



UNIVERSIDAD DE
COSTA RICA

Universidad de Costa Rica

CITIC - PRIS Lab



Introducción a la Programación paralela de GPU's con CUDA y OpenCL



Saúl Calderón Ramírez



PATTERN RECOGNITION AND
INTELLIGENT SYSTEMS
RESEARCH LABORATORY

Modelo de Plataforma: Orígenes de OpenCL

Gráficos en 3D: Necesidad de procesamiento en tiempo real

Dispositivos con procesamiento paralelo masivo(más tareas más lentas, que menos más rápidas)

Uso para otras aplicaciones, mediante *SDK's* particulares de cada fabricante (CUDA)

Apple decidió crear una interfaz común para sus dispositivos, tarea heredada por el **grupo Khronos**



Tarjeta gráfica ATI Radeon 7970
con 2048 Stream Processors

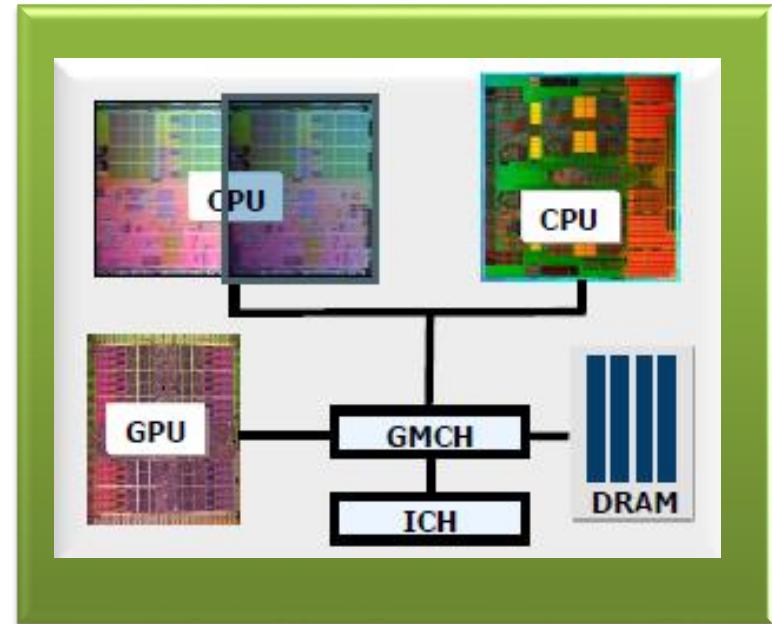
Modelo de plataforma: Computación heterogénea

Nuestro mundo es heterogéneo!

CPU's con múltiples núcleos.

GPU's con múltiples núcleos

Procesadores de propósito específico
(DSP's, FPGAs etc).



GPU: Graphics processor Unit.

GMCH: Graphics memory control hub.

ICH: Input/Output Control hub

Cliff Woolley, NVIDIA Developer Technology Group

Modelo de plataforma: Fundamentos

OpenCL

Open Computing Language : Marco de trabajo de computación paralela en plataformas heterogéneas.

Diseñado para utilizar todos los recursos en los sistemas: CPU's, GPU's, FPGA's, etc, **a la vez**

Abstira detallés específicos de la plataforma, con paralelismo a nivel de **datos e instrucción**

Desarrollado por el Khronos Compute Working Group, con representantes de AMD, Intel y Nvidia

Modelo de plataforma: Fundamentos

OpenCL

Basado en el estándar C99

Funciones básicas, pero además extiende funciones vectoriales

Existen perfiles de escritorio y sistemas móviles

Intel, AMD y Nvidia, Samsung, Altera, entre otras, tienen implementaciones de la interfaz

Introducción a OpenCL: CUDA y OpenCL

Compute Unified Device Architecture: Lenguaje propietario de NVIDIA para programación de GPU's

Esquema de programación similar a OpenCL

OpenCL especifica una interfaz de programación necesitando algunas funciones adicionales

Rendimiento de CUDA y OpenCL pueden llegar a ser iguales teóricamente

Modelo de plataforma OpenCL

CUDA	OpenCL
Host	Host
-	Context (collection of devices)
Device	Device
Streaming Module (SM)	Compute Unit
Streaming Processor (SP)	Processing Element (PE)
Global Memory	Global Memory
Constant Memory	Constant Memory
Shared Memory	Local Memory
Register and Local Memory	Private Memory
Host Program	Host Program
Kernel (Device Program)	Kernel (Device Program)
Grid	NDRange (index space)
Block	Work Group
Thred	Work Item
threadIdx.x, threadIdx.y, threadIdx.z	get_local_id(0), get_local_id(1), get_local_id(2)
blockIdx.x * blockDim.x + threadIdx.x	get_global_id(0)
threadIdx.x	get_local_id(0)
blockIdx.x * blockDim.x	get_global_size(0)
blockDim.x	get_local_size(0)

