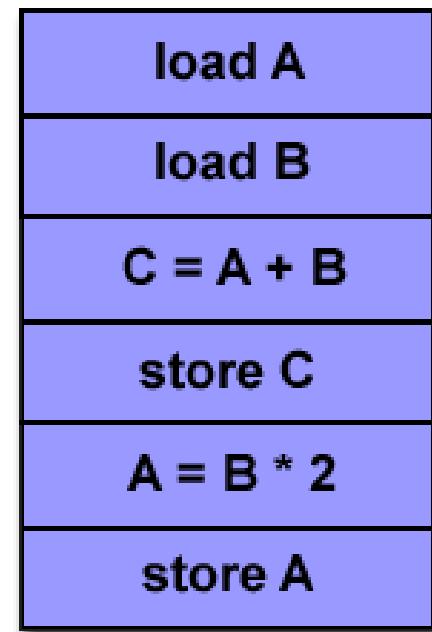
- ▶ 1966: Flynn definió dos dimensiones:
 - Instrucciones
 - Datos



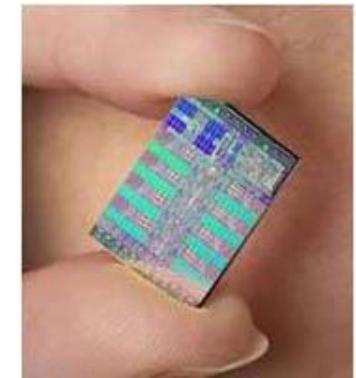
SISD

▶ Computación clásica serial:

- Single Instruction
- Single Data
- Ejecución
- determinísitca

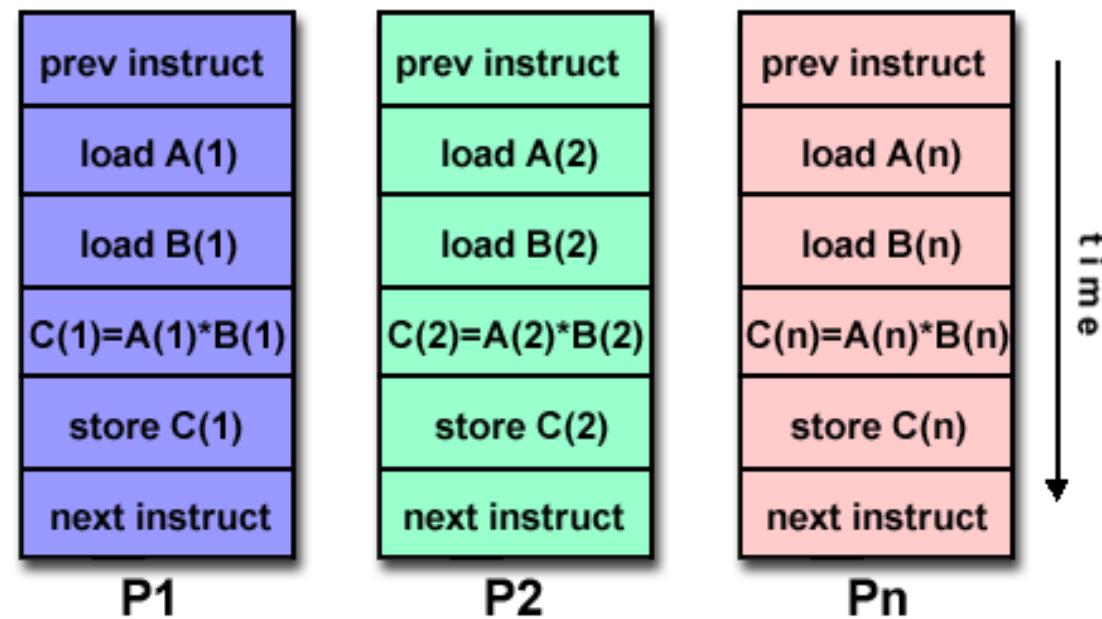


SIMD



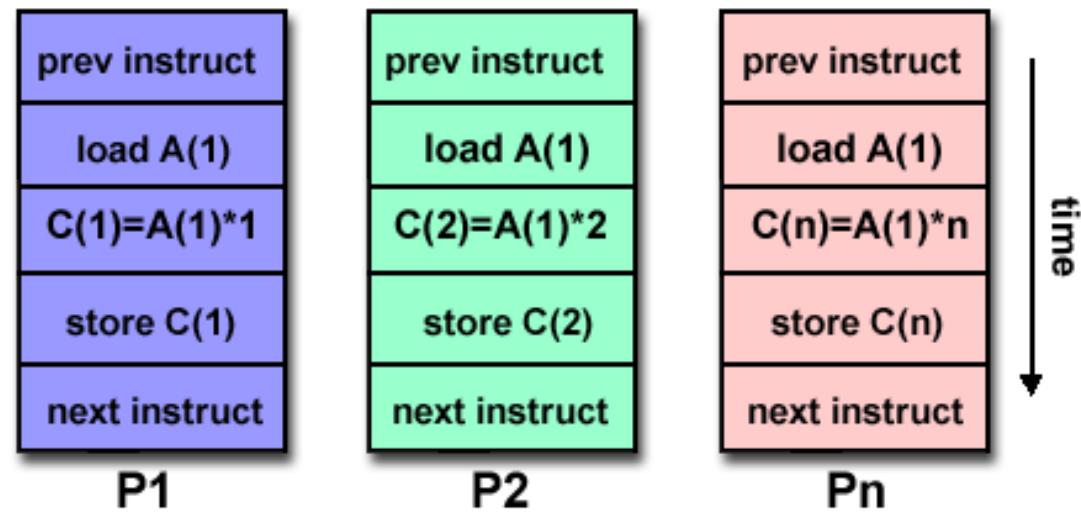
▶ Single Instruction Multiple Data

- Cada unidad de procesamiento ejecuta la misma instrucción
- Pero pueden operar en diferentes datos
- Utilizado para problemas de alta regularidad (procesamiento de imágenes y video)



MISD

- ▶ Multiple Instruction Single Data
 - No muchas implementaciones reales
 - Diferentes unidades de procesamiento actua sobre una misma porción de memoria
 - Utilizado para múltiples filtros de frecuencias
 - Crack criptográfico

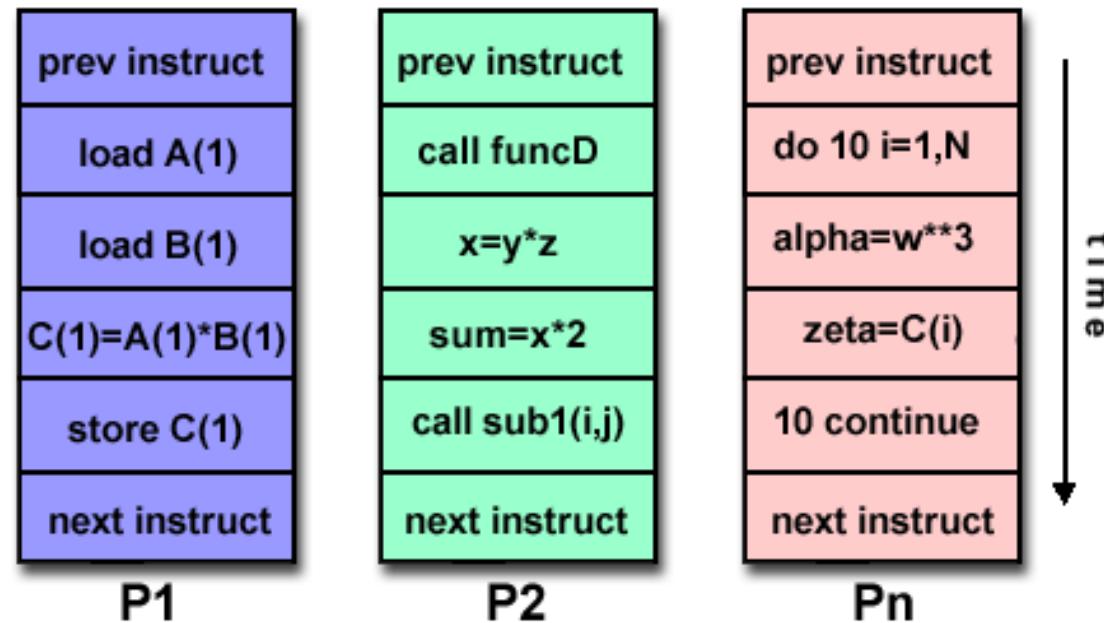


MIMD



▶ Multiple Instruction Multiple Data

- Cada Procesador puede ejecutar diferentes instrucciones
- Cada Procesador puede acceder diferentes datos
- Puede ser:
 - Síncrono
 - Asíncrono
- SuperComputers
- Clusters
- Grids
- MultiCores



Nomenclatura

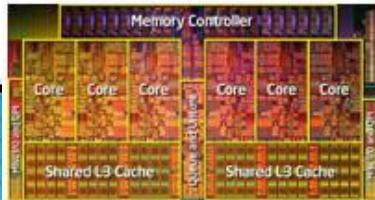
- ▶ HPC
- ▶ Node -> Computer in a Box (múltiples CPUs)
- ▶ CPU -> múltiples Cores
- ▶ Cores -> single execution unit



Supercomputer - each blue light is a node

Node - standalone
Von Neumann computer

CPU / Processor / Socket - each has multiple cores / processors.

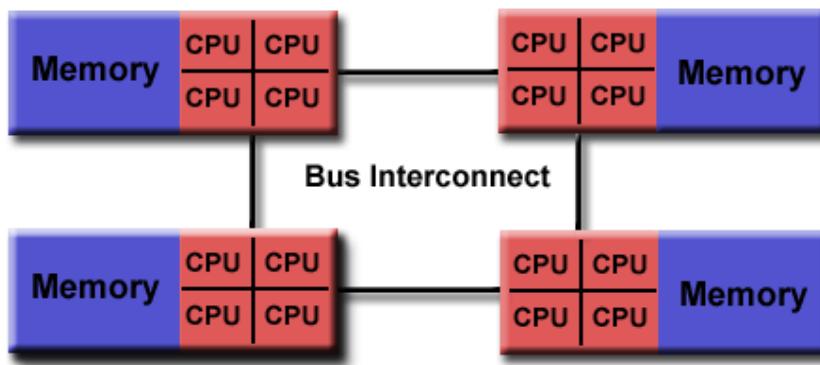


Arquitectura de Memoria

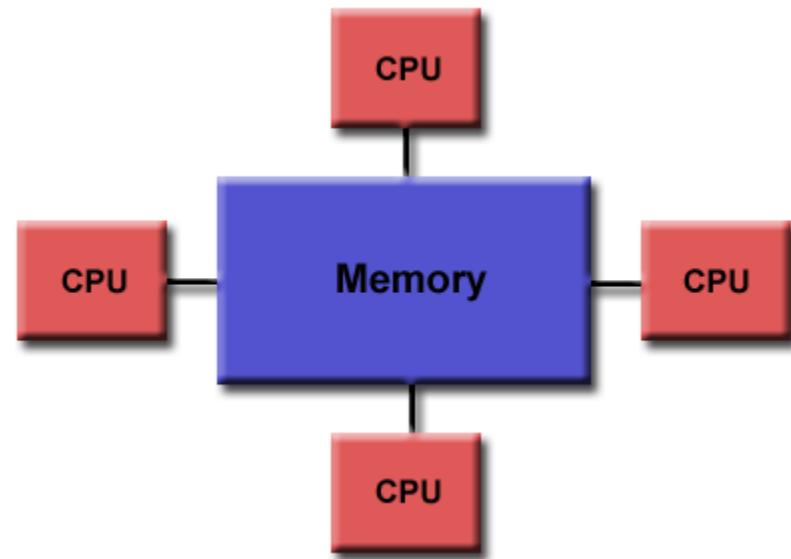
» Memoria Compartida,
Distribuida, Hibrida.

Memoria Compartida

- ▶ Todos los Nodos acceden a la memoria como un espacio global
- ▶ Cada cambio en la memoria es visible por todos los demás nodos



NUMA (Non Uniform Memory Access)



UMA (Uniform Memory Access)

Memoria Compartida

▶ **Ventajas:**

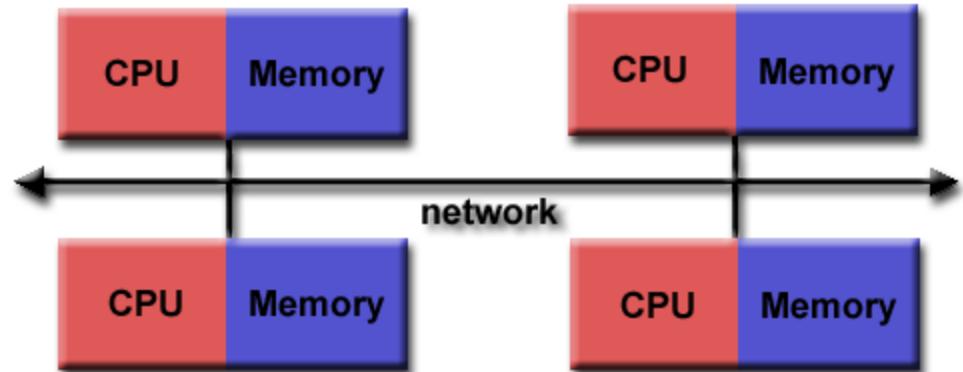
- Espacio de direcciones global es amigable para programar

▶ **Desventajas:**

- Poca escalabilidad (+CPUs → +Trafico)
- Responsabilidad del programador de mantener coherencia
- Precio

Memoria Distribuída

- ▶ Cada Procesador tiene su propia memoria independiente
- ▶ No hay mapeo global
- ▶ El programador define explicitamente como y cuando intercambiar datos



Memoria Distribuída

▶ Ventajas:

- Escalabilidad (+CPUs -> proporcional a la mem)
- Sin overhead de acceso a la memoria
- Precio (COTS)

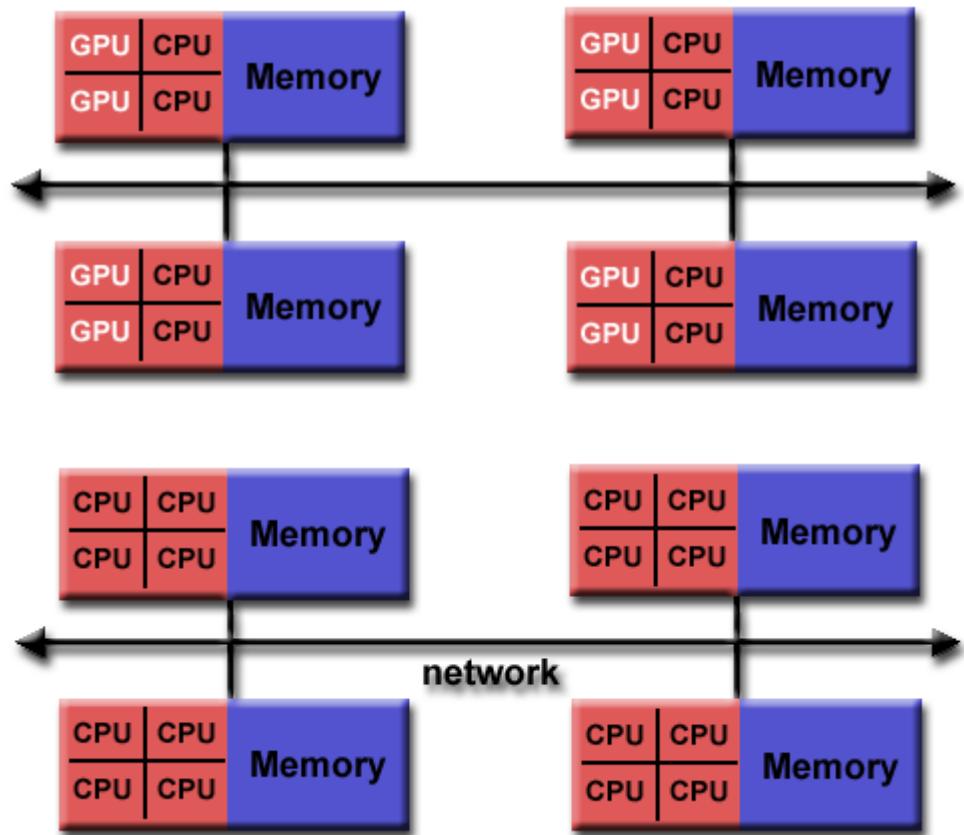
▶ Desventajas:

- Responsabilidad del programador de explicitar pasajes de memorias
- Puede ser complejo mapear estructuras globales

Memoria Compartida-Distribuida

- ▶ La mayoría de las HPC usan una arquitectura mixta:

- ▶ Usan una red overlay de nodos con memoria propia y otros con compartida

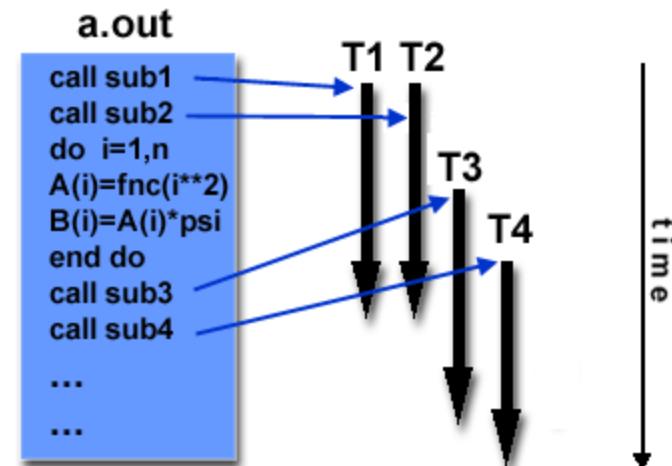


Modelos de Programación

- » Memoria compartida, Threads, MPI, Data Parallel, SPMD, MPMD

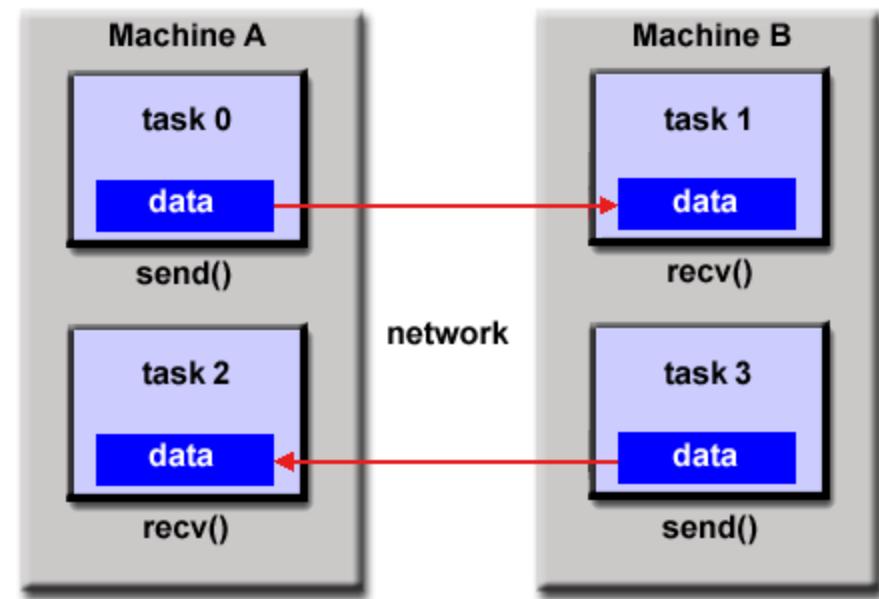
Modelo Threads

- ▶ Un proceso puede tener múltiple caminos concurrentes
- ▶ a.out “crea” subrutinas que se pueden ejecutar concurrentemente que comparten el espacio de memoria
- ▶ Se logra por medio de directivas de compilador
- ▶ Ejemplo: OpenMP