Equivalencias entre CUDA y OpenCL

Cliff Woolley, NVIDIA Developer Technology Group

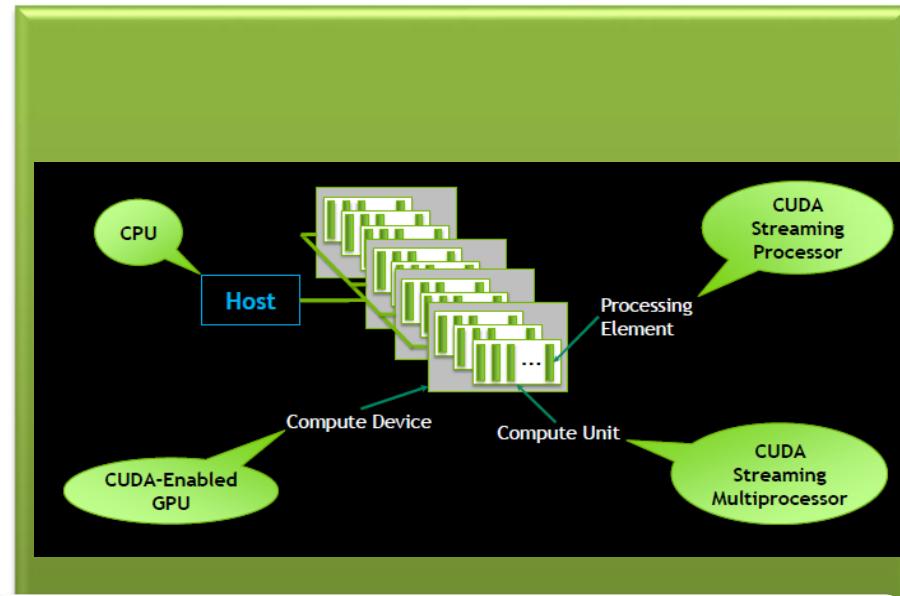
Modelo de plataforma OpenCL

Host: Usa memoria principal

Compute Device: Ejecuta código paralelizado, soporta OpenCL

Compute Unit: Unidad de hardware con varios núcleos

Processing element: Núcleo de procesamiento



Modelo de plataforma de OpenCL en Arq. CUDA
Cliff Woolley, NVIDIA Developer Technology Group

Modelo de ejecución OpenCL



El código del *Host* puede escribirse en C/C++



El código del *Device* debe estar en C y se define en una función *Kernel* ejecutada en cada “hilo”



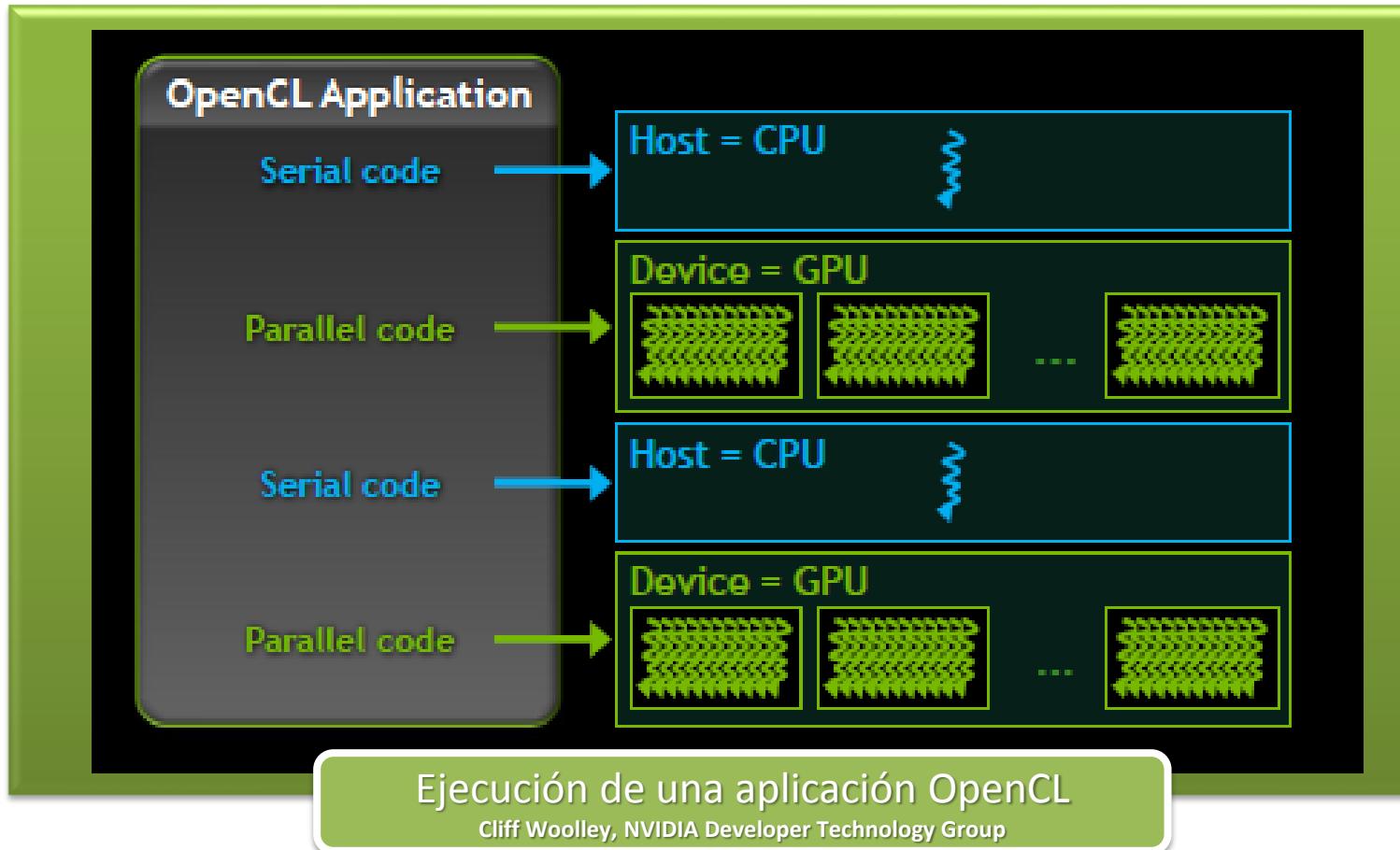
El código del *Host* especifica los comandos de transferencia de datos entre el *Host* y el *Device*