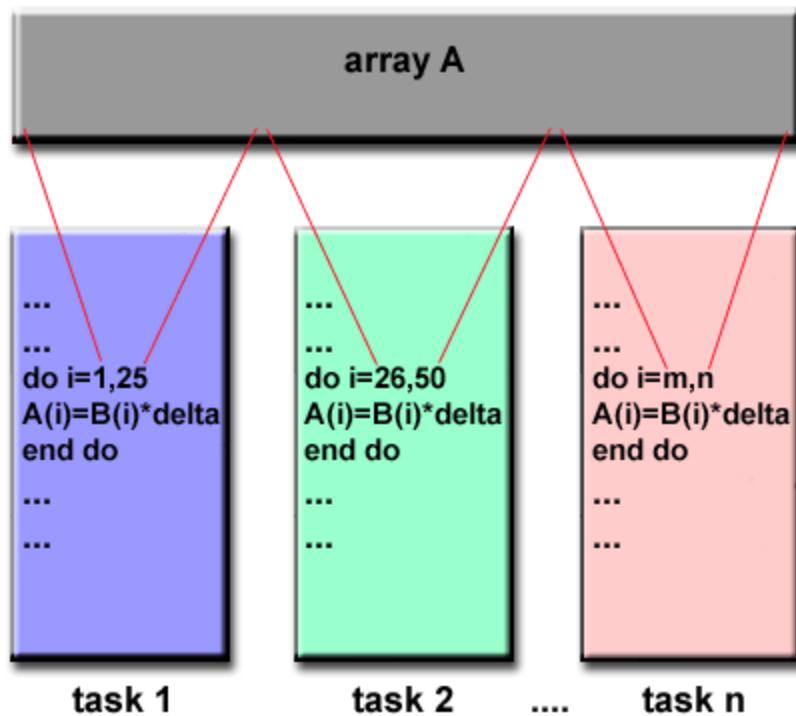


Modelo de Pasaje de Mensajes

- ▶ Procesos tienen su propia memoria local
 - ▶ Se comunican por medio de transmisión de mensajes (Tx y Rx routines) logradas por llamadas a subrutinas de librerías
-
- ▶ Ejemplo
Estándard MPI

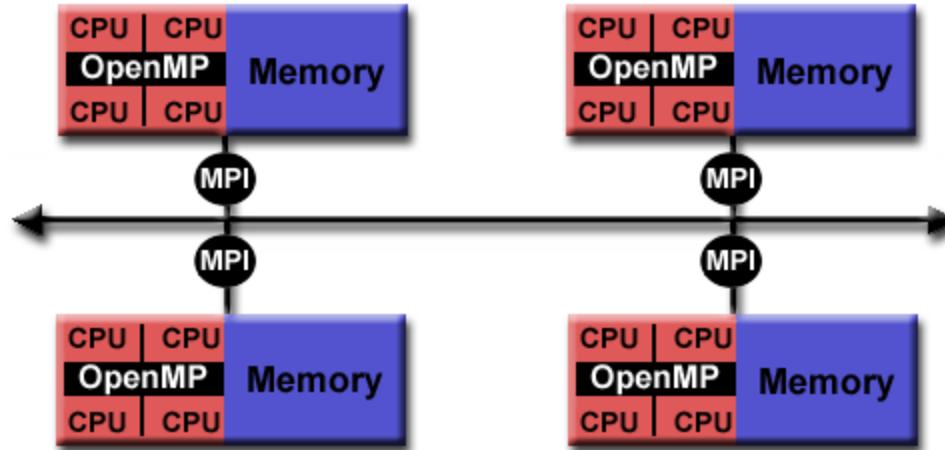


Modelo de Paralelismo de Datos



- ▶ La mayor parte del trabajo se enfoca en una estructura de datos dada (array)
- ▶ Cada proceso trabaja sobre una porción exclusiva del dato
- ▶ *Mem Compartida:* todos acceden
- ▶ *Mem Distribuida:* se divide el dato

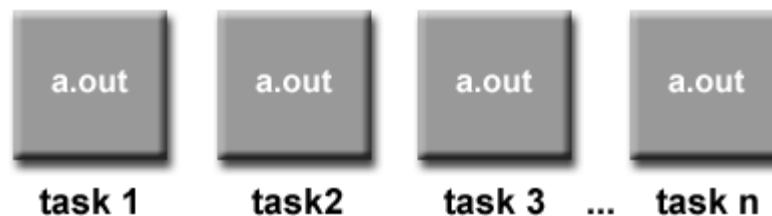
Modelos Híbridos



- ▶ Más clásico: Message passing model (MPI) + threads model (OpenMP)
- ▶ Más popular: MPI + GPU (Graphics Processing Unit)

Modelo SPMD

- ▶ Single Program Multiple Data (SPMD):
- ▶ Se puede ver como un SIMD de más alto nivel donde cada instrucción es ahora un programa
- ▶ Los programas SPMD son lógicamente identicos pero están diseñados para permitir diferentes “branchs”. Diferentes programas pueden ejecutar diferentes instrucción.

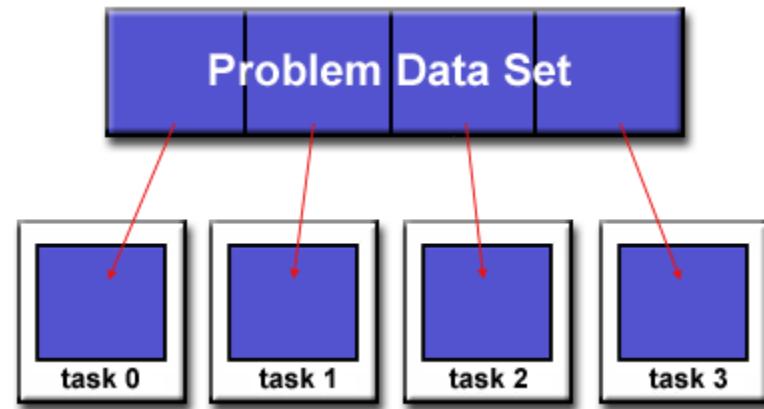


Parámetros de Diseño

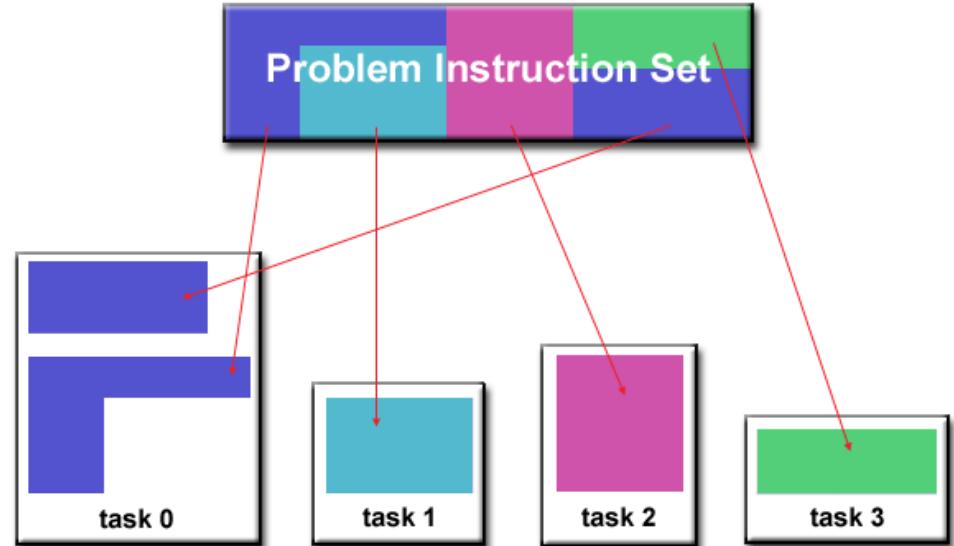
» Partición, Amdahl's Law

Partición del problema

- ▶ Domain Decomposition:



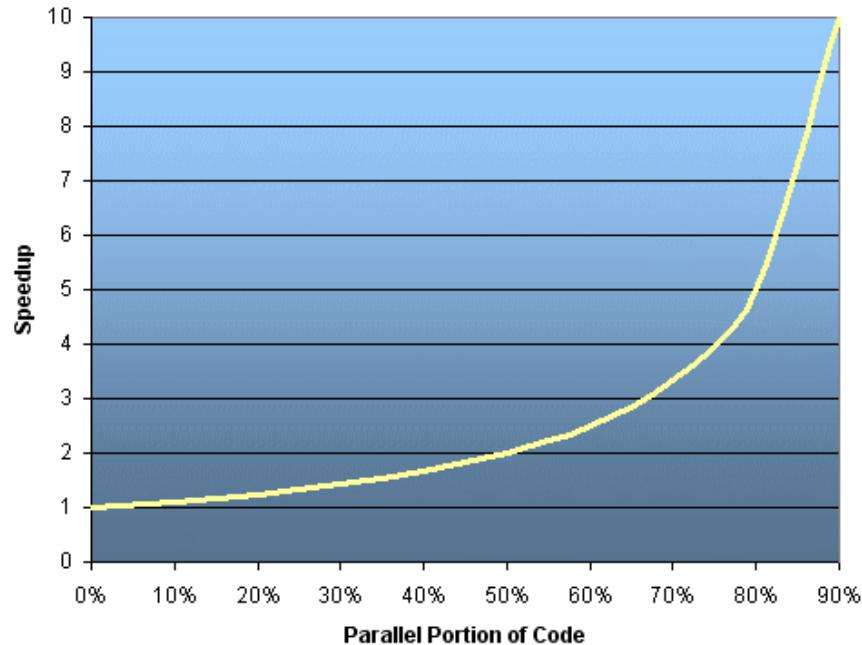
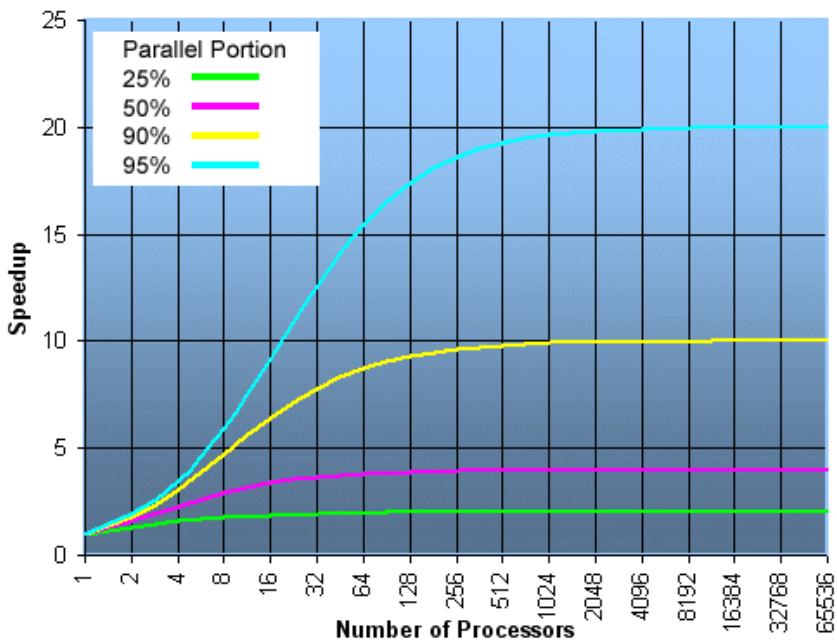
- ▶ Functional Decomposition:



Ley de Amdahl's

$$\text{speedup} = \frac{1}{1 - P}$$

P=fracción de código paralelizable



$$\text{speedup} = \frac{1}{\frac{P}{N} + \frac{S}{N}}$$

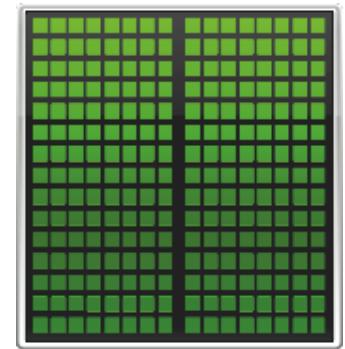
N=número de procesadores

GPGPU

» General
Purpose
Graphic
Processing
Unit

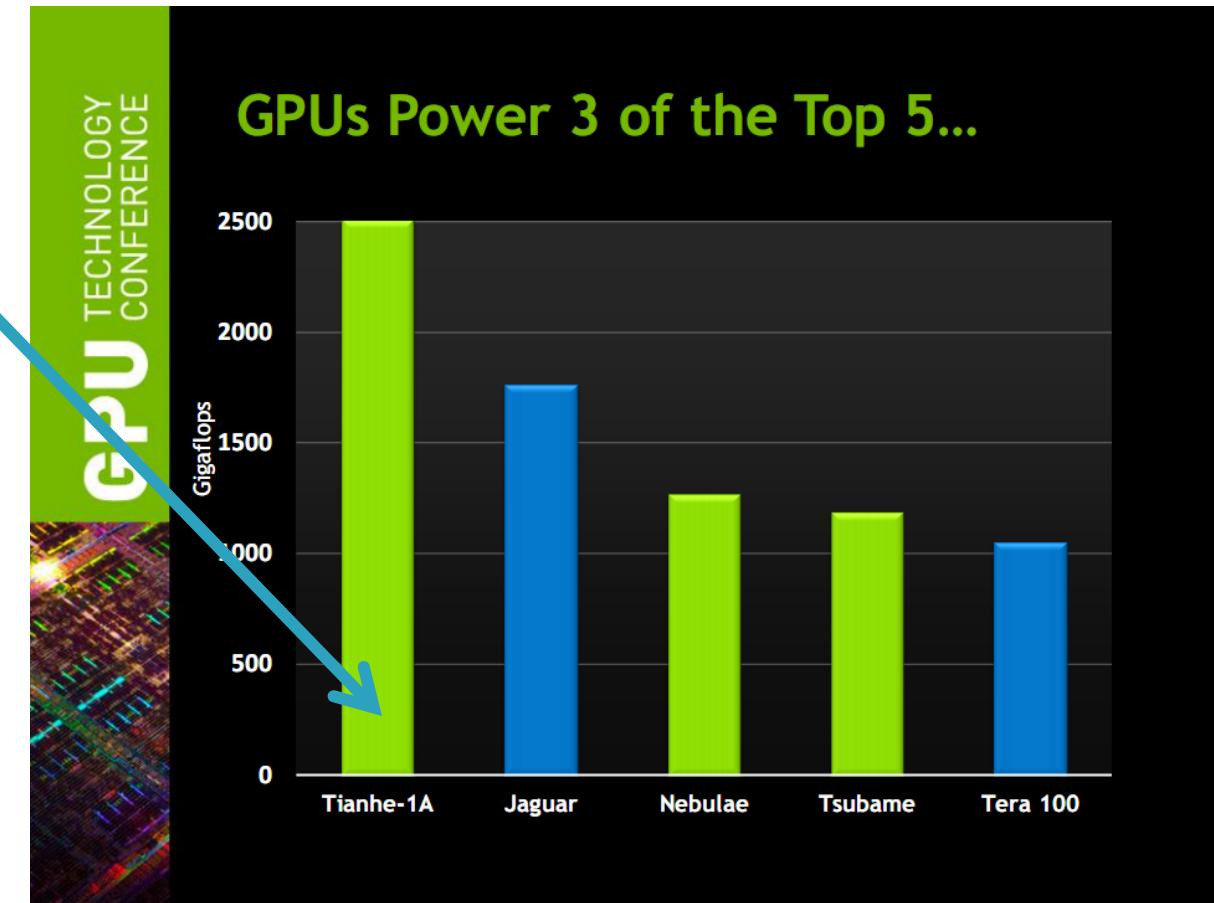
Que es GPU?

- ▶ GPU = Graphics Processing Unit
 - Highly Parallel
 - Highly Programmable
 - Commodity Hardware (“Desktop Supercomputing”)
- ▶ Nvidia’s GTX580: 16 x 32-wide multiprocessors
 - 512 ALUs
- ▶ 24,576 concurrent threads



GPU en Supercomputing

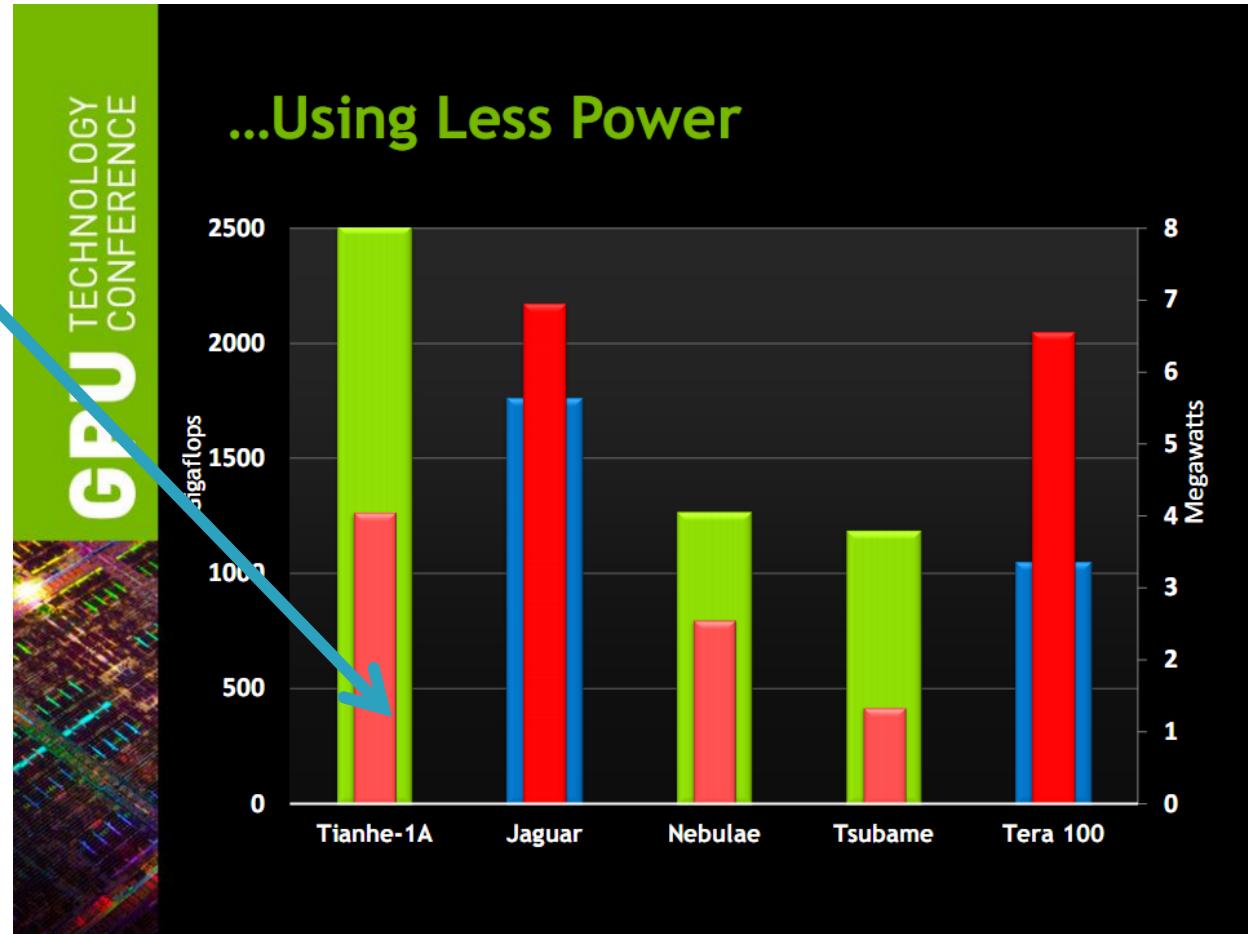
7,168 Nvidia
Tesla M2050



*Slide from GTC 2011, *GPU Computing: Past, Present and Future*, David Luebke, NVIDIA

GPU en Supercomputing

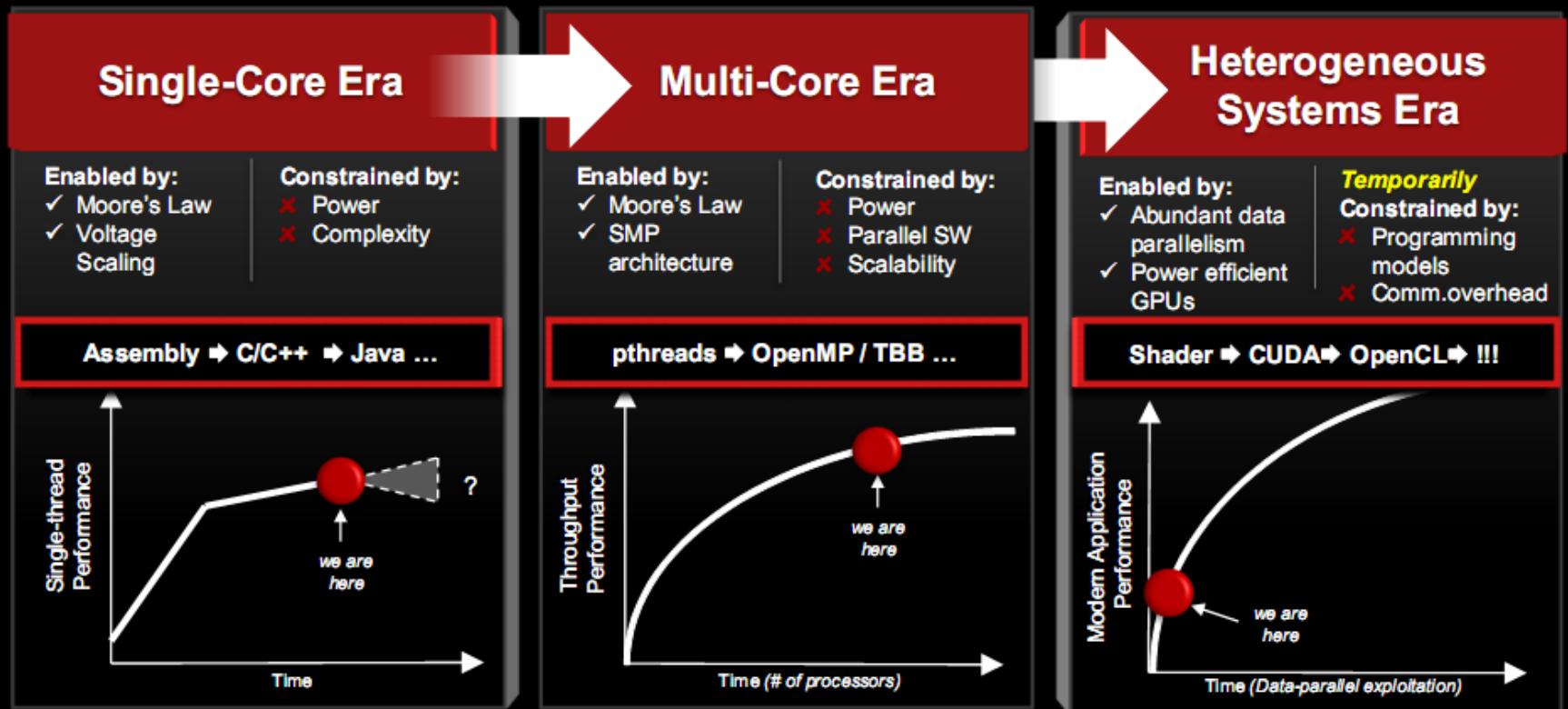
7,168 Nvidia
Tesla M2050



*Slide from GTC 2011, *GPU Computing: Past, Present and Future*, David Luebke, NVIDIA

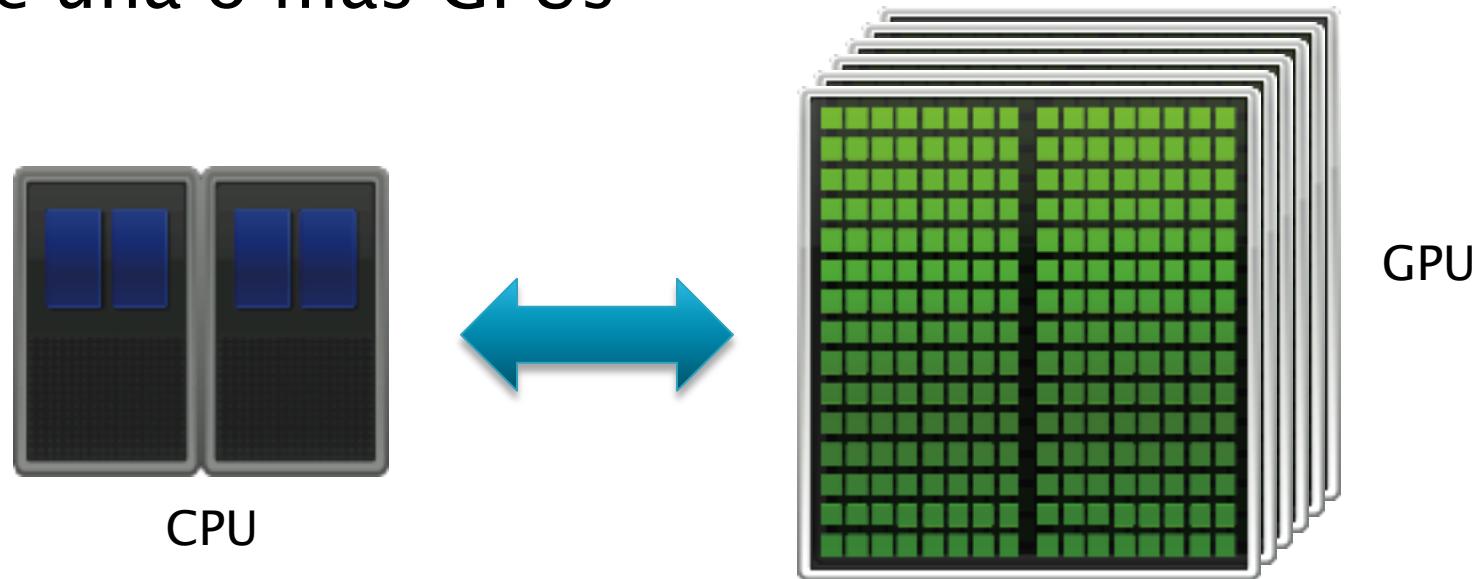
GPU en el tiempo

A NEW ERA OF PROCESSOR PERFORMANCE



Limitaciones de GPGPU

- ▶ El problema es que GPGPU propone modelos muy centralizados en explotar el paralelismo de una o mas GPUs



- ▶ NO responden al modelo mas genérico de sistemas heterogeneos

Hacia sistemas heterogeneos

- ▶ GPGPU NO fue pensado para aprovechar diferentes dispositivos al máximo
- ▶ NO permite explotar TODAS las capacidades de un sistema.
- ▶ Filosofía “*Exploit what the user paid for*”

OpenCL

» Un framework para sistemas heterogéneos

Sistemas Heterogéneos

► Realidad: Sistemas Heterogéneos:

- CPUs: Múltiples Núcleos
- GPUs: De Fixed función a Programable Pipeline
- Diferencias:
 - Arquitecturas
 - Jerarquías de Memoria
 - Operaciones Vectoriales



► Oportunidad:

- Aprovechar al máximo estos sistemas

► Resolución:

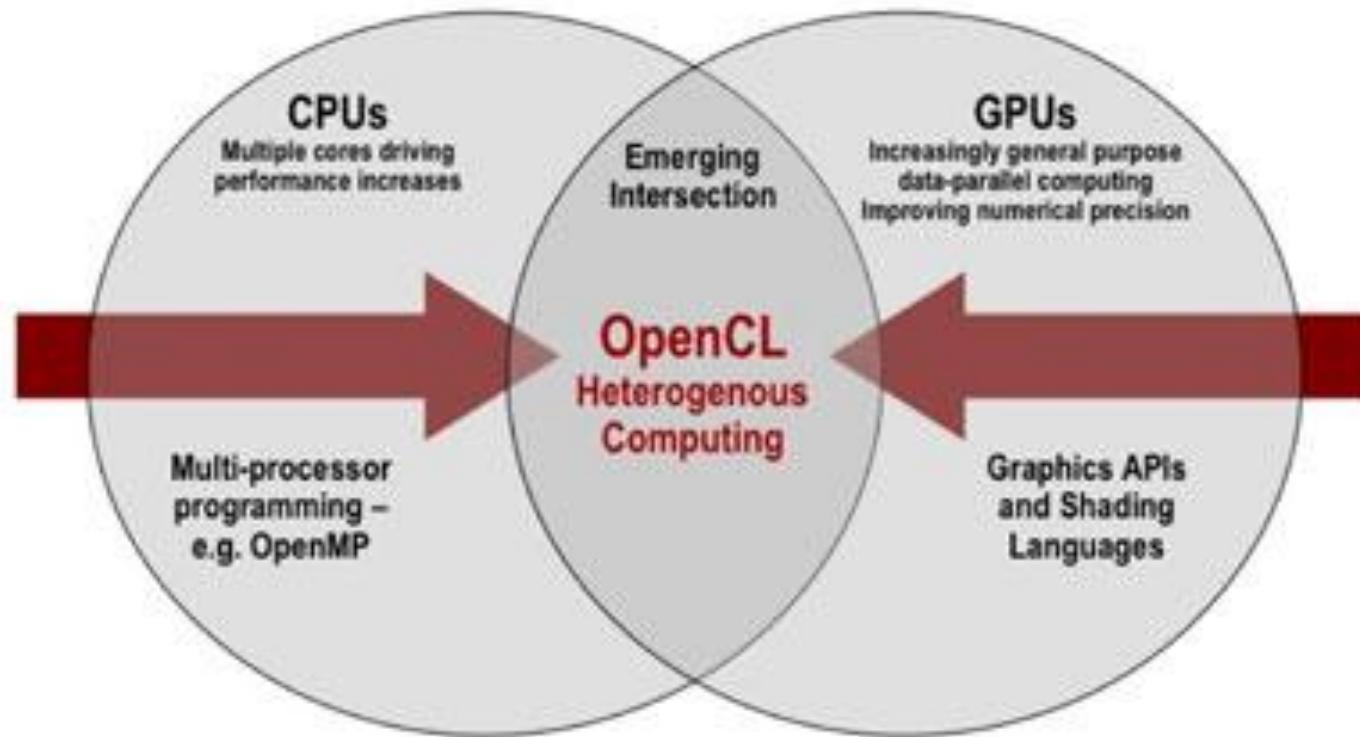
- Intel, AMD, Apple, IBM, Nvidia, entre otros se unen al Khronos Group para crear OpenCL



OpenCL 1.2

Fundamentos OpenCL

Processor Parallelism

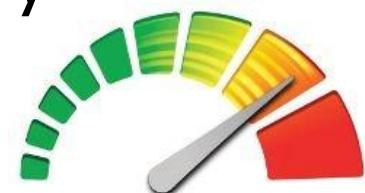


OpenCL – Open Computing Language

Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

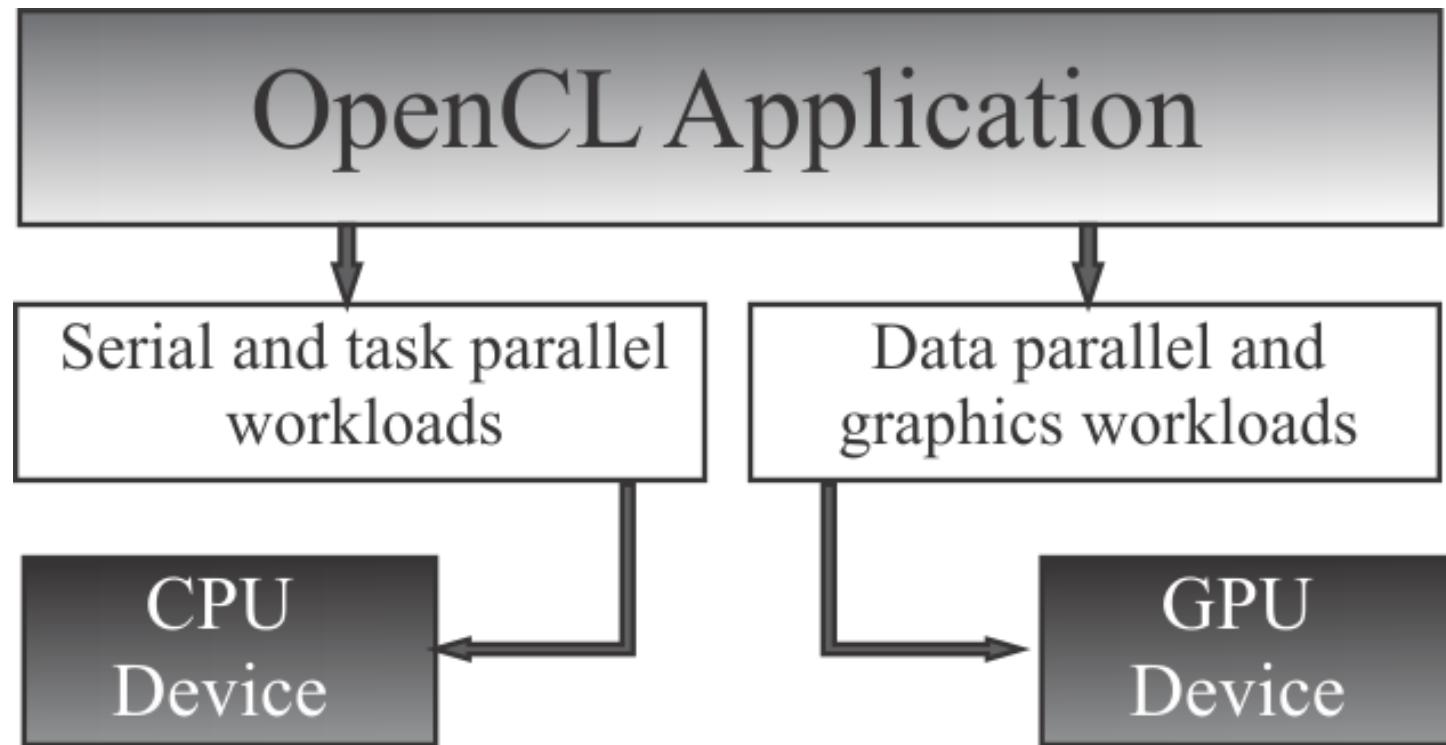
Fundamentos OpenCL

- ▶ Estandar abierto, Cross-platform, Cross-Operating System, Cross-vendor
- ▶ Orientado a la programación de colecciones de CPU, GPU, DSPs, entre otros
- ▶ Incluye dispositivos portátiles, PCs, Servidores y Mainframes.
- ▶ Permite al programador asignar tareas a los dispositivos mas convenientes
- ▶ Explota el paralelismo a nivel sistema y dispositivo
- ▶ Código Único basado en C99



Fundamentos OpenCL

- ▶ OpenCL en un sistema heterogéneo



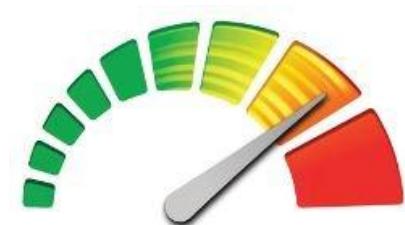
Fundamentos OpenCL

- ▶ El objetivo principal de OpenCL es lograr escribir código eficiente y portable simultáneamente
- ▶ “El Java del HPC” -> No, java oculta el hardware, OpenCL lo expone
- ▶ OpenCL no es solo un lenguaje, es un Framework para sistemas heterogéneos

Divide and Conquer...

OpenCL:

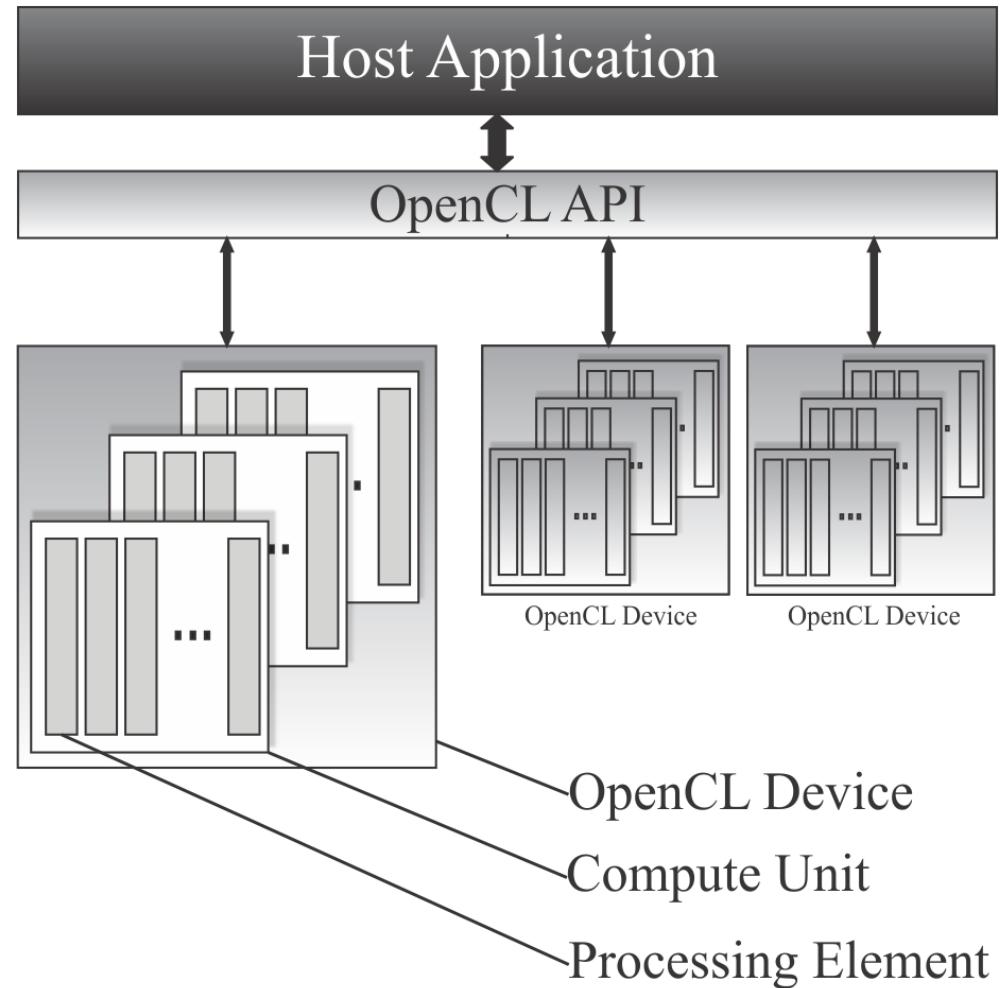
- ▶ Modelo de Plataforma
- ▶ Modelo de Ejecución
- ▶ Modelo de Memoria
- ▶ Modelo de Programación



OpenCL 1.2

Modelo de Plataforma

- ▶ **Device:** Dispositivo compatible con OpenCL (CPU, GPU)
- ▶ **Compute Unit:** Conjunto de procesadores (Stream multiprocessor in Nvidia)
- ▶ **Proc. Element:** Elemento unico de procesamiento (ALU o streaming processor in Nvidia)