Además especifica cuándo y donde se ejecuta el código del *Device*

Modelo de ejecución OpenCL

Diagrama de ejecución de un programa en Open CL



Modelo de ejecución OpenCL



Para cada algoritmo se debe definir un dominio de computación N -dimensional



Se descompone cada tarea en *work-items*



Work-group: grupo de *work-items*, con acceso a una memoria compartida, pueden realizar operaciones de sincronización

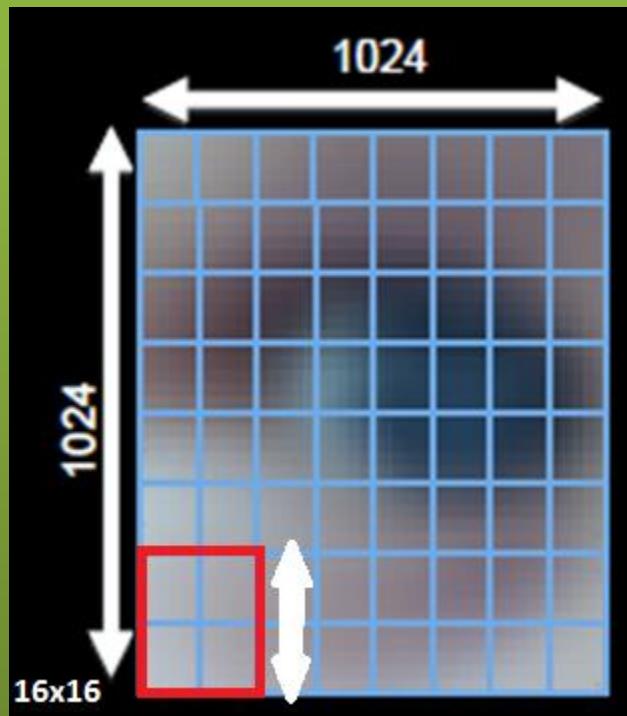


El código de cada *work-item* se define en el *kernel*, el cual se ejecuta en el *Compute Device*