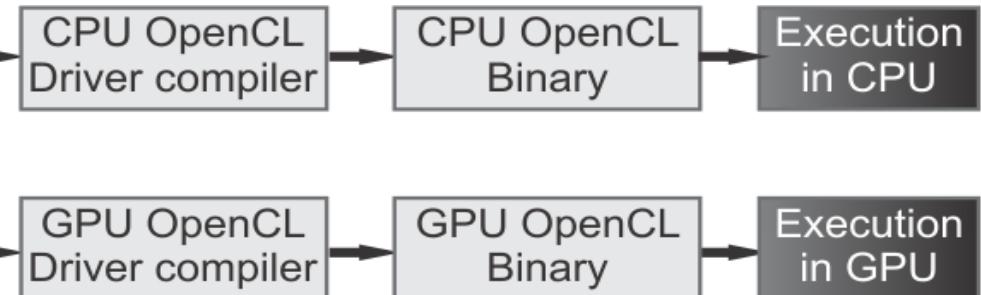


Modelo de Ejecución

- ▶ **Host:** Programa central, gestiona dispositivos
 - Escrito en C, C++, NET, Py, y otros. APIs y Wrappers populares
- ▶ **Kernels:** Similares a una función en C
 - Declarados en el Host.
 - Compilado en tiempo de ejecución (Portabilidad)
 - Por device (Performance)

Kernel Source Code

```
kernel void
horizontal_reflect(read_only image2d_t src,
                    write_only image2d_t dst)
{
    int x = get_global_id(0); // x-coord
    int y = get_global_id(1); // y-coord
    int width = get_image_width(src);
    float4 src_val = read_imagef(src, sampler,
                                (int2)(width-1-x,
y));
    write_imagef(dst, (int2)(x, y), src_val);
}
```



Modelo de Ejecución

- ▶ Kernels: Similares a una función en C

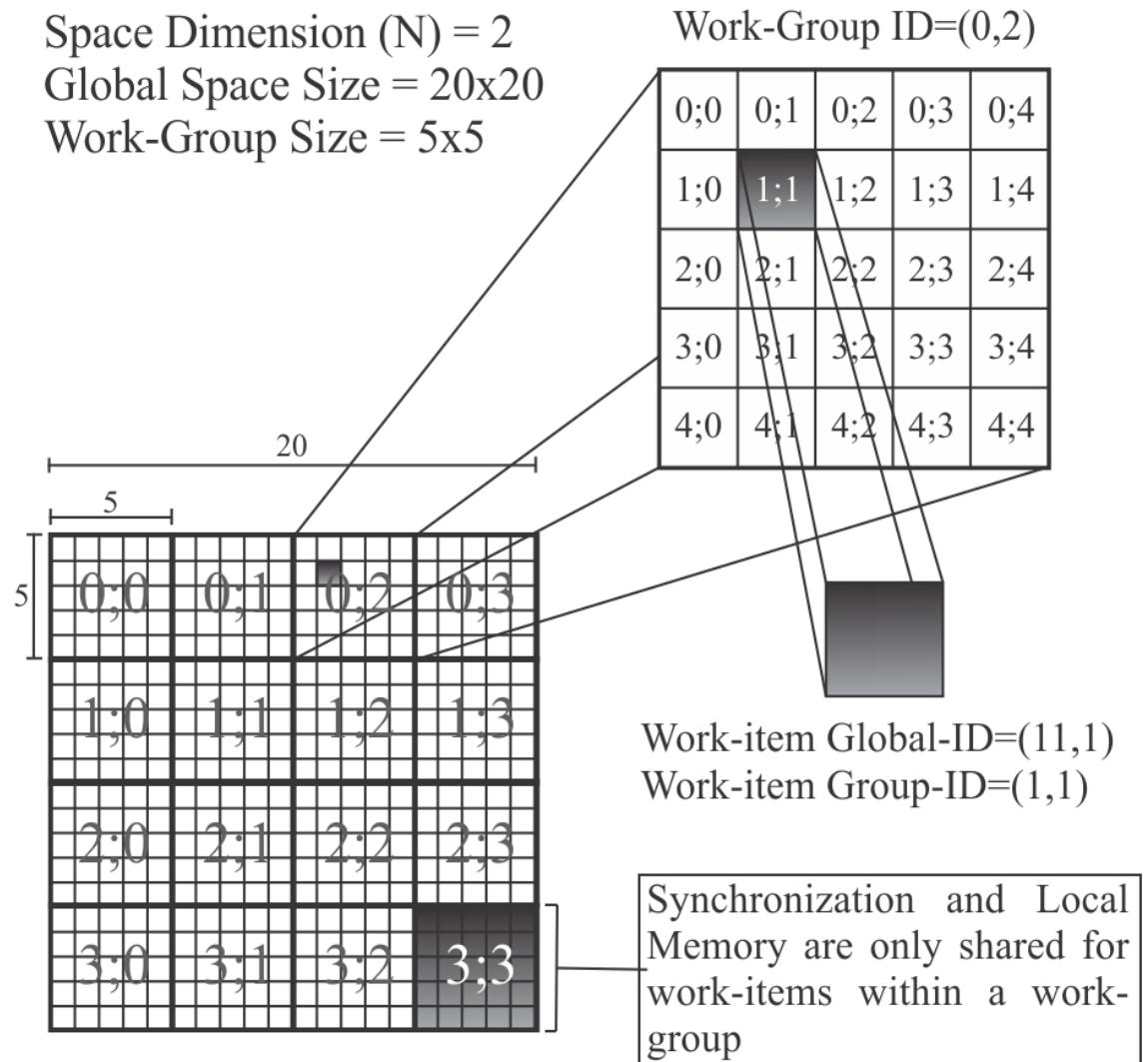
```
__kernel void  
vadd(__global const float *a,  
      __global const float *b,  
      __global float *result) {  
    int id = get_global_id(0);  
    result[id] = a[id] + b[id];  
}
```

Modelo de Ejecución

▶ Kernels:

- Puestos en cola por el Host
- Definidos en una dimensión “N = 1, 2, o 3”
- Cada elemento “work-item” es un punto (x), (x,y), o (x,y,z) en el espacio “N”

Space Dimension (N) = 2
Global Space Size = 20x20
Work-Group Size = 5x5

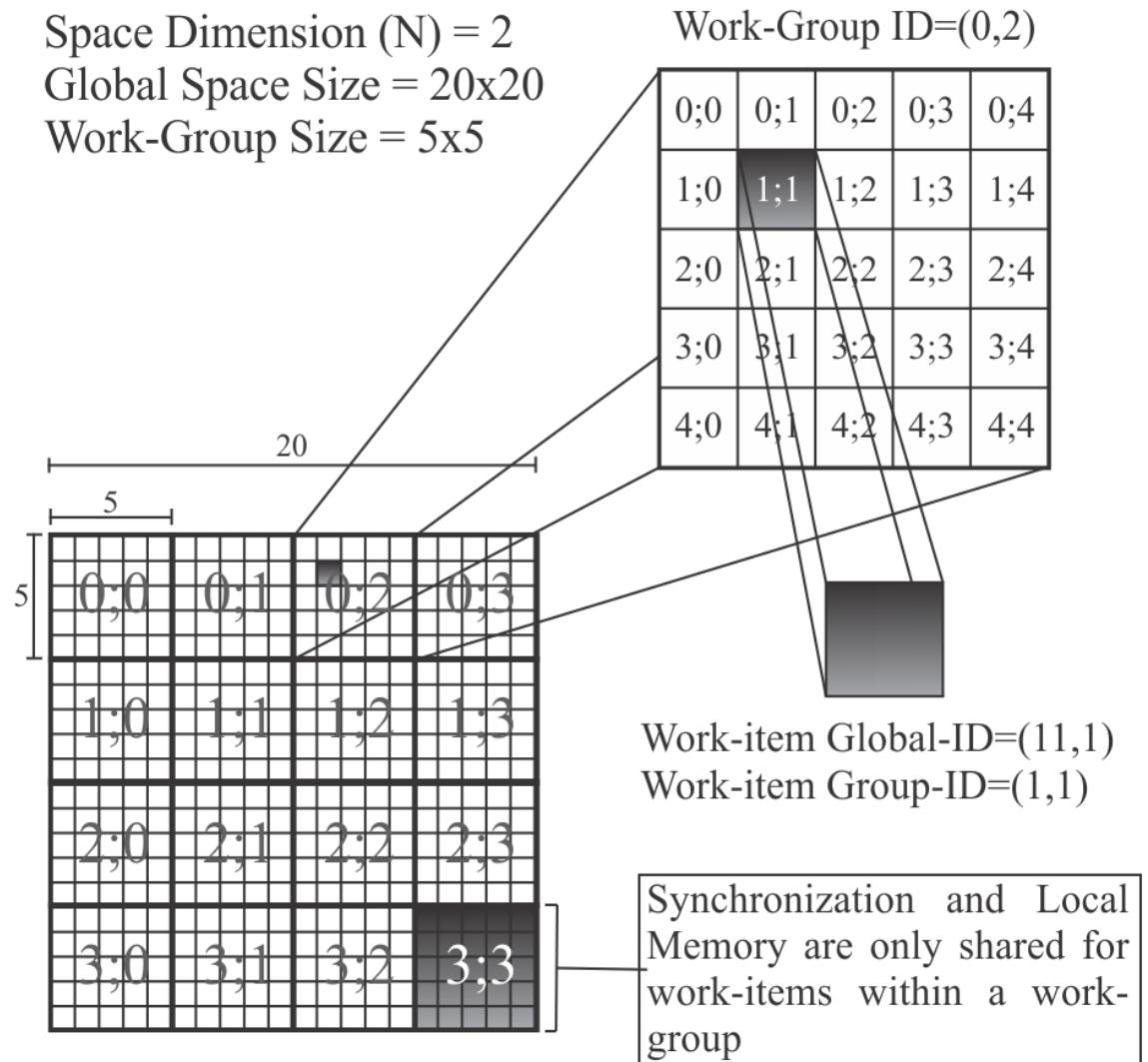


Modelo de Ejecución

▶ Kernels:

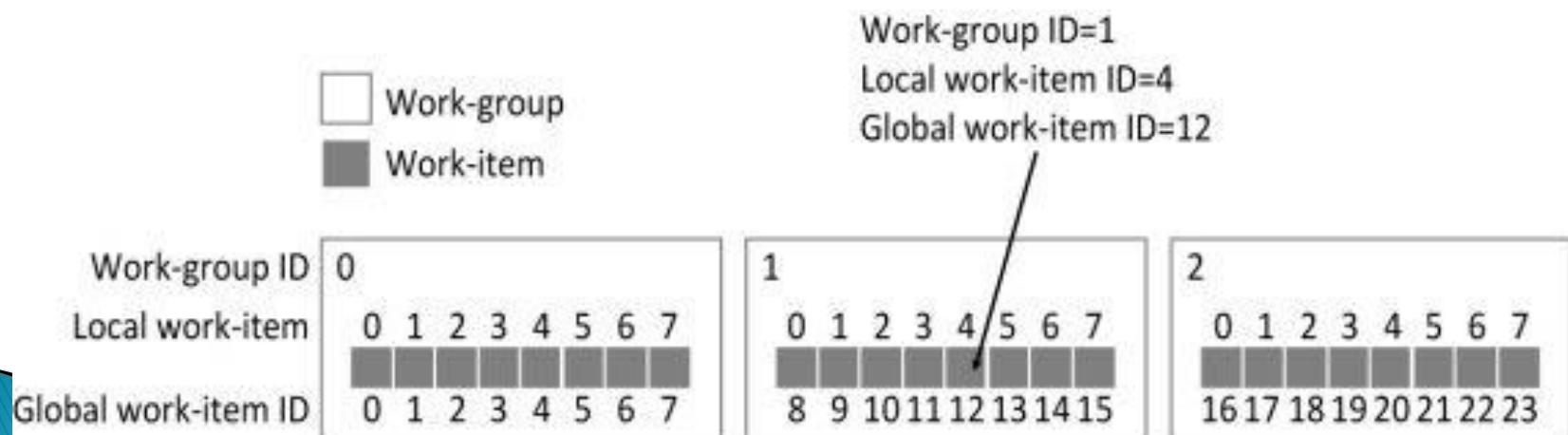
- “work-group” son conjunto de “work-items” en “N” en los que se garantiza ejecución concurrente en una unidad de procesamiento.
- Comparten memoria y syncro

Space Dimension (N) = 2
Global Space Size = 20x20
Work-Group Size = 5x5



Modelo de Ejecución

- ▶ Kernels: Cada Work-Item ejecuta el mismo código, aunque pueden tomar “branches” diferentes (SPMD)
 - Diferencia de Datos
 - Diferencia de Global IDs
 - Diferencia de Local IDs

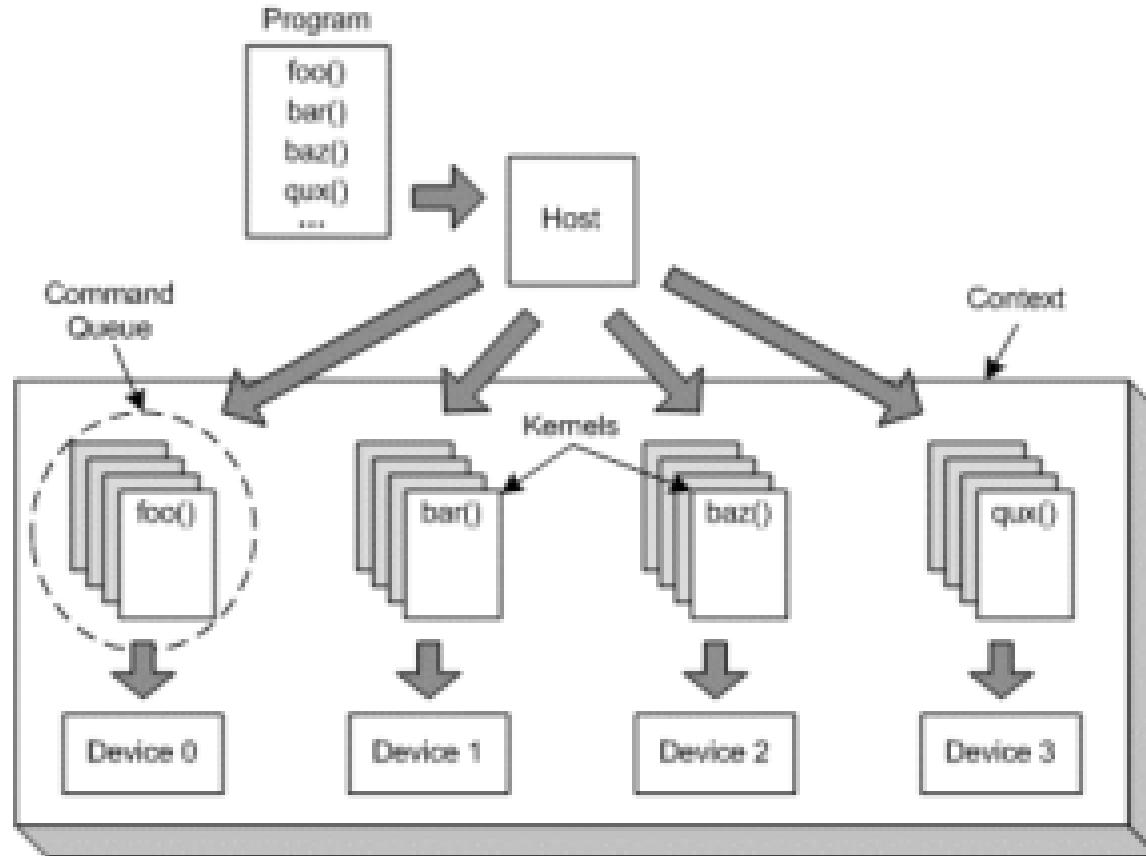


Modelo de Ejecución

- ▶ Host:
- ▶ Administrar la ejecución de OpenCL por medio de “contextos” (conjunto de OpenCL Devices)
- ▶ Pone en cola (enqueue) escritura/lectura de memoria, programas (kernels), y sincronismos
- ▶ Las colas de ejecución se garantizan ordenadas. NO así la ejecución de Kernels

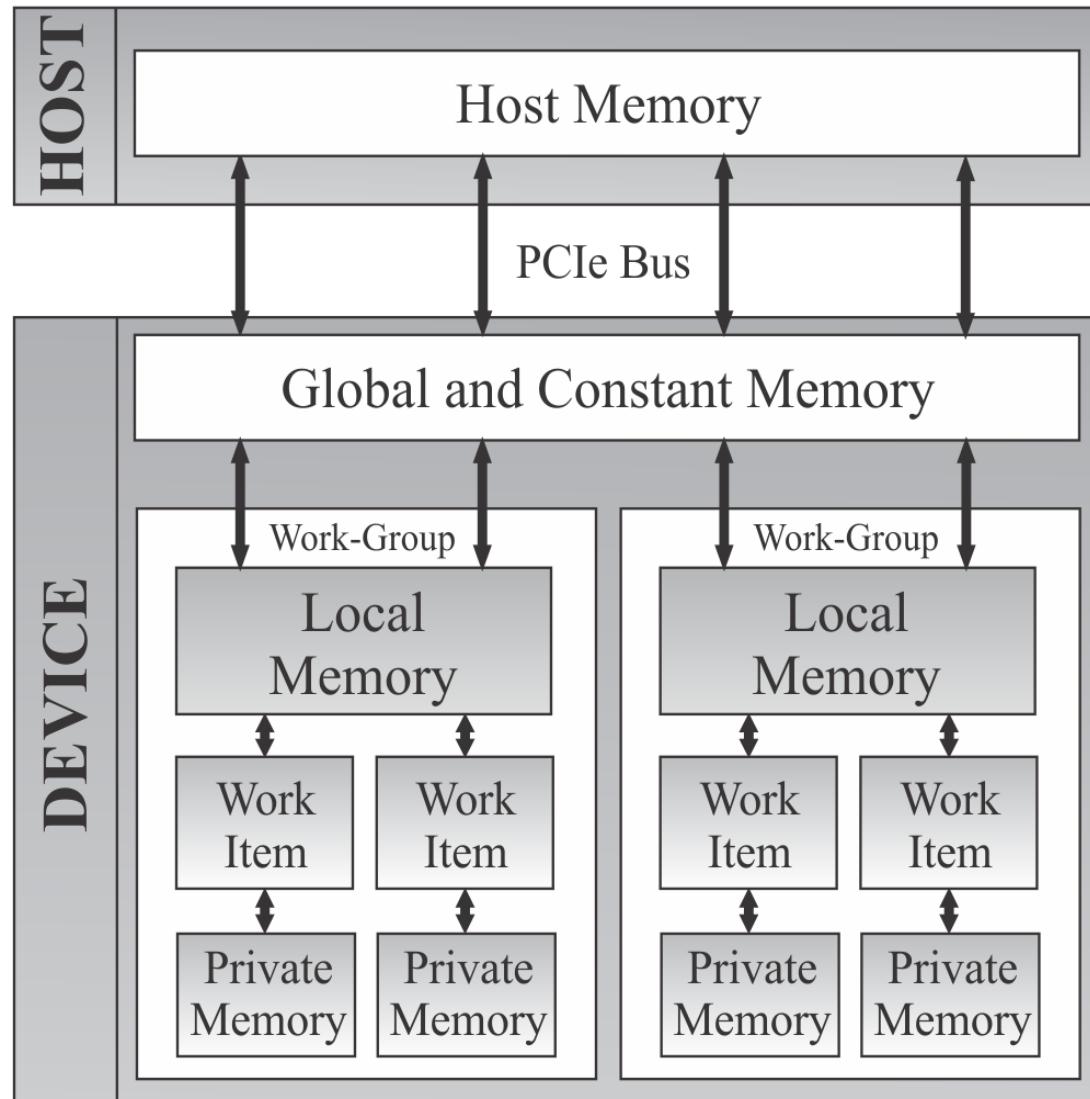
Modelo de Ejecución

▶ Host:



Modelo de Memoria

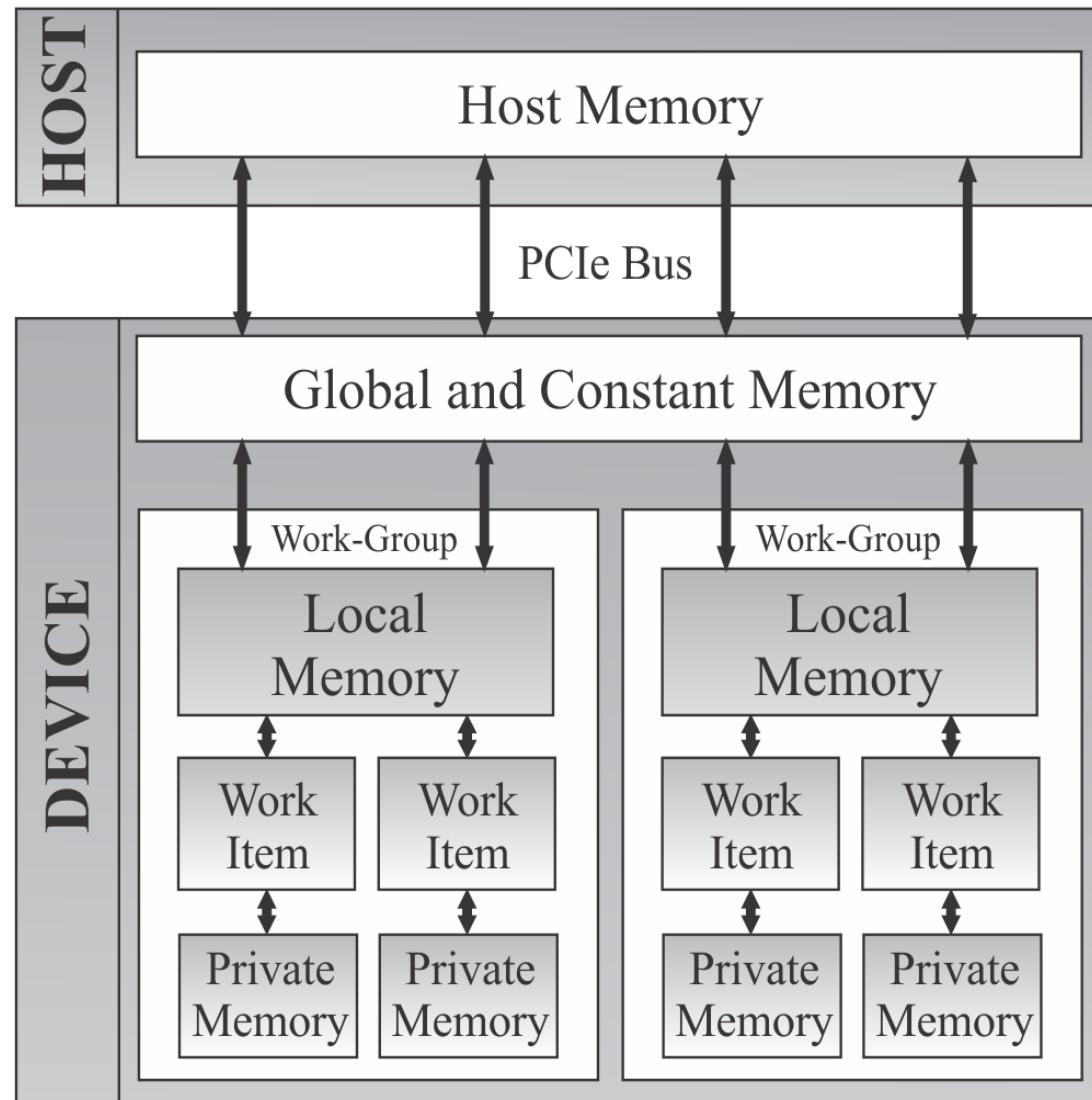
- ▶ **Host:** Host side
- ▶ **Global:** Accesible para todo los WI
- ▶ **Constant:** Read Only
- ▶ **Local:** Accesible para un Work-Group
- ▶ **Private:** Accesible para un Work-Item



Modelo de Memoria

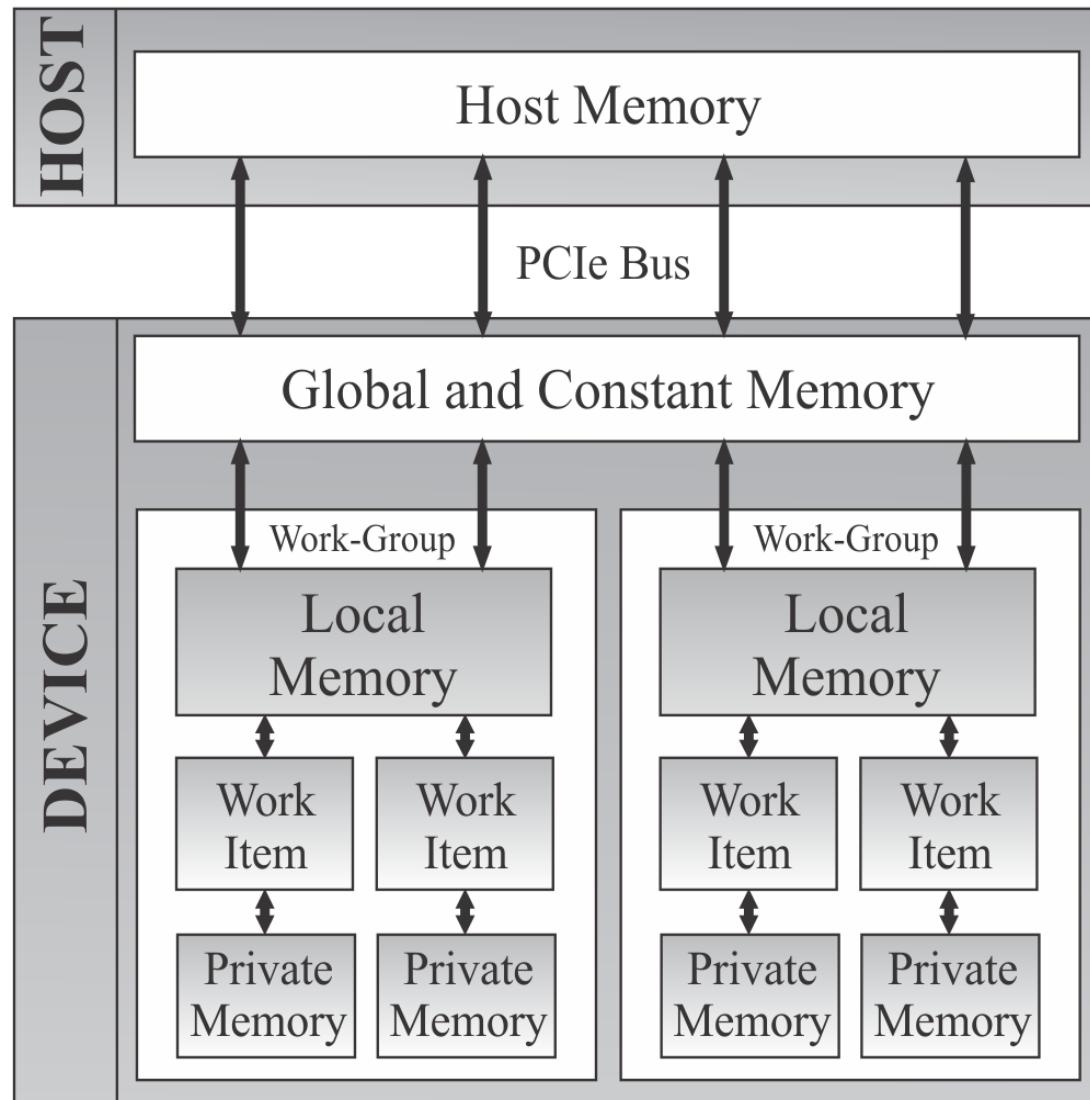
- ▶ **Modelo: Relaxed
(incosistente)**

i.e: Necesita sincronismo explícito para mantener la memoria consistente (Barriers)



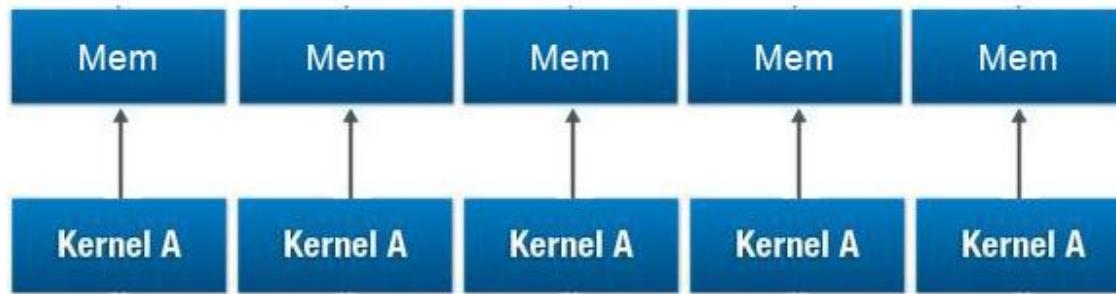
Modelo de Memoria

- ▶ DataRates:
 - ▶ PCIe: 1Gbps
 - ▶ PCIe_{x16}: 16Gbps
- ▶ GPU_{RAM}: 150Gbps
- ▶ *“Importancia de minimizar las mem. transfer”*

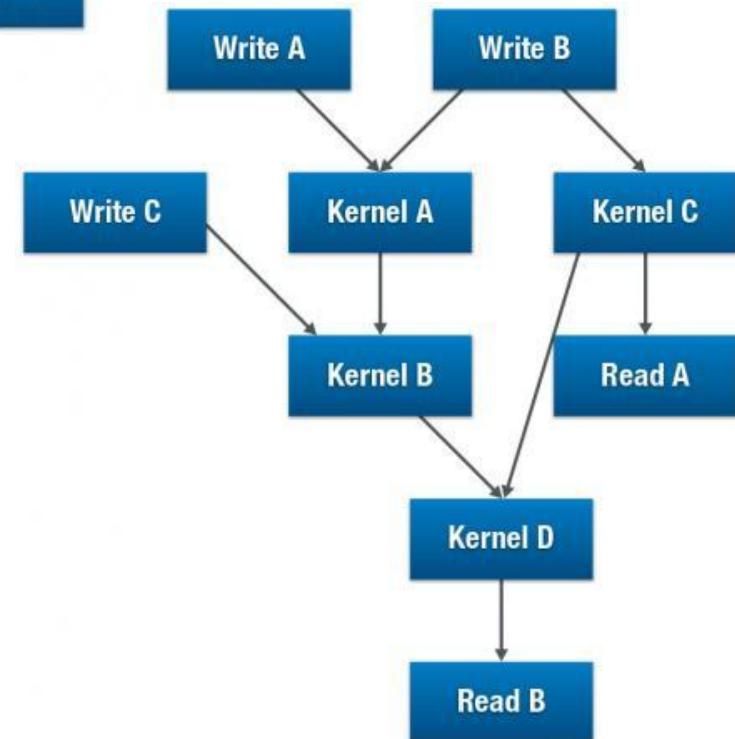


Modelo de Programación

- ▶ Modelo de Paralelismo de Datos:

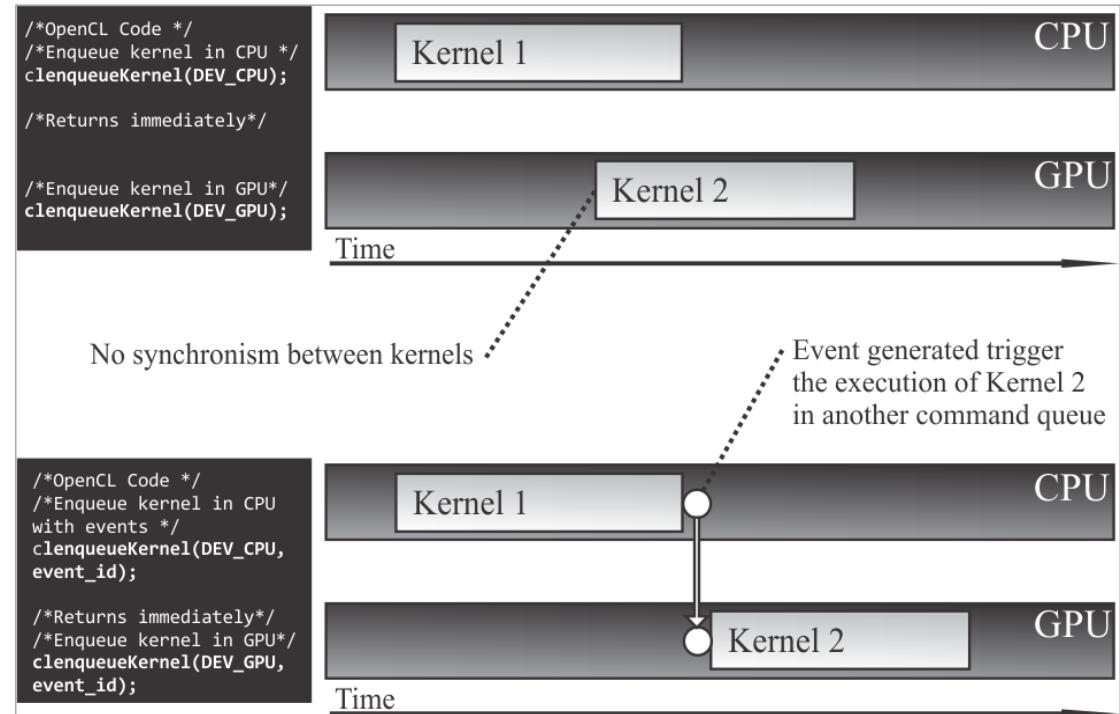


- ▶ Modelo de Paralelismo de Tareas:



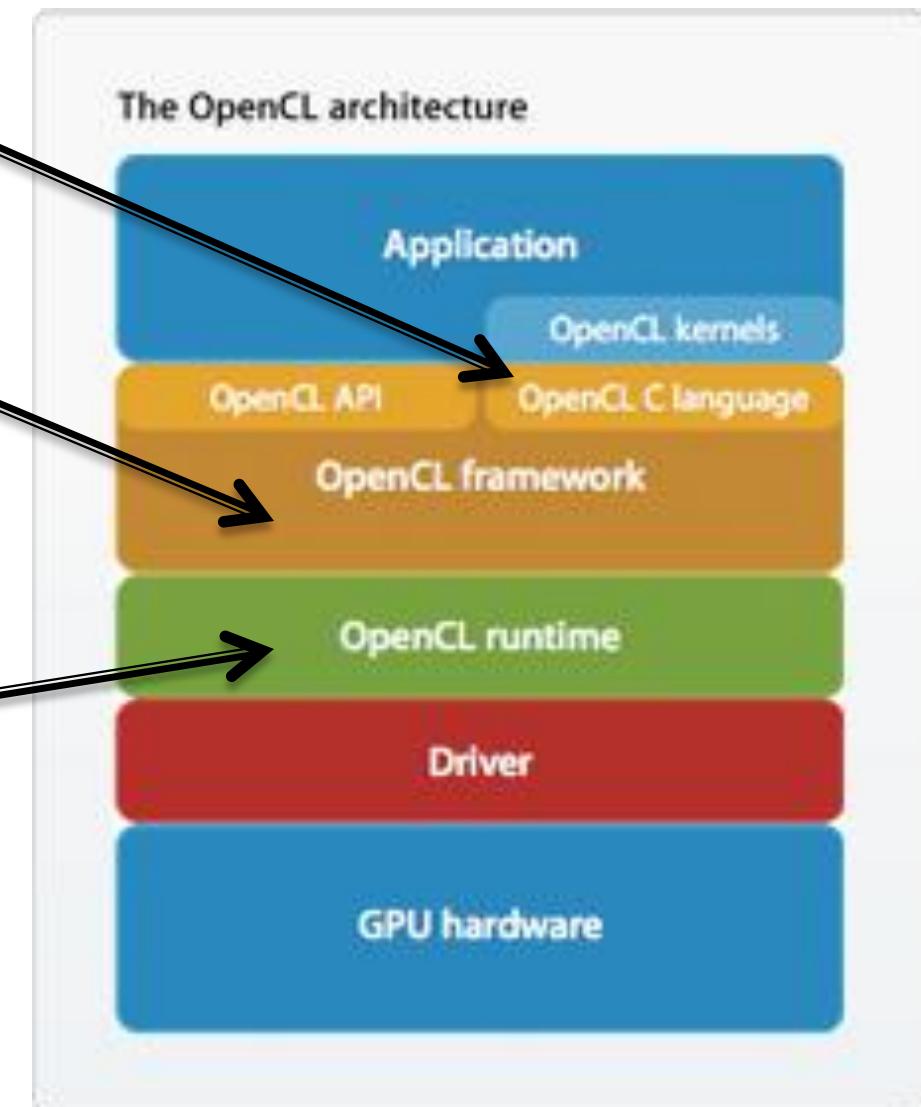
Modelo de Programación

- ▶ Sincronismo en Device:
 - Barriers: **barrier();** Todos los Kernels esperan en la barrera a los demás
- ▶ Sincronismo en Host:
 - Eventos



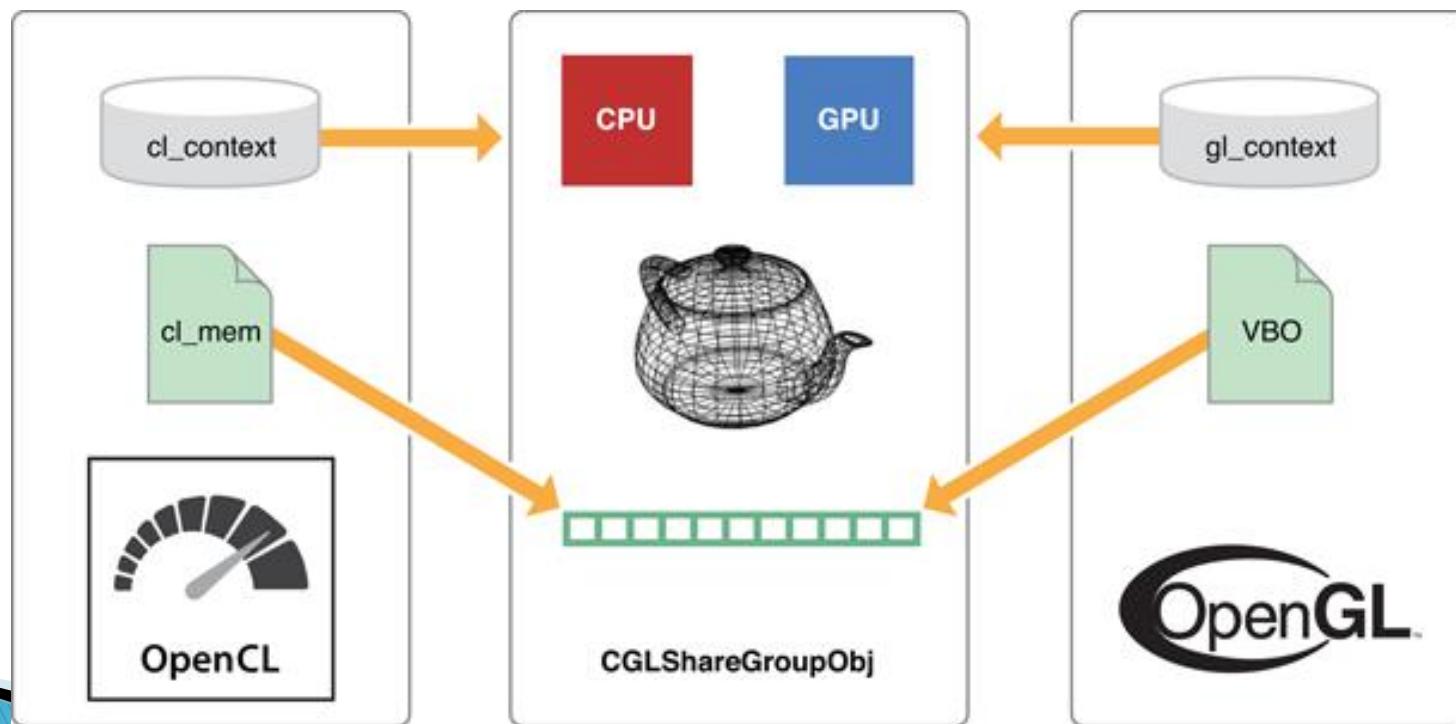
FrameWork

- ▶ **Compilador:** interpreta el lenguaje OpenCL (kernels)
- ▶ **Plataforma:** permite por medio del SO acceder a los dispositivos OpenCL
- ▶ **RunTime:** Gestiona un dispositivo en particular