Modelo de ejecución OpenCL



Ejemplo: Workgroup de 16×16 en problema de dimensión 2 con 1024×1024 elementos

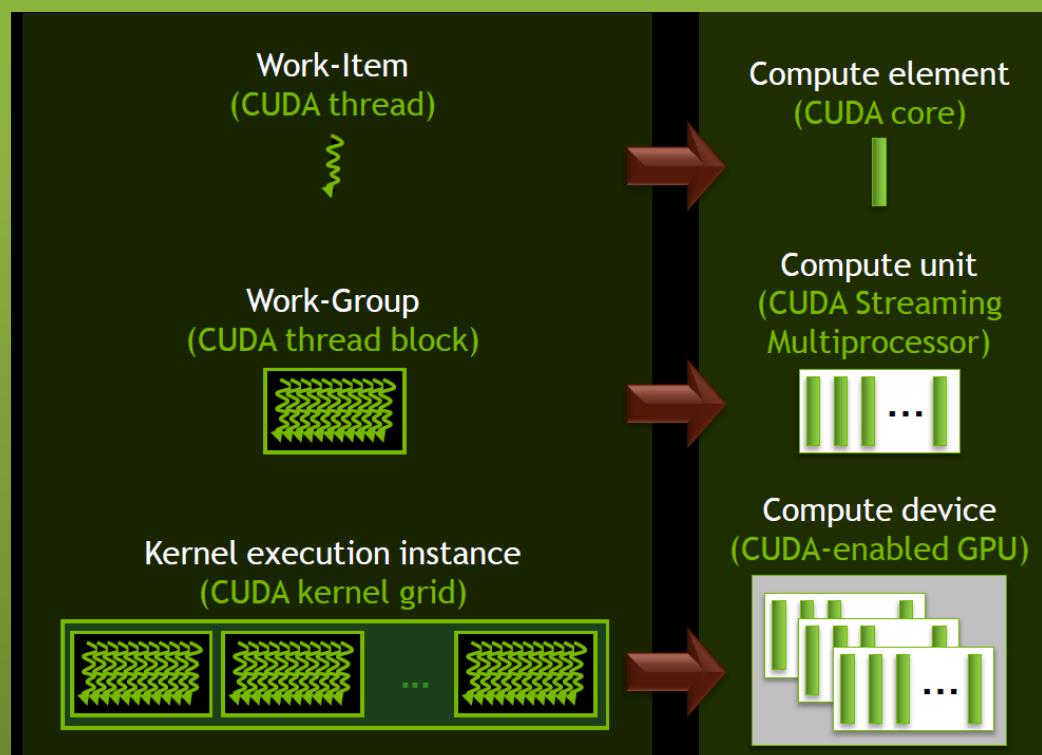


Problema de dimensión 2

Cliff Woolley, NVIDIA Developer Technology Group

Modelo de ejecución OpenCL

Un *work-item* por *processing element* y un *work-group* por *compute unit*



Mapeo del modelo de ejecución con la infraestructura

Cliff Woolley, NVIDIA Developer Technology Group

Modelo de ejecución OpenCL



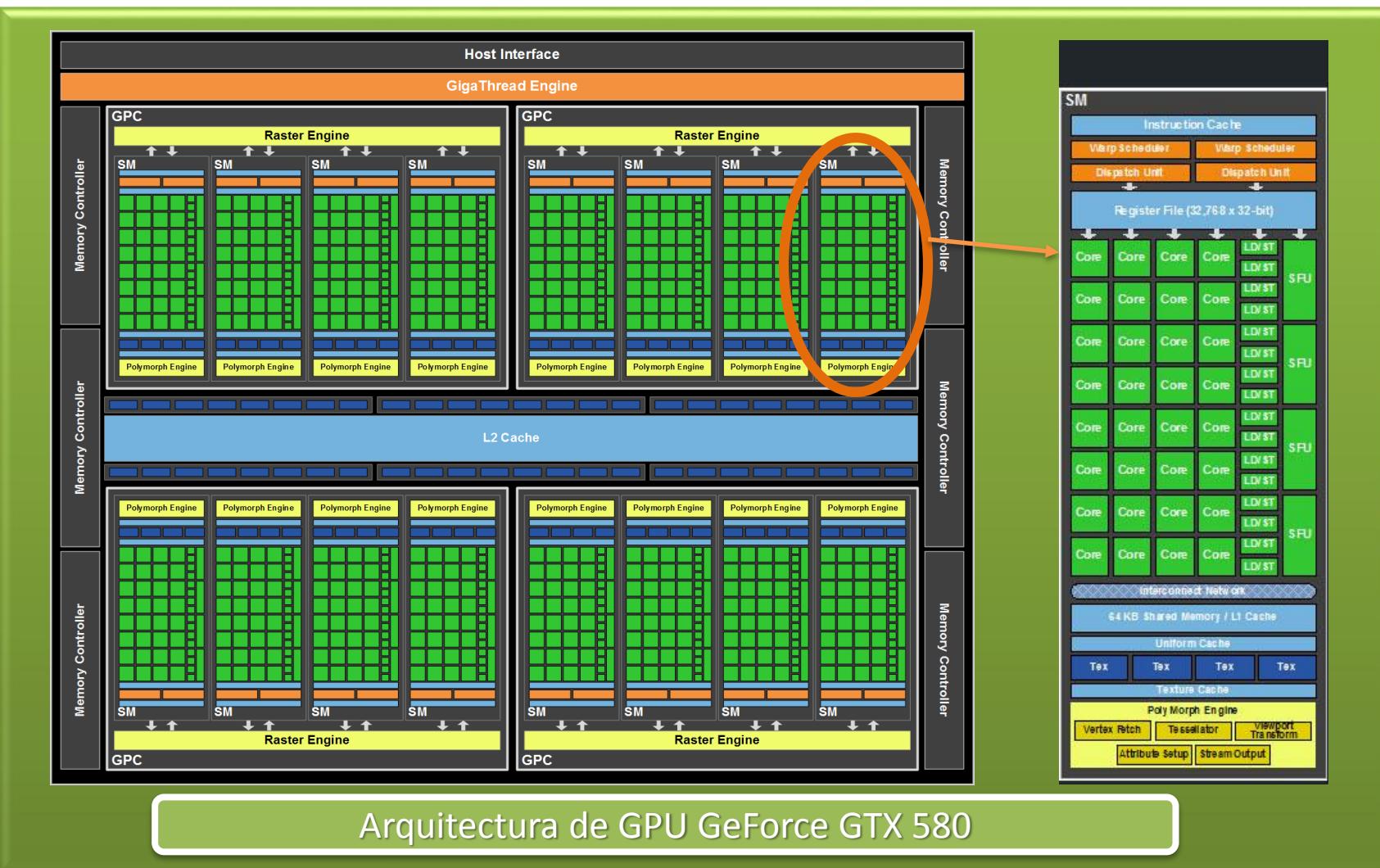
Ejemplo: Función *Kernel* ejecutada en el dispositivo:

```
__kernel void vector_add(__global const int *A, __global const
int *B, __global int *C) {
    // Get the index of the current element to be processed
    int i = get_global_id(0);

    // Do the operation
    C[i] = A[i] + B[i];
}
```

Correspondencia Modelo de ejecución-hardware

Arquitectura de una Geforce GTX 580: 32 CUDA cores per Streaming Multiprocessor (16)



Modelo de ejecución en OpenCL



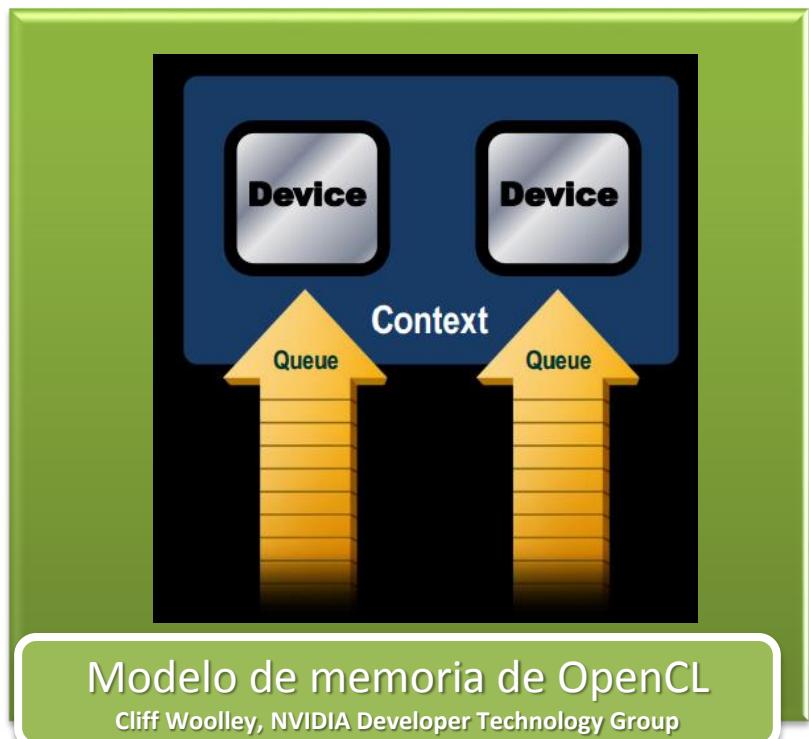
OpenCL usa un concepto abstracto de *Device* el cual permite crear un entorno heterogéneo de ejecución