FrameWork

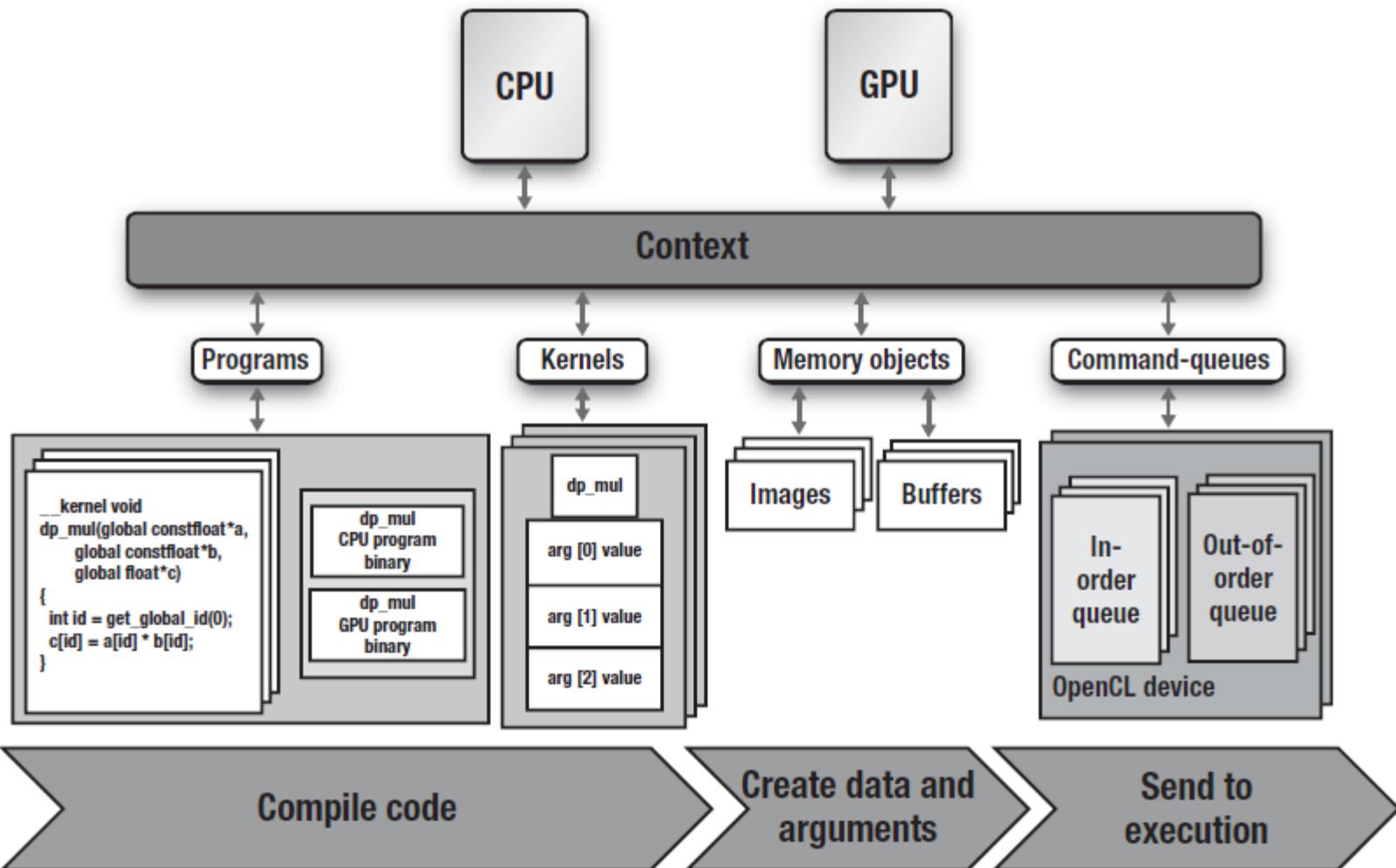
- ▶ Ventaja de usar GPUs con HPC:
 - Comparten espacios de memoria... es muy facil realizar cálculos con OpenCL y trabajarlos con shaders para una aplicación 3D



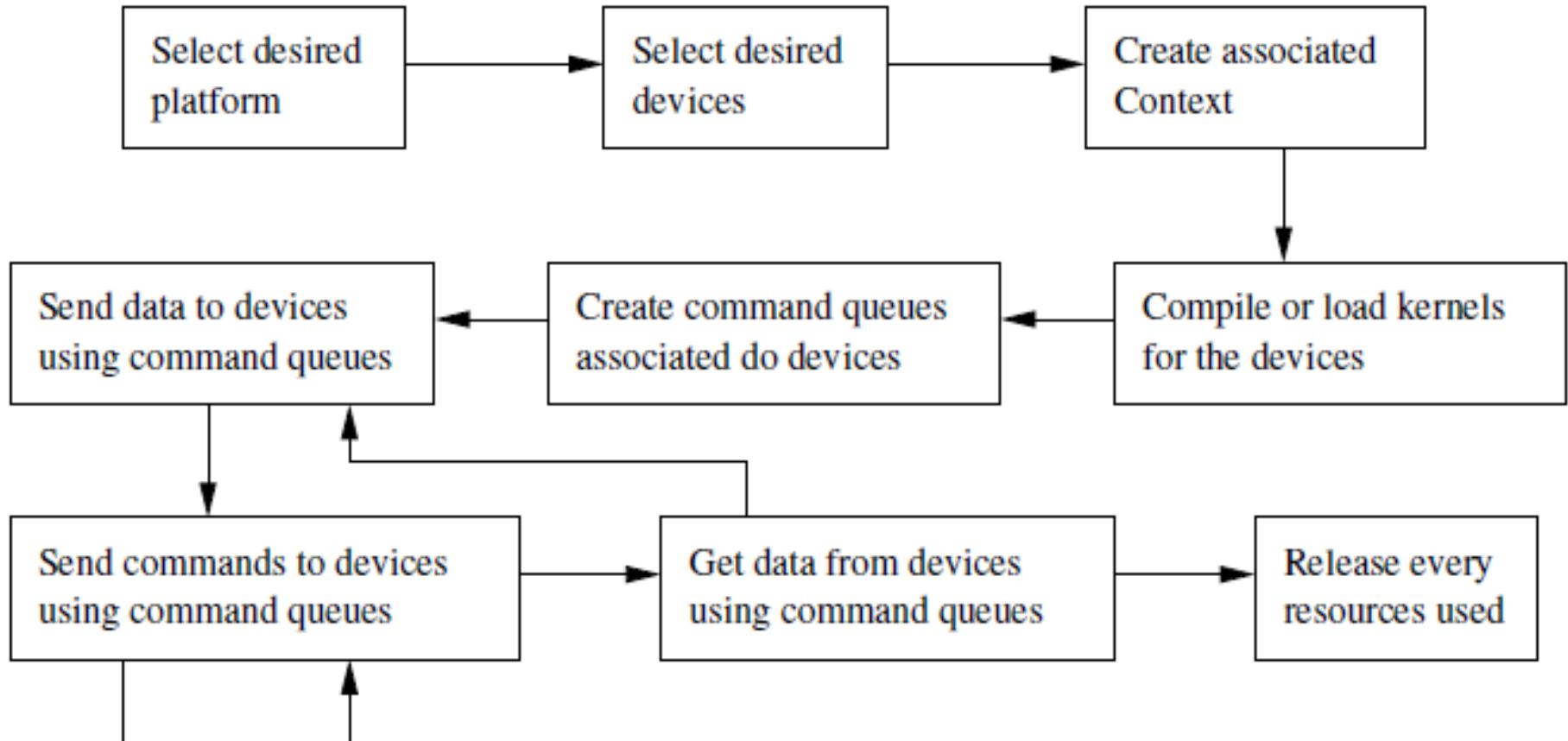
Implementación – Host

» Código Host OpenCL
OpenCL API

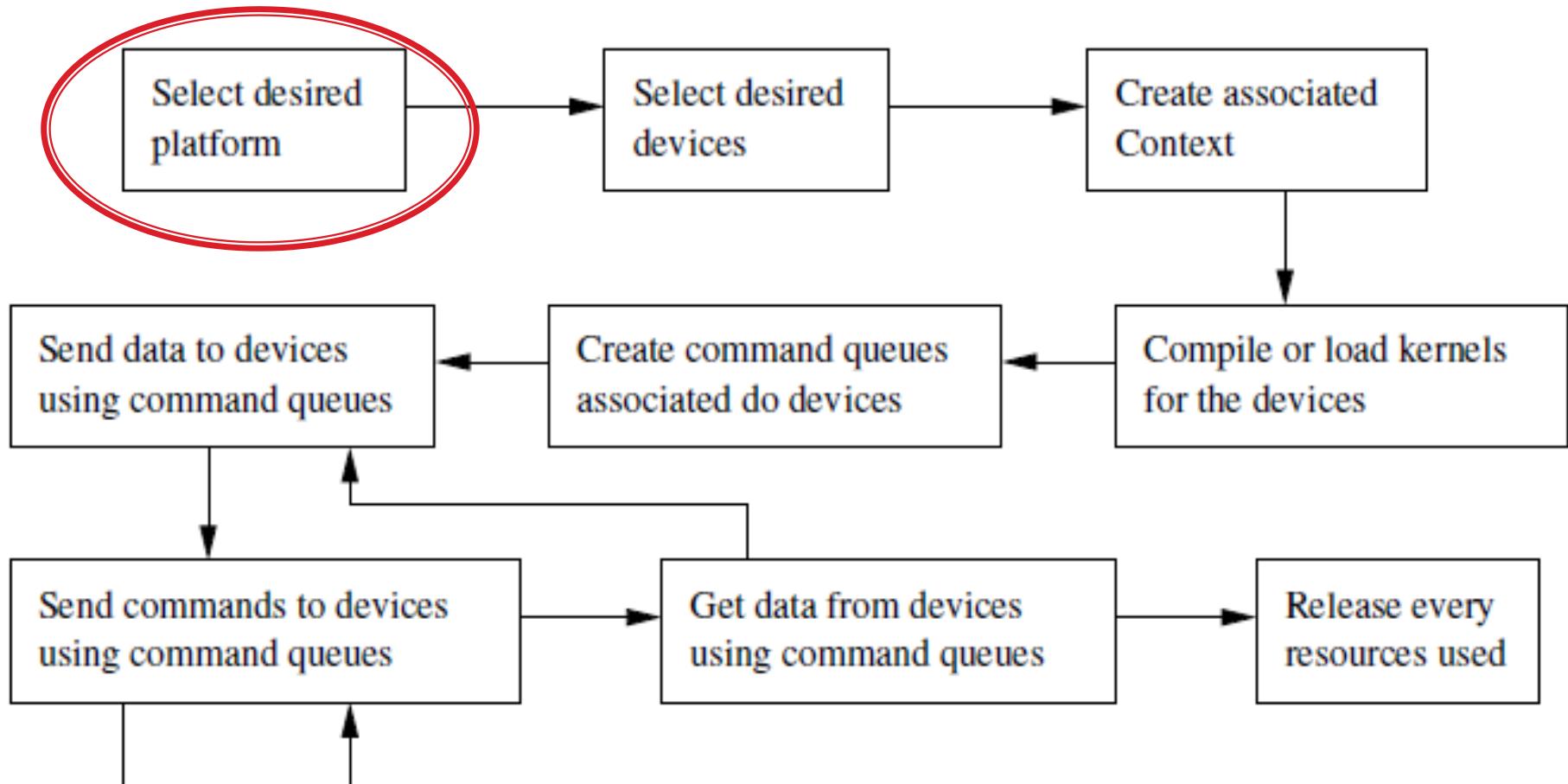
WorkFlow



WorkFlow – Host



Selección de Plataforma



Selección de Plataforma

```
#include<CL/cl.h>
cl_uint num_platforms;
```

Devuelve la cantidad de plataformas

```
clGetPlatformIDs(NULL,NULL, &num_platforms);
```

```
cl_platform_id platforms[num_platforms];
```

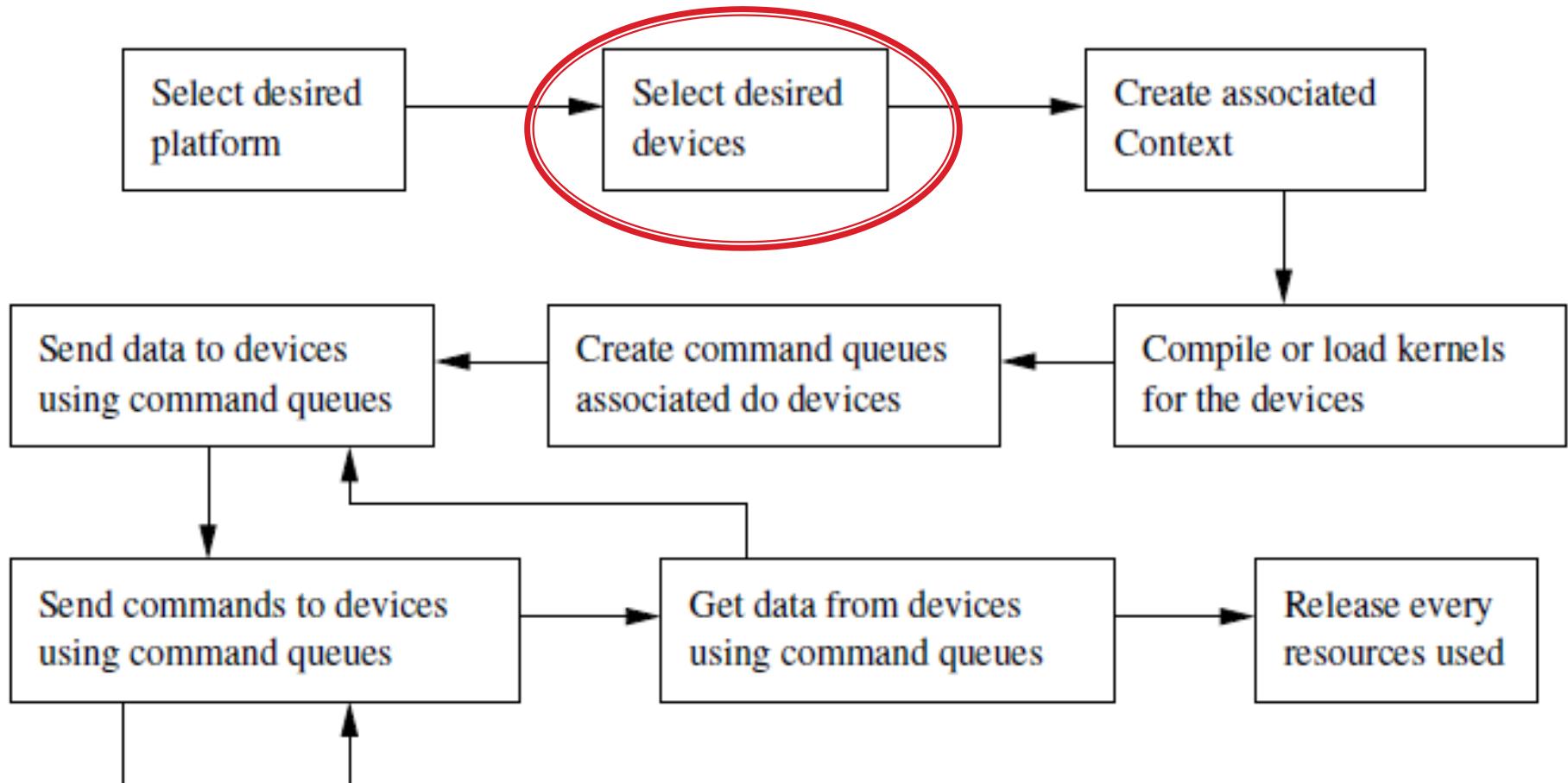
```
clGetPlatformIDs(num_platforms,platforms,NULL);
```

```
for (int i=0; i<=num_platforms; i++) {
```

```
    clGetPlatformInfo(platforms[i], ...);}
```

Llena la lista de las plataformas disponibles

Selección de Dispositivos



Selección de Dispositivos

```
#include<CL/cl.h>
cl_uint num_devices;

clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL,
NULL, NULL, &num_devices);

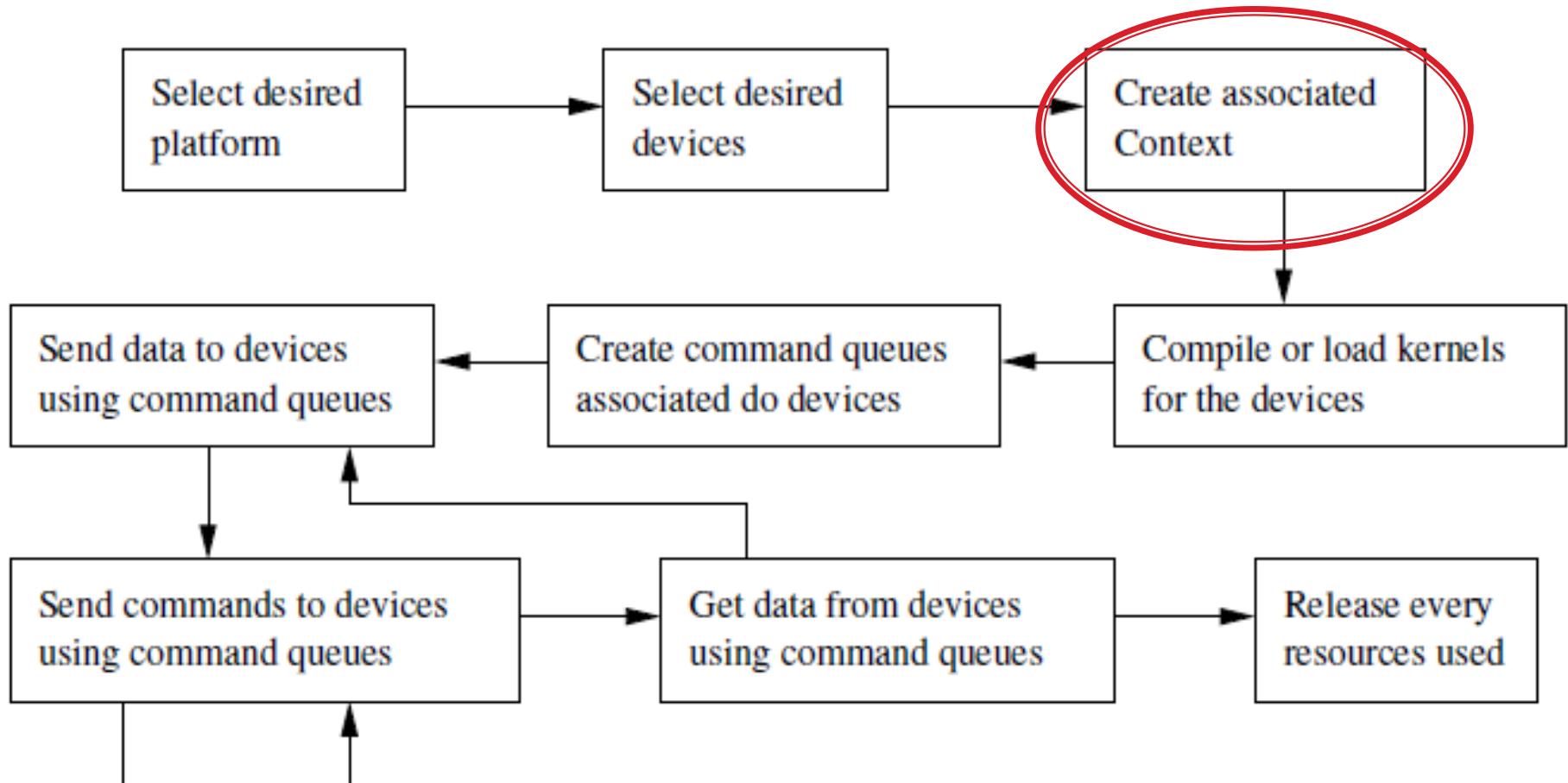
cl_device_id devices[num_devices];

clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL,
num_devices, devices, NULL);
for (int i=0; i<=num_devices; i++) {
    clDeviceInfo(devices[i], ...);}
```