Contexto: Ambiente en el que se ejecutarán los *work-units*, definido por los *Devices* (estados de su memoria y sus propiedades)



Cola de comandos: Estructura que almacena todos los comandos enviados por el *Host* a un *Compute Device*, ej: copia de datos de *Host* a *Device*, ejecución del kernel



Modelo de memoria de OpenCL

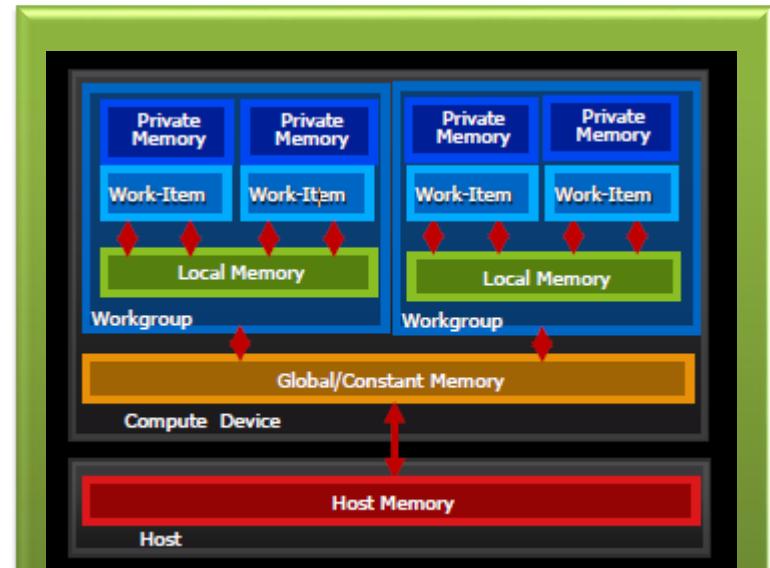
Memoria privada: Para un *work-item*, en caché

Memoria local: Compartida en un *work-group*, en caché, más rápida

Memoria global/constante: Visible para todos los *work-groups*, más lenta

Memoria del Host: En el CPU

Se debe mover la memoria del host ->global del dispositivo->local y viceversa, **no soporta** memoria dinámica dentro del dispositivo



Modelo de memoria de OpenCL

Cliff Woolley, NVIDIA Developer Technology Group

Programación en OpenCL



Capa de plataforma: Consulta de la plataforma y creación del contexto (no existe en CUDA)



Compilador C99: Compila el código a ejecutar en el dispositivo



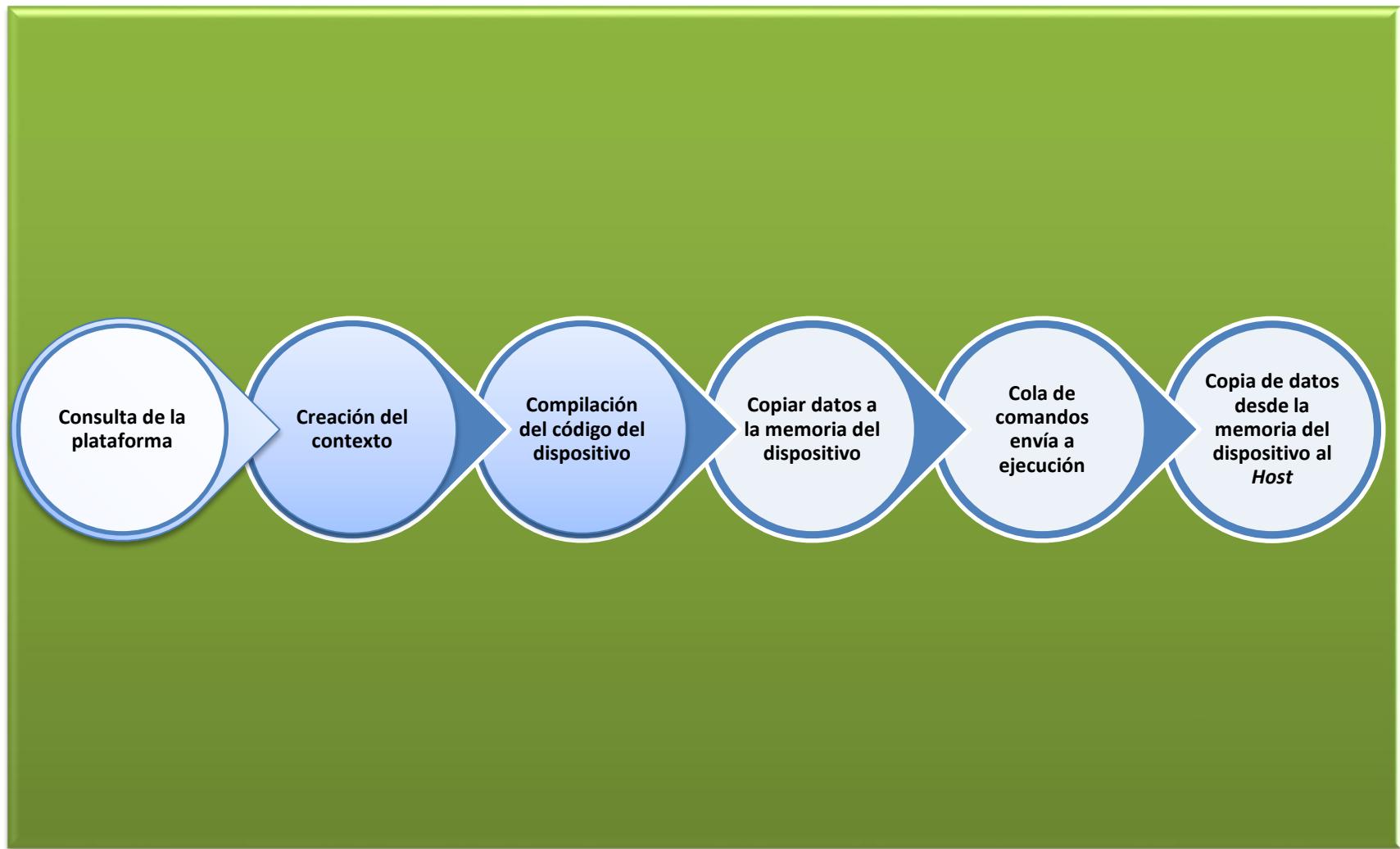
Manejo de ejecución: Manejo de la cola de commandos y comunicación de memorias *Host-Device*



Existen *bindings* en java, C++, Phyton, etc

Programación en OpenCL: Pasos básicos

Pasos básicos de una aplicación OpenCL



Programación en OpenCL: Capa de plataforma(C++)

 Creación de la plataforma: Para averiguar las plataformas disponibles:

```
static cl_int Platform ::get(VECTOR_CLASS<Platform> * platforms)
```

 Obtención de los dispositivos de la plataforma: Dispositivos disponibles en una plataforma específica:

```
cl_int Platform::getDevices(cl_device_type type, VECTOR_CLASS<Device> *devices)
```

cl_device_type:

CL_DEVICE_TYPE_GPU (GPU's en SLI)

CL_DEVICE_TYPE_CPU

CL_DEVICE_TYPE_ACCELERATOR (Aceleradores de física AGEIA, DSP's)

CL_DEVICE_TYPE_ALL

CL_DEVICE_TYPE_DEFAULT

Programación en OpenCL: Capa de plataforma(C++)



Creación del contexto: A partir de una lista de dispositivos:

```
Context::Context(VECTOR_CLASS<Device>& devices)
```



Manejo de errores: Retornan un código de error, una excepción (C++ wrapper) y también se puede agregar una función en el *callback*

Programación en OpenCL: Compilador C99



Creación estructura archivo : Se lee el archivo *.cl* con el código del *Kernel*:

```
std::ifstream sourceFile("vector_add_kernel.cl")
```



Creación de la hilera con el contenido del archivo: Lectura con un iterador por todo el archivo(ultimo parámetro indica el final del iterador):

```
string  
sourceString(istreambuf_iterator<char>(sourceFile),(istreambuf_iterator<char>()))
```

Programación en OpenCL: Compilador C99



Creación de la estructura del código fuente: A partir de hilera

```
Program::Sources source(1, std::make_pair(sourceString.c_str(),  
sourceString.length() + 1))
```



Creación de la instancia del programa:

```
cl::Program::Program(const Context& context, const Sources& sources,  
cl_int * err = NULL)
```

Programación en OpenCL: Compilador C99 y Manejo de ejecución



Compilación del código del *kernel*: La función *build*, crea el código máquina del programa:

```
program.build();
```



Creación del *kernel*: A partir del programa compilado y el nombre de la rutina, se crea el *kernel*:

```
Kernel kernel(Program program, String routineName);
```

Programación en OpenCL: Compilador C99 y Manejo de ejecución



Creación de la cola de comandos: Instancia de la clase *Command Queue* encargada de manejar comandos (copia de datos al dispositivo, ejecución de kernel):

```
CommandQueue queue = CommandQueue(Context context, Device device)
```



Creación de espacio para el buffer en el dispositivo:

```
Buffer bufDeviceA = Buffer(Context context, cl_mem_flags type, int bytesSize);
```

cl_mem_flags:

CL_MEM_READ_WRITE (Escritura y lectura)

CL_MEM_WRITE_ONLY (Solo escritura)

CL_MEM_READ_ONLY (Solo lectura)

Programación en OpenCL: Manejo de ejecución



Escritura de datos en el dispositivo: Copia los datos al dispositivo:

```
queue.enqueueWriteBuffer(bufDeviceA, cl_blocking blocking, size_t offset, int  
bytesSize, const void * ptrHostA);
```

cl_bool:

CL_TRUE (Escritura con bloqueo)
CL_FALSE (Escritura sin bloqueo)



Asignación de parámetros al *kernel*: Se enlistan los parámetros a recibir por el kernel:

```
kernel.setArg(int parameterNumber, Buffer bufferOnDevice);
```

Programación en OpenCL: Manejo de ejecución



Definición de la cantidad de hilos globales y locales y ejecución:

```
queue.enqueueNDRangeKernel(Kernel kernel, NDRANGE nullRange, NDRANGE  
globalNumThreads, NDRANGE localNumThreads);
```

nullRange: En la revisión actual (1.2) se envía el valor NullRange.

globalNumThreads / **localNumThreads** = Cantidad de *workgroups*



Lectura del resultado desde el dispositivo:

```
queue.enqueueReadBuffer(Buffer deviceBuffer cl_blocking blocking, size_t offset,  
int bufferByteSize, int *hostBuffer)
```

Programación en OpenCL: Manejo de ejecución



Código del *kernel*:

```
__kernel void vector_add(__global const int *A, __global const int *B, __global int *C) {  
  
    // Get the index of the current element to be processed  
    int i = get_global_id(0);  
  
    // Do the operation  
    C[i] = A[i] + B[i];  
}
```