Devuelve la cantidad de devices

Llena la lista de los devices disponibles

Creación de Contexto

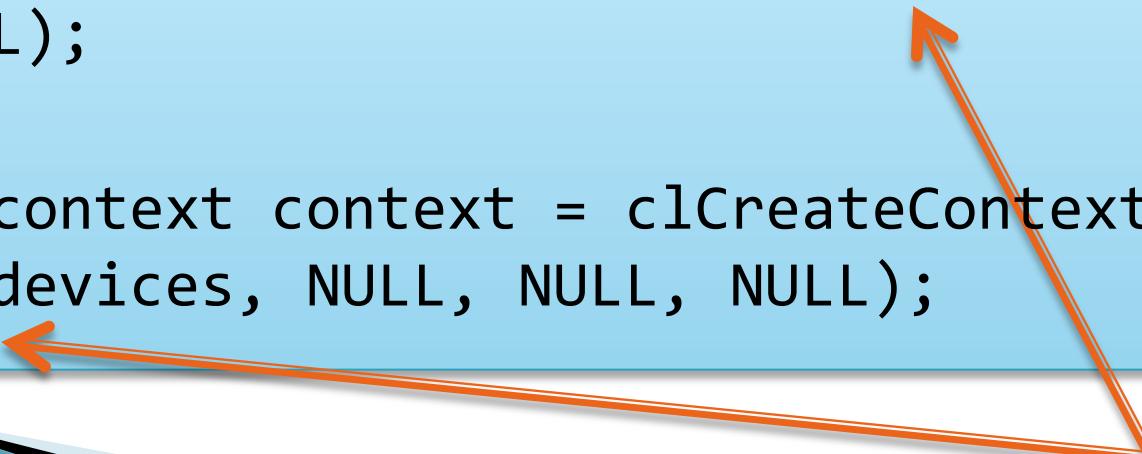


Creación de Contexto

```
#include<CL/cl.h>
cl_context_properties properties[] =
{CL_CONTEXT_PLATFORM,
(cl_context_properties)platform_id,0};

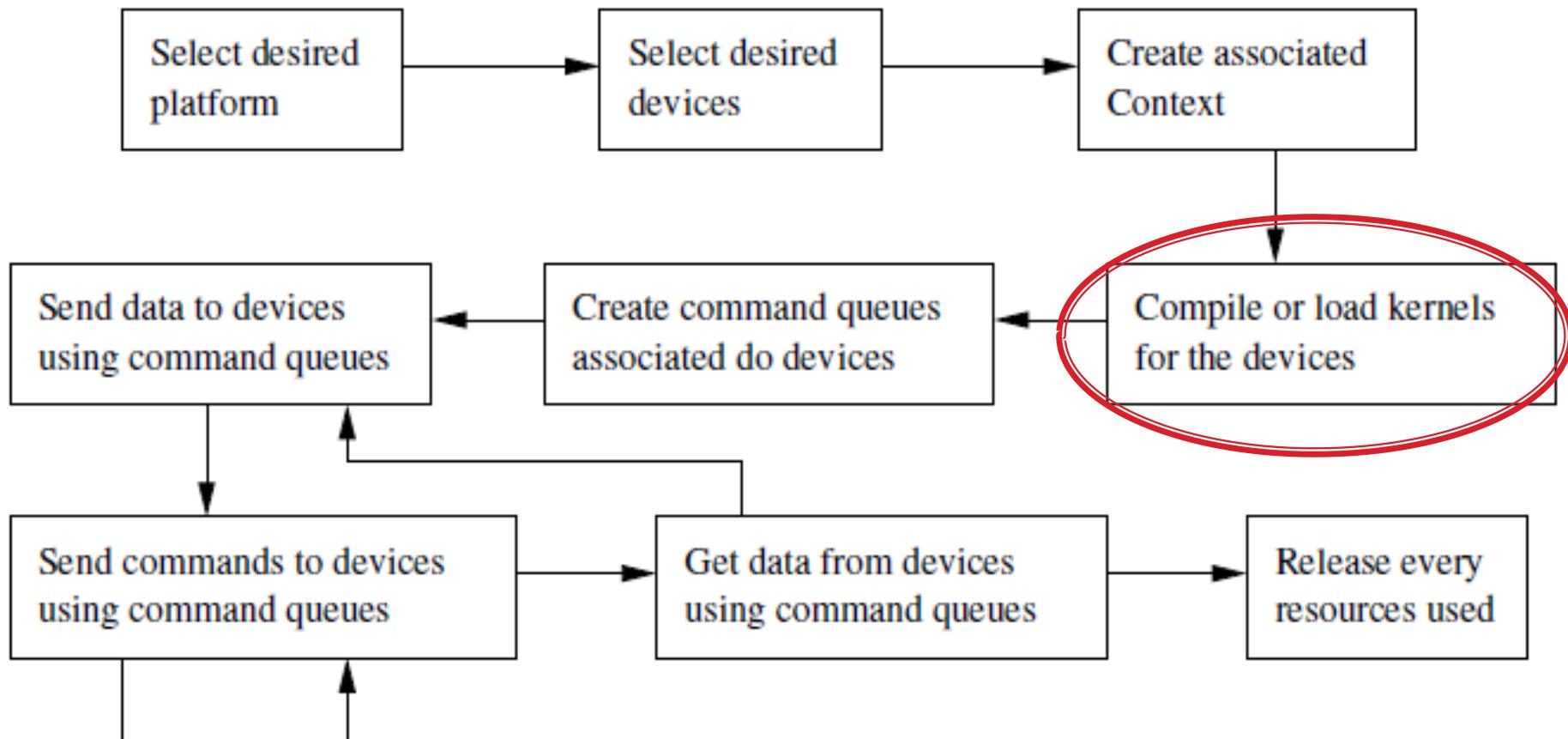
cl_context context = clCreateContextFromType
(properties, CL_DEVICE_TYPE_GPU, NULL, NULL,
NULL);

cl_context context = clCreateContext (properties,
2, devices, NULL, NULL, NULL);
```



Una de estas
dos

Compilación de Kernels



Compilación de Kernels

```
cl_program program;
```

Después vemos como escribir los Kernels

```
program = clCreateProgramWithSource(context,  
string_count, strings, NULL, NULL);
```

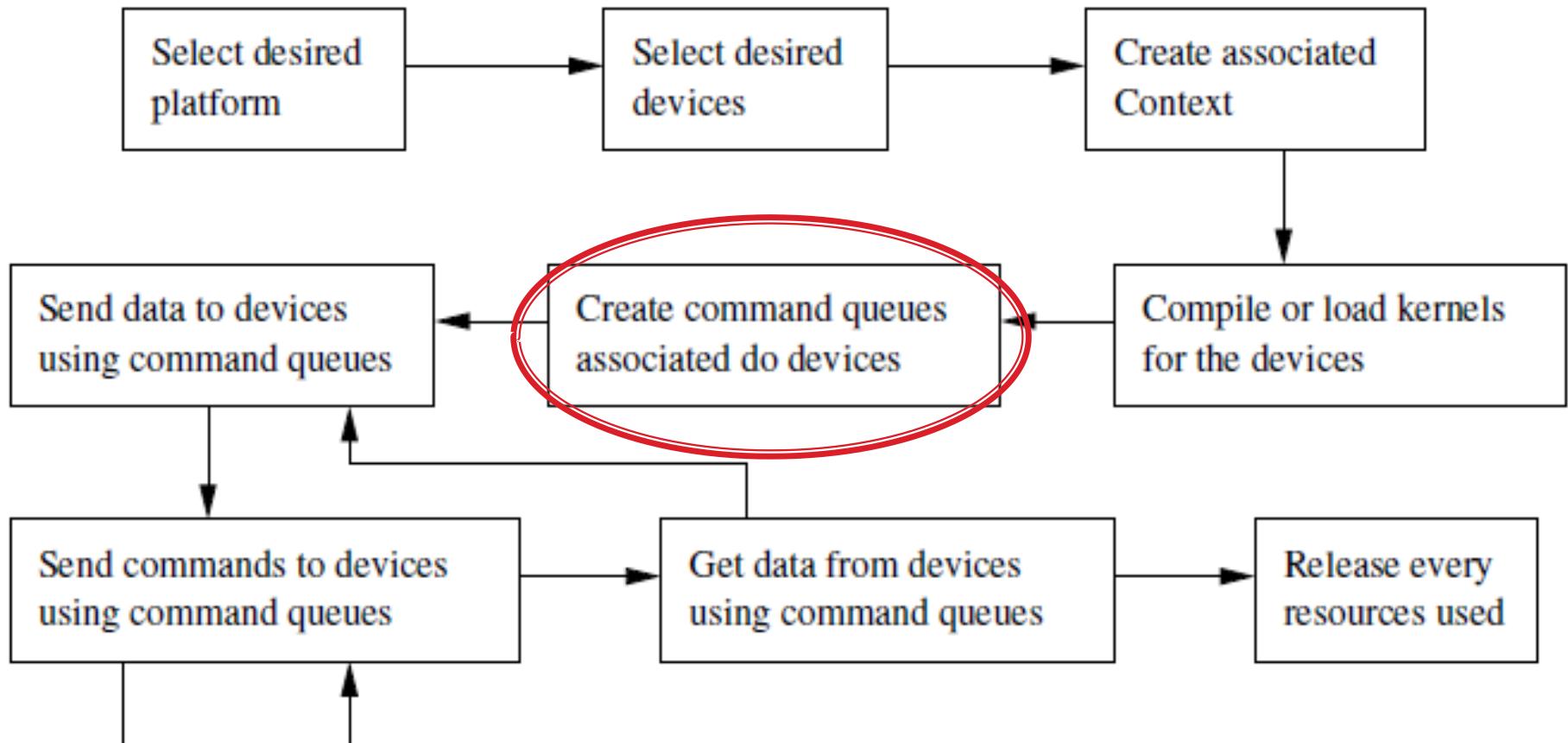
```
clBuildProgram(program, num_devices, device_list,  
options, NULL, NULL);
```

```
cl_kernel kernel = clCreateKernel(program,  
"kernelName", NULL);
```

Extrae kernels puntuales
de la compilación

Si device_list=NULL se
compila para todos los
devices

Creación de Colas de comandos



Creación de Colas de comandos

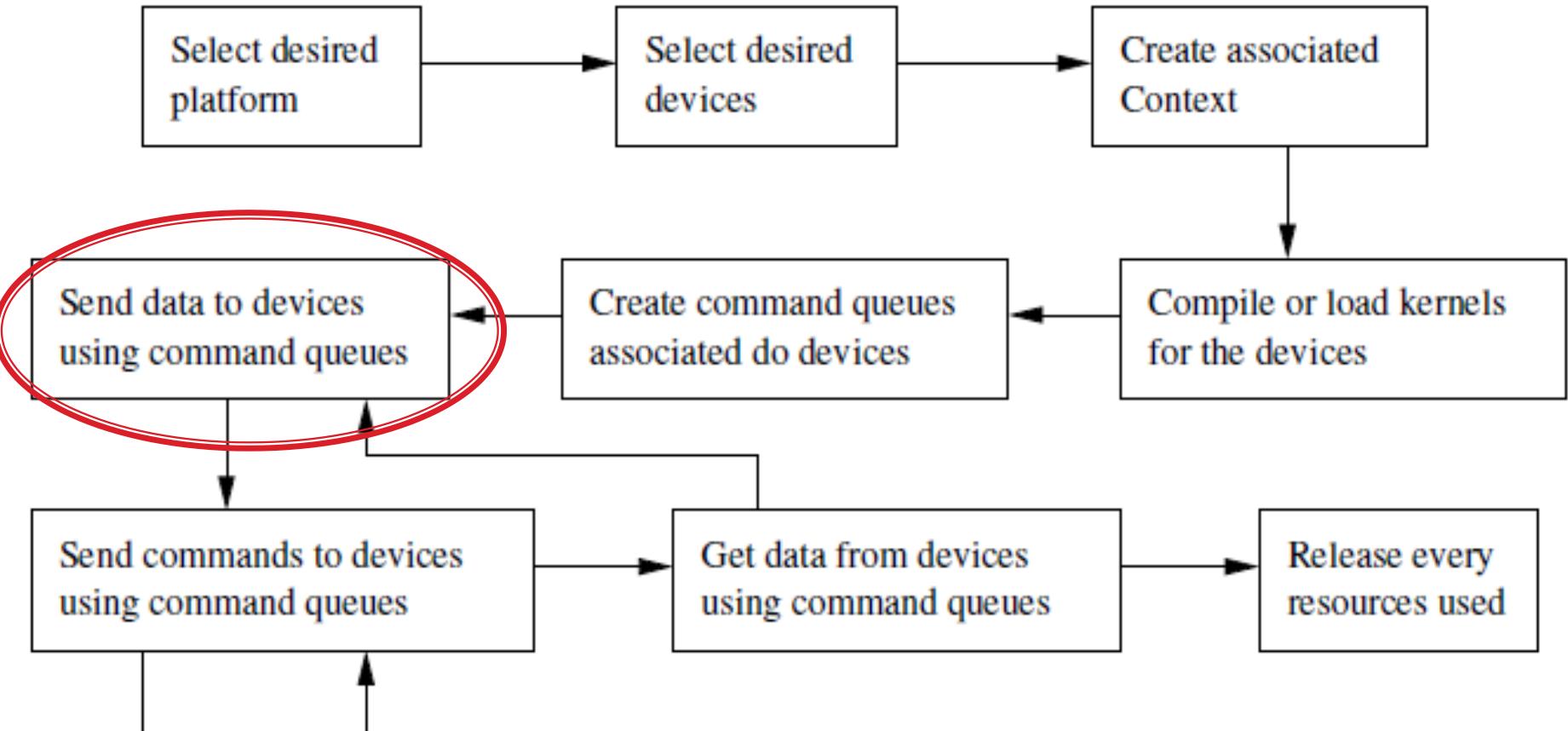
```
cl_command_queue queue;
```

```
queue = clCreateCommandQueue(context,  
devices[chosen_device], 0, NULL);
```

Cada cola de comandos
pertenece a un
dispositivo único dado

Otras opciones como
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE
pueden entrar aquí

Envío de datos al Device



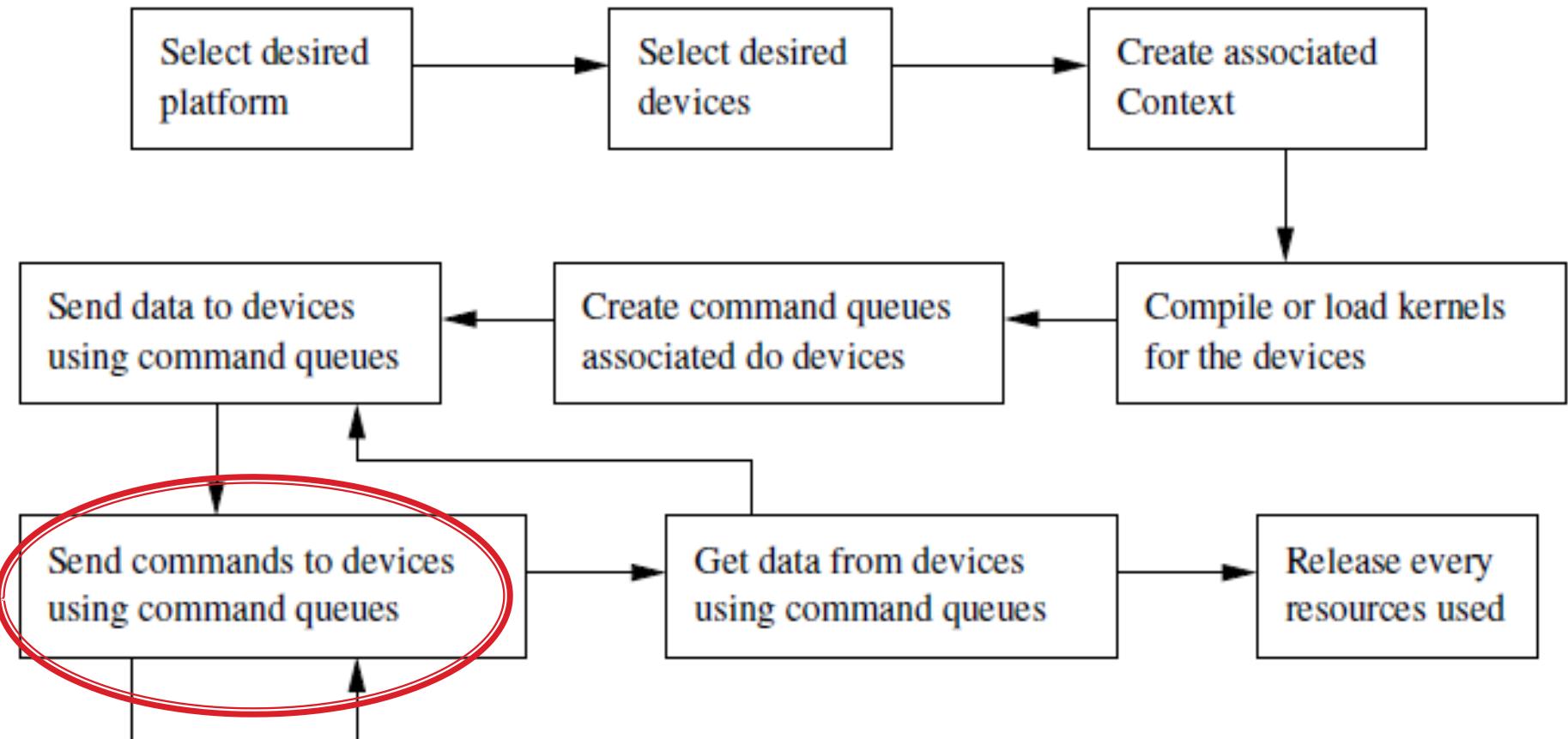
Envío de datos al Device

```
cl_mem write_buffer = clCreateBuffer(context,  
CL_MEM_WRITE_ONLY, buffer_size, NULL, NULL);  
  
clEnqueueWriteBuffer(queue, write_buffer,  
CL_TRUE, 0, buffer_size, data_in, 0, NULL, NULL);
```

Primero creamos Buffers asociados al contexto. No están asociados a un device en particular, lo que permite reutilizarlos en diferentes devices.