Existen programas para revisar sintaxis de un *kernel* como el *Intel Kernel Builder*

Programación en OpenCL: Manejo de ejecución

Código del *kernel*: Procesamiento de matrices

```
__kernel void vector_add(__global const int *ptrDevA, __global const int *ptrDevB,
__global int *ptrDevRes, int columns) {
    // Get the index of the current element to be processed
    int curRow = get_global_id(0);
    int curCol = get_global_id(1);
    int elementA;
    int elementB;
    int value;
    elementA = ptrDevA[curRow * columns + curCol];
    elementB = ptrDevB[curRow * columns + curCol];
    value = elementA + elementB*320 + 564*elementA + elementB % 5;
    ptrDevRes[curRow * columns + curCol] = value;
}
```

Programación en OpenCL: Consejos

GPU's optimizados para datos numéricos tipo *float*

Utilizar grupos de trabajo con varios hilos en GPU's (un grupo por *Compute Unit*)

Evitar usar la memoria global, hacer *cacheo* a memoria local es necesario

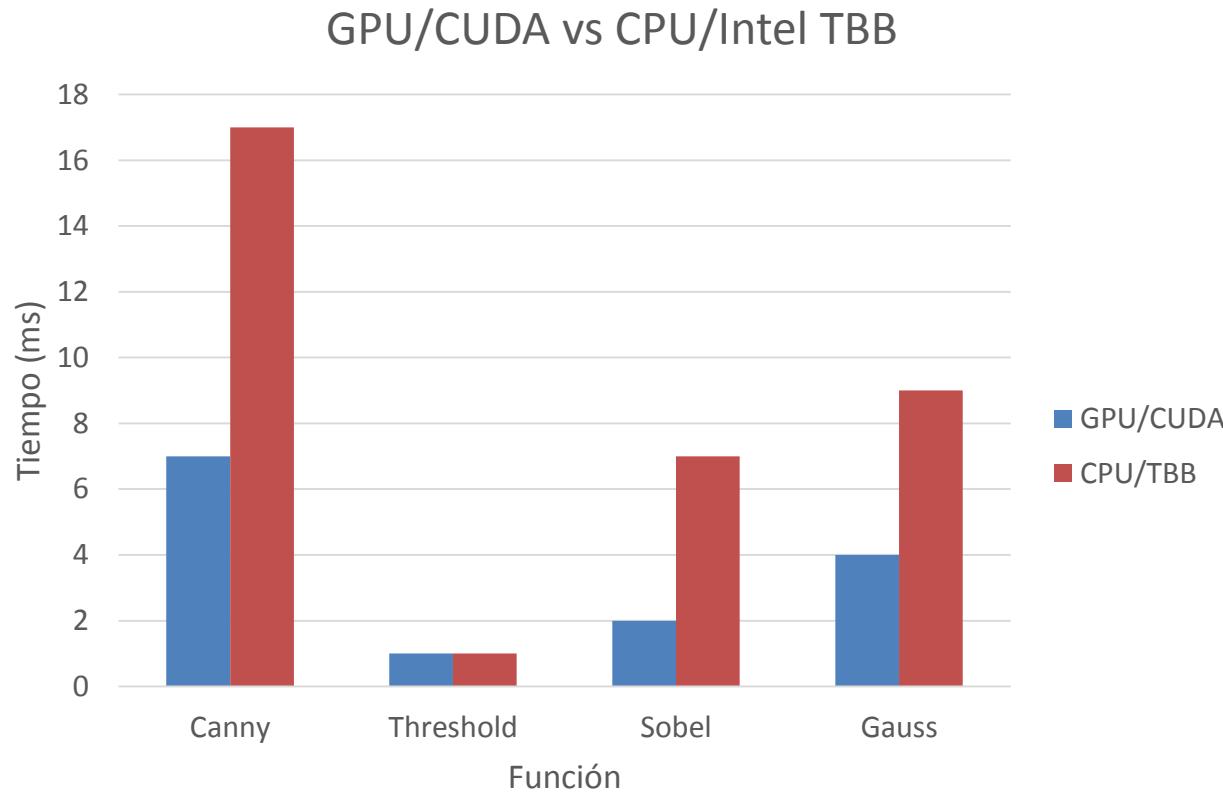
Las funciones a usar dentro del *kernel* deben estar implementadas explícitamente en OpenCL, existen librerías como la FFTW que proveen funcionalidades más complejas

Programación en OpenCL: Consejos

- La memoria dinámica dentro del *kernel* no es soportada, se debe asignar antes de su ejecución (arreglos de punteros no son posibles)
- “*Pensemos en masivo*”, *Kernels* con tareas sencillas pero ejecutándose con otros 10 mil en paralelo
- Diseñar e implementar el *kernel* en C primero, y asegurar su funcionamiento correcto
- Existen librerías con implementaciones en CUDA y OpenCL, como OpenCV

Programación en OpenCL: Evaluación

Ejemplo de aceleración: CPU vs GPU, OpenCV en arquitectura Fermi (Geforce GTX 460, 336 CUDA Cores)