Escribimos el arreglo “data_in” en el buffer “write_buffer”

Ejecutamos Kernels

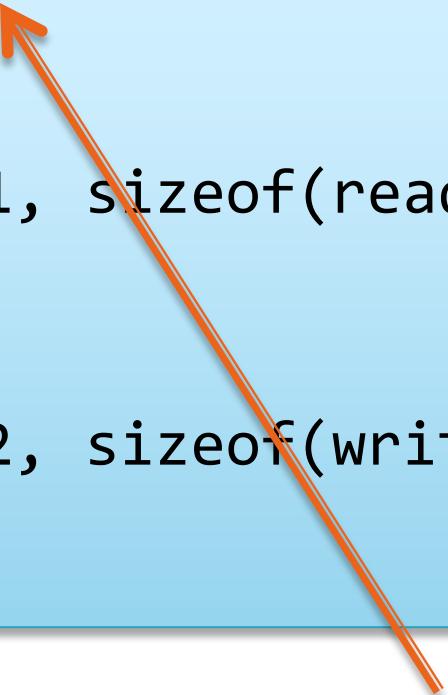


Ejecutamos Kernels

```
clSetKernelArg (kernel, 0, sizeof(data_size),  
(void*)&data_size);
```

```
clSetKernelArg (kernel, 1, sizeof(read_buffer),  
(void*)&read_buffer);
```

```
clSetKernelArg (kernel, 2, sizeof(write_buffer),  
(void*)&write_buffer);
```



Primero asociamos los
parametros del Kernel a buffers.
Este caso son 3 argumentos

Ejecutamos Kernels

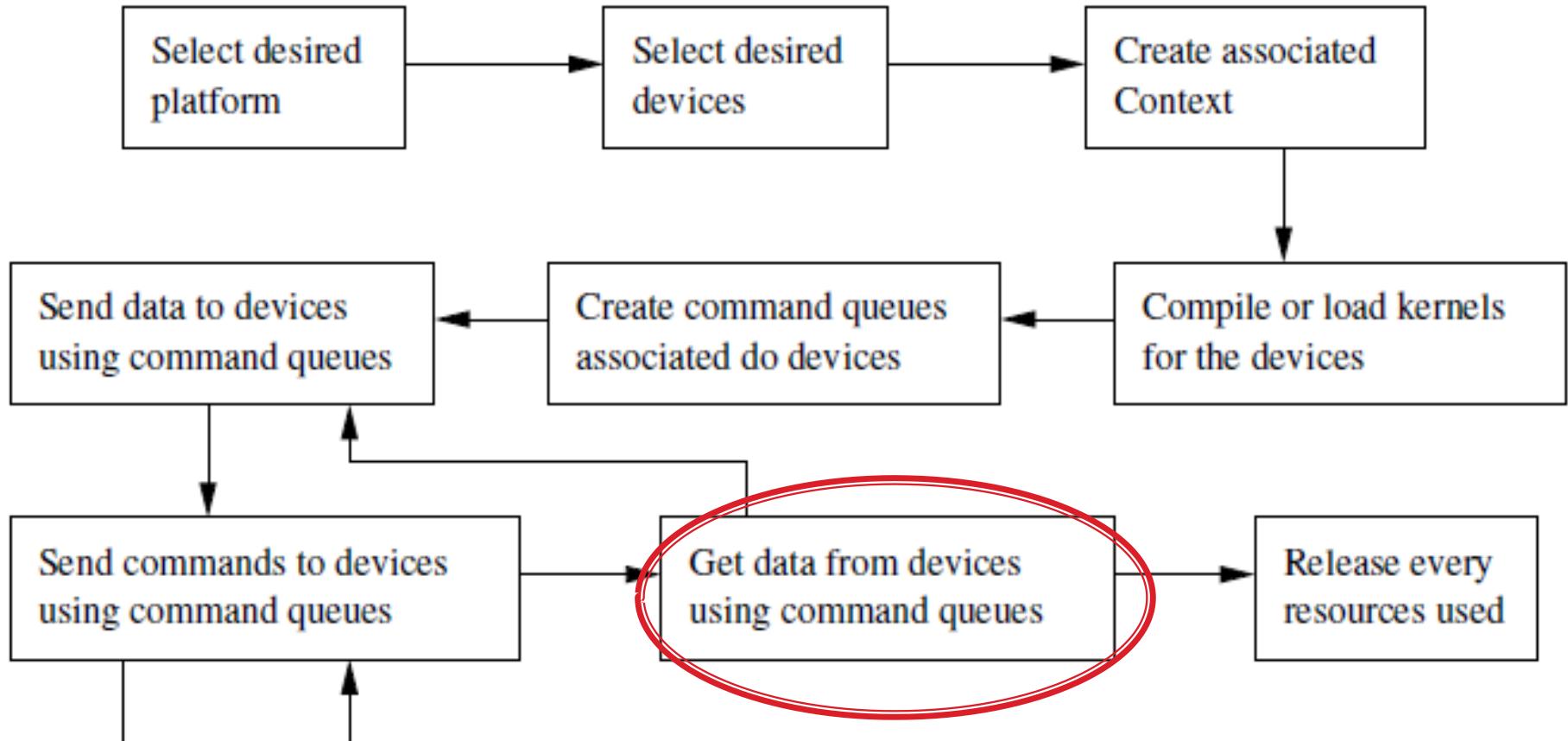
```
size_t localWorkSize[] = {32};  
size_t globalWorkSize[] = {32*10};
```

```
clEnqueueNDRangeKernel(queue, kernel, 1, NULL,  
globalWorkSize, localWorkSize, 0, NULL, NULL);
```

Ponemos el Kernel en cola de ejecución

Luego seteamos los valores del tamaño de WorkGroup y Global

Recuperamos datos del Device



Recuperamos datos del Device

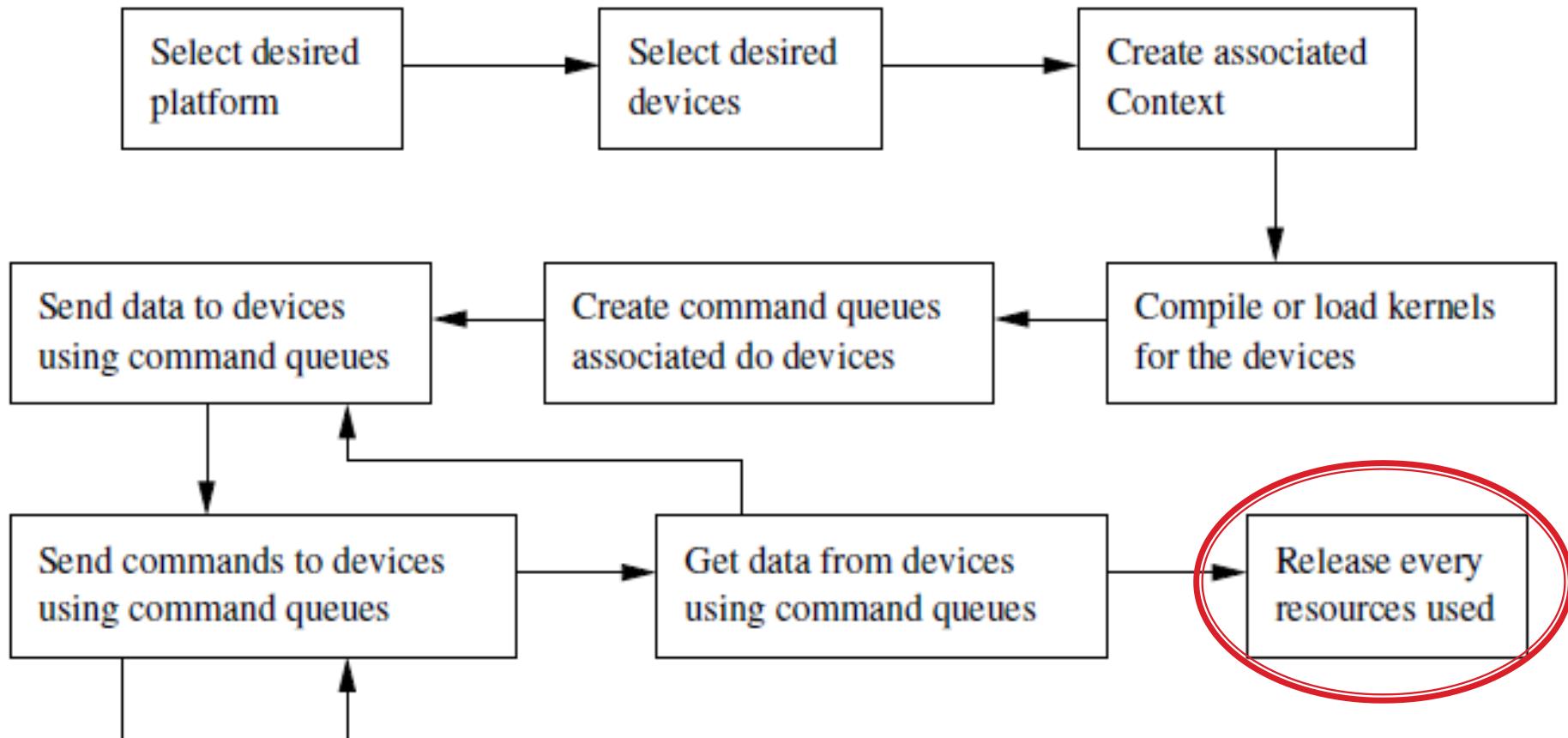
```
cl_mem read_buffer = clCreateBuffer(context,  
CL_MEM_READ_ONLY, buffer_size, NULL, NULL);
```

```
clEnqueueReadBuffer(queue, read_buffer, CL_TRUE,  
0, buffer_size, data_out, 0, NULL, NULL);
```

Primero creamos Buffers asociados al contexto. No están asociados a un device en particular, lo que permite reutilizarlos en diferentes devices.

Leemos en el arreglo “data_out”
en el buffer “read_buffer”

Liberamos recursos ocupados



Liberamos recursos ocupados

- ▶ buffers (`clReleaseMemObject`)
- ▶ events (`clReleaseEvent`)
- ▶ kernel (`clReleasekernel`)
- ▶ programs (`clReleaseProgram`)
- ▶ queues (`clReleaseCommandQueue`)
- ▶ context (`clReleaseContext`)

Sincronización de las colas

```
clEnqueueReadBuffer(queue, read_buffer, CL_TRUE,  
0, buffer_size, data_out, 0, NULL, NULL);
```

```
event_t event_list [] = {event1, event2};  
event_t event;  
clEnqueueReadBuffer(queue, read_buffer, CL_TRUE,  
0, buffer_size, data_out, 2, event list, event);
```

Podemos pasarle una cantidad de eventos a esperar para ejecutar la acción y el evento que genera al terminar la misma

Cuando pasamos ... , 0, **NULL, NULL**) al final de las funciones “Enqueue” se vuelven bloqueantes y no retornan hasta terminar el comando

Lista de Funciones Host

- ▶ Consulta de platform. **clGetPlatformIDs();**
- ▶ Creación de contexto **clCreateContext();**
- ▶ Creación de contexto **clCreateContextFromType();**
- ▶ Creación de colas **clCreateCommandQueue();**
- ▶ Creación de programa **clCreateProgramWithSource();**
- ▶ Compilación de prog. **clBuildProgram();**
- ▶ Creación de kernel **clCreateKernel();**
- ▶ Creación de Buffers **clCreateBuffer();**
- ▶ Encolar Buffers **clEnqueueWriteBuffer();**
- ▶ Encolar Buffers **clEnqueueReadBuffer();**
- ▶ Argumentos de Kernel **clSetKernelArg();**
- ▶ Encolar Kernel **clEnqueueNDRangeKernel();**

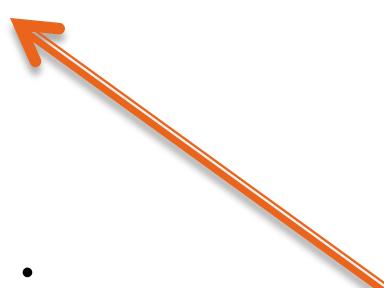
Implementación - Kernel

» Código Kernel OpenCL
OpenCL C Language

OpenCL Kernels

- ▶ Como dijimos, son similares a funciones C.
- ▶ Se ejecutan en el CPU, GPU, o cualquier otro dispositivo.
- ▶ Lenguaje basado en C99

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



Función suma en serie en C

OpenCL Kernels

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

Función suma en serie en C

```
kernel void scalar_add (global const float *a,  
                        global const float *b,  
                        global float *result) {  
    int id = get_global_id(0);  
    result[id] = a[id] + b[id];  
}
```

Función suma en paralelo

OpenCL Kernels

```
kernel void scalar_add (global const float *a,  
                      global const float *b,  
                      global float *result) {  
    int id = get_global_id(0);  
    result[id] = a[id] + b[id];  
}
```

$N = 1$



$\text{get_global_id}(0) = 7$



7	9	13	1	31	3	0	76	33	5	23	11	51	77	60	8
---	---	----	---	----	---	---	----	----	---	----	----	----	----	----	---

+

34	2	0	13	18	22	6	22	47	17	56	41	29	11	9	82
----	---	---	----	----	----	---	----	----	----	----	----	----	----	---	----

=

41	11	13	14	49	25	6	98	80	22	79	52	80	88	69	90
----	----	----	----	----	----	---	----	----	----	----	----	----	----	----	----

Tipos de datos - OpenCL C99

Escalares

- ▶ `bool`
- ▶ `char`
- ▶ `uchar`
- ▶ `short`
- ▶ `ushort`
- ▶ `float`
- ▶ `int`
- ▶ `uint`
- ▶ `long`
- ▶ `ulong`
- ▶ `double`
- ▶ `void`

Vectores

- ▶ `charn`
- ▶ `ucharn`
- ▶ `shortn`
- ▶ `ushortn`
- ▶ `floatn`
- ▶ `intn`
- ▶ `uintn`
- ▶ `longn`
- ▶ `ulongn`
- ▶ `doublen`

Built In Func - OpenCL C99

- ▶ `size_t get_global_id(uint dimindx)`

Retorna el ID global del Kernel

- ▶ `size_t get_local_id(uint dimindx)`

Retorna el ID local del Kernel

- ▶ `size_t get_group_id(uint dimindx)`

Retorna el ID de grupo del Kernel

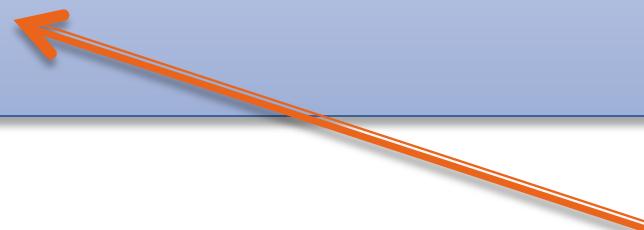
Ejemplos

- » Programa Ejemplo, suma de arreglo escalar

Ejemplo 1 - Suma de Arreglo

▶ Kernel.cl:

```
__kernel void scalar_add (__global const float *a,  
                         __global const float *b,  
                         __global float *result) {  
    int id = get_global_id(0);  
    result[id] = a[id] + b[id];  
}
```



Leido de
Kernel.cl

Ejemplo 1 - Suma de Arreglo

▶ Main.cpp:

```
int main(int argc, char** argv)
{
    // Create a Context on first available platform
    context = CreateContext();
    // Create an CQ on the first available device
    commandQueue = CreateCommandQueue(context,
&device);
```

Funciones
implementadas

Ejemplo 1 - Suma de Arreglo

▶ Main.cpp:

(...)

```
// Create OpenCL program from Kernel.cl
// kernel source
program = CreateProgram(context, device,
"Helloworld.cl");

// Create OpenCL kernel
kernel = clCreateKernel(program, "scalar_add",
NULL);
(...)
```

Funciones
implementadas

Ejemplo 1 - Suma de Arreglo

▶ Main.cpp:

```
(...)  
// Create memory objects that will be used as  
// arguments to kernel. First create host memory  
// arrays that will be used to store the  
// arguments to the kernel  
  
cl_mem memObjects[3] = { 0, 0, 0 };  
float result[ARRAY_SIZE];  
float a[ARRAY_SIZE];  
float b[ARRAY_SIZE];  
(...)
```

Ejemplo 1 - Suma de Arreglo

▶ Main.cpp:

```
(...)  
// Create memory objects that will be used as  
// arguments to kernel. First create host memory  
// arrays that will be used to store the  
// arguments to the kernel
```

```
CreateMemObjects(context, memObjects, a, b)
```

```
(...)
```



Funciones
implementadas

Ejemplo 1 - Suma de Arreglo

▶ Main.cpp:

```
(...)
```

```
// Set the kernel arguments (result, a, b)
```

```
errNum = clSetKernelArg(kernel, 0,  
sizeof(cl_mem), &memObjects[0]);
```

```
errNum |= clSetKernelArg(kernel, 1,  
sizeof(cl_mem), &memObjects[1]);
```

```
errNum |= clSetKernelArg(kernel, 2,  
sizeof(cl_mem), &memObjects[2]);
```

```
(...)
```

Ejemplo 1 - Suma de Arreglo

▶ Main.cpp:

```
(...)
```

```
// Queue the kernel up for execution across the  
array
```

```
size_t globalWorkSize[1] = { ARRAY_SIZE };  
size_t localWorkSize[1] = { 1 };
```

```
errNum = clEnqueueNDRangeKernel(commandQueue,  
kernel, 1, NULL, globalWorkSize, localWorkSize,  
0, NULL, NULL);
```

```
(...)
```

Ejemplo 1 - Suma de Arreglo

▶ Main.cpp:

```
(...)  
// Read the output buffer back to the Host  
  
errNum = clEnqueueReadBuffer(commandQueue,  
memObjects[2], CL_TRUE, 0, ARRAY_SIZE *  
sizeof(float), result, 0, NULL, NULL);  
  
// Print output file  
}
```

Preguntas

?

De donde estudio?

- ▶ De estas filminas, son bastante “autocontenidoas”
- ▶ OpenCL Kronos Specification:
<http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf>
- ▶ OpenCL Implementation & Performance Comparison:
<http://www.argencon.org.ar/sites/default/files/002.pdf>
- ▶ Ejemplos publicados en la página de la materia:
<http://cs.famaf.unc.edu.ar/~dmoisset/adc/>

Como instalo las librerías?

- ▶ GPU Nvidia: [Nvidia Drivers](#)
 - Nvidia GeForce GTS/GT/GTX
- ▶ GPU ATI/AMD: [ATI Catalyst Drivers](#)
 - AMD Radeon/Mobility HD 6800, HD 5x00 series GPU, iGPU HD 6310/HD 6250
- ▶ Intel OpenCL: [Intel OpenCL SDK](#)
 - Soporta 3rd Gen Intel (i3, i5, i7), Intel HD Graphics 4000/2500
- ▶ Lista de productos compatibles:
<http://www.khronos.org/conformance/adopters/conformant-products/>

Ejercicio 1

- ▶ Modificar el **código Host** del Ejemplo 1 (suma de arreglo) para que realice la suma con 3 arreglos de la siguiente manera:

$$A + B = C;$$

luego,

$$C + D = E;$$

Ejercicio 2

- ▶ Modificar el **código Kernel** del Ejemplo 1 (suma de arreglo) para que realice la suma con 3 arreglos de la siguiente manera:

$$A + B + C = E;$$

Ejercicio 3

- ▶ Implementar el Ejercicio 1 con manejo de eventos
- ▶ Recordar que:

```
event_t event_list [] = {event1, event2};  
event_t event;  
clEnqueueReadBuffer(queue, read_buffer, CL_TRUE,  
0, buffer_size, data_out, 2, event list, event);
```



Podemos pasarle una cantidad de eventos a esperar para ejecutar la acción y el evento que genera al terminar la misma

Ejercicio 4

- ▶ Escribir un Kernel que realice la multiplicación de dos matrices
- ▶ Cuanto debe ser N?
- ▶ Consejo: plantee inicialmente matrices cuadradas

$$\mathbf{A}_{m \times n} \times \mathbf{B}_{n \times m} = \mathbf{C}_{m \times n} \text{ with } c_{ij} = \sum_{k=1}^n a_{ik} + b_{kj}$$

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11} \cdot b_{11} + a_{12} \cdot b_{21} & a_{11} \cdot b_{12} + a_{12} \cdot b_{22} \\ a_{21} \cdot b_{11} + a_{22} \cdot b_{21} & a_{21} \cdot b_{12} + a_{22} \cdot b_{22} \end{bmatrix}$$