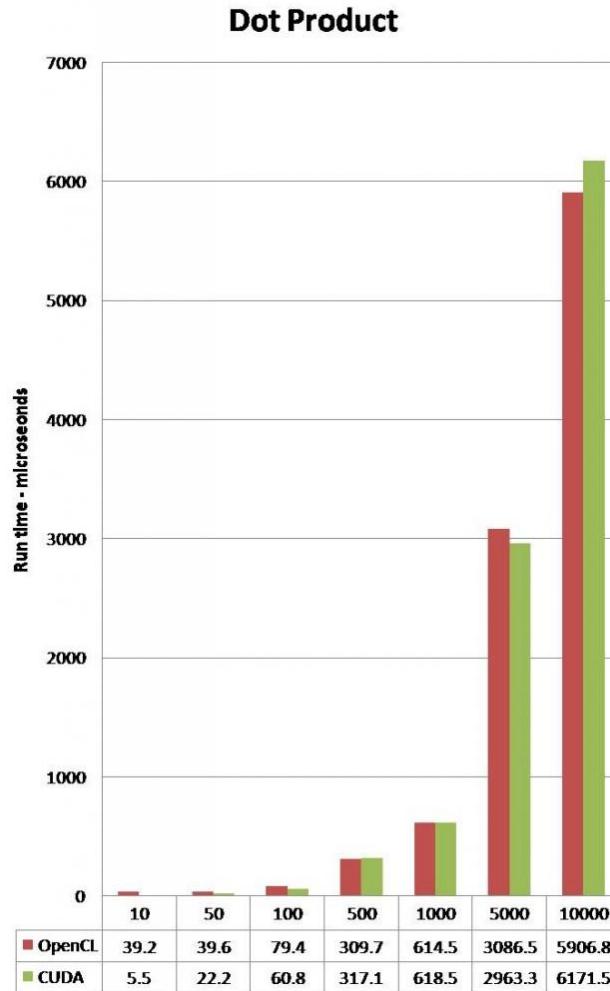


Programación en OpenCL: Evaluación

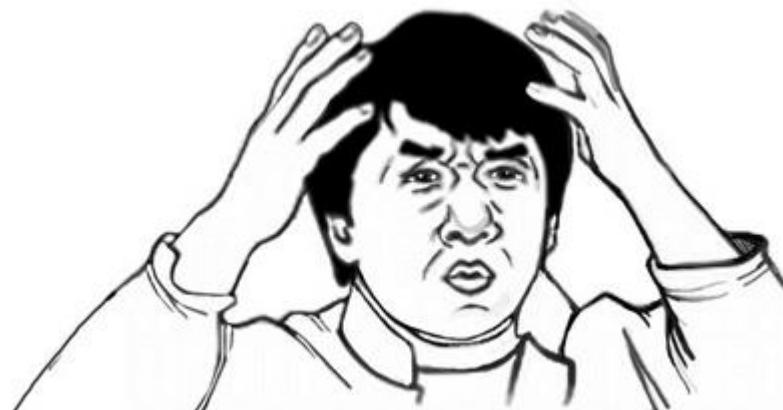
CUDA vs OpenCL: Sobrecarga de OpenCL para pocos datos le da leve desventaja en promedio, pero para muchos datos, es igual o más efectivo que CUDA

Pruebas en una Tesla C2050

tomado de
<http://blog.accelereyes.com/blog/2010/05/10/nvidia-fermi-cuda-and-opencl/>



Ahora... ...olviden todo lo que vimos... por que
con lo que viene la programación heterogénea
será más sencilla!



Tomado de Hola soy
German, demasiado vacilón

Programación en OpenCL: Tendencias



OpenACC: Facilitar programación heterogénea



Definición de directivas de preprocesador para paralelizar bucles (*for*, *while*), al estilo *OpenMP*



Código paralelo de bucles generado antes de compilación con: `#pragma acc kernels`



NVIDIA ha desenfocado recursos de *OpenCL* y los ha asignado a *OpenACC*

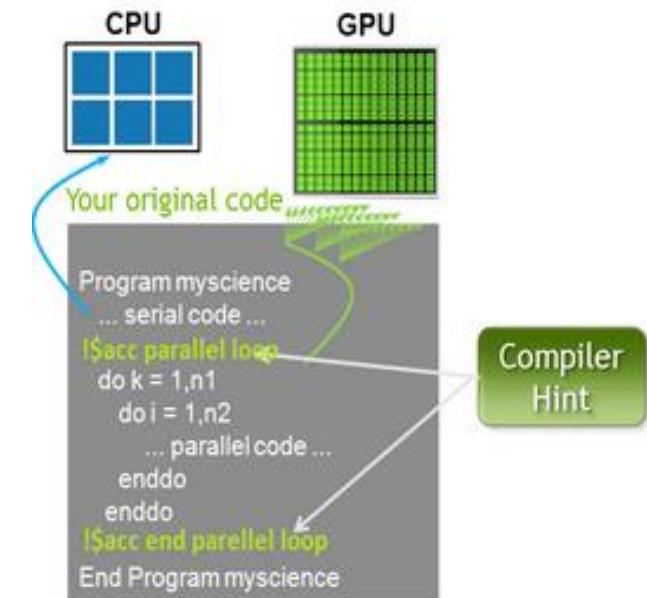
Programación en OpenCL: Tendencias y Discusión

A Silver bullet or a trail of broken promises? Dr. Michael O'Boyle
Institute for Computing Systems Architecture, University of Edinburgh, UK

Programadores elitistas: Las herramientas de programación paralela (PP) automáticas han fallado

Problema: Subutilización de recursos, menos investigadores y aplicaciones pueden acceder a las ventajas de la PP

Oportunidades: Utilizar técnicas de *aprendizaje automático extensivamente* para desarrollar compiladores más inteligentes



OpenACC®
DIRECTIVES FOR ACCELERATORS

Bibliografía y materiales adicionales

Benedict R. Gaster, L. H. (2012). *Heterogeneous Computing with OpenCL*. Morgan Kauffman

Introduction to OpenCL, Cliff Woolley, NVIDIA Technology Group

Repositorio git: https://github.com/saul-calderonramirez/taller_ocl



Obrigado!

Danke!

Grazie!

Gracias!

Thanks!

merci beaucoup!