

IMT2112 - Algoritmos Paralelos en Computación Científica

Programar con OpenCL

Elwin van 't Wout

28 de noviembre de 2019



PONTIFICIA
UNIVERSIDAD
CATÓLICA
DE CHILE

Facultad de Matemáticas • Escuela de Ingeniería

imc.uc.cl

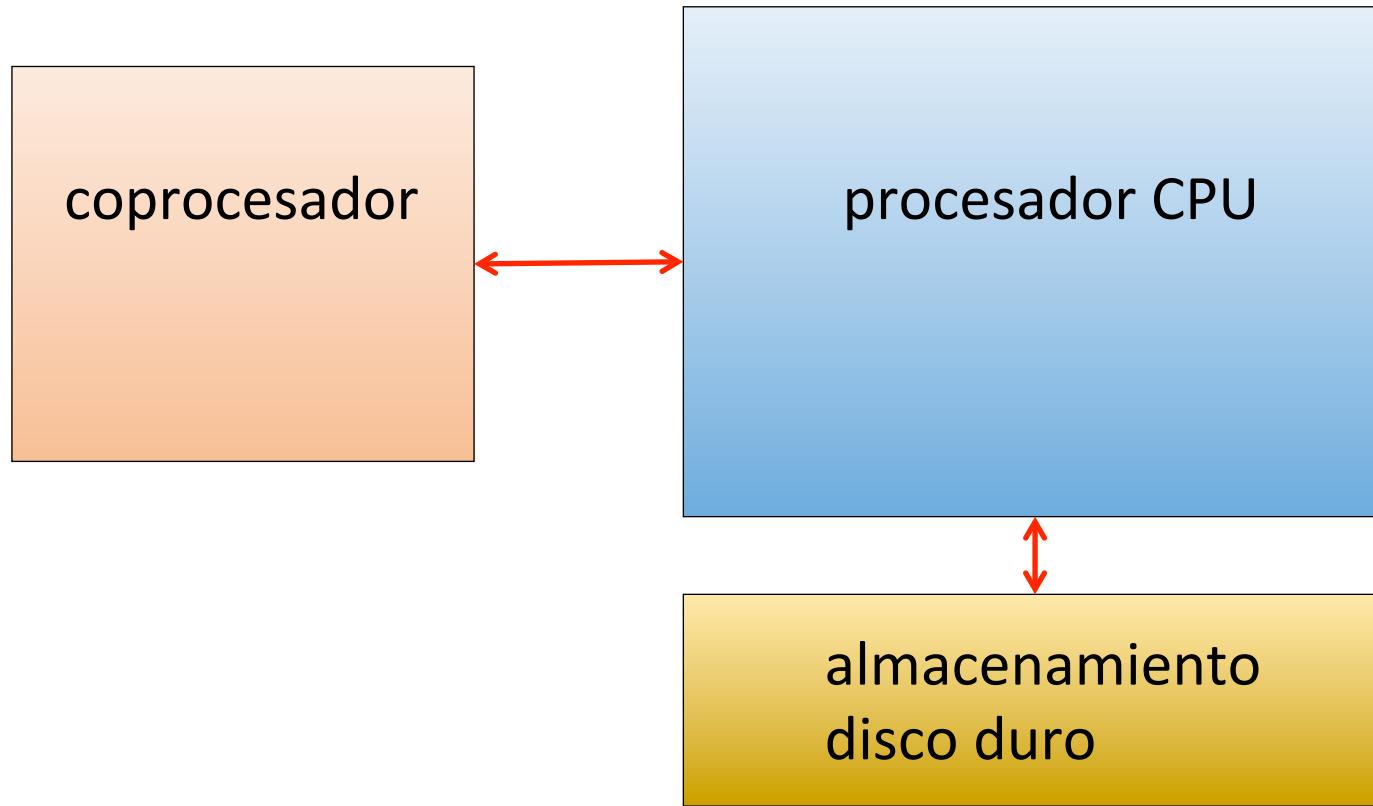
Clase previa

- Sistemas con tarjetas gráficas

Agenda

- ¿Como programar con OpenCL en tarjetas gráficas?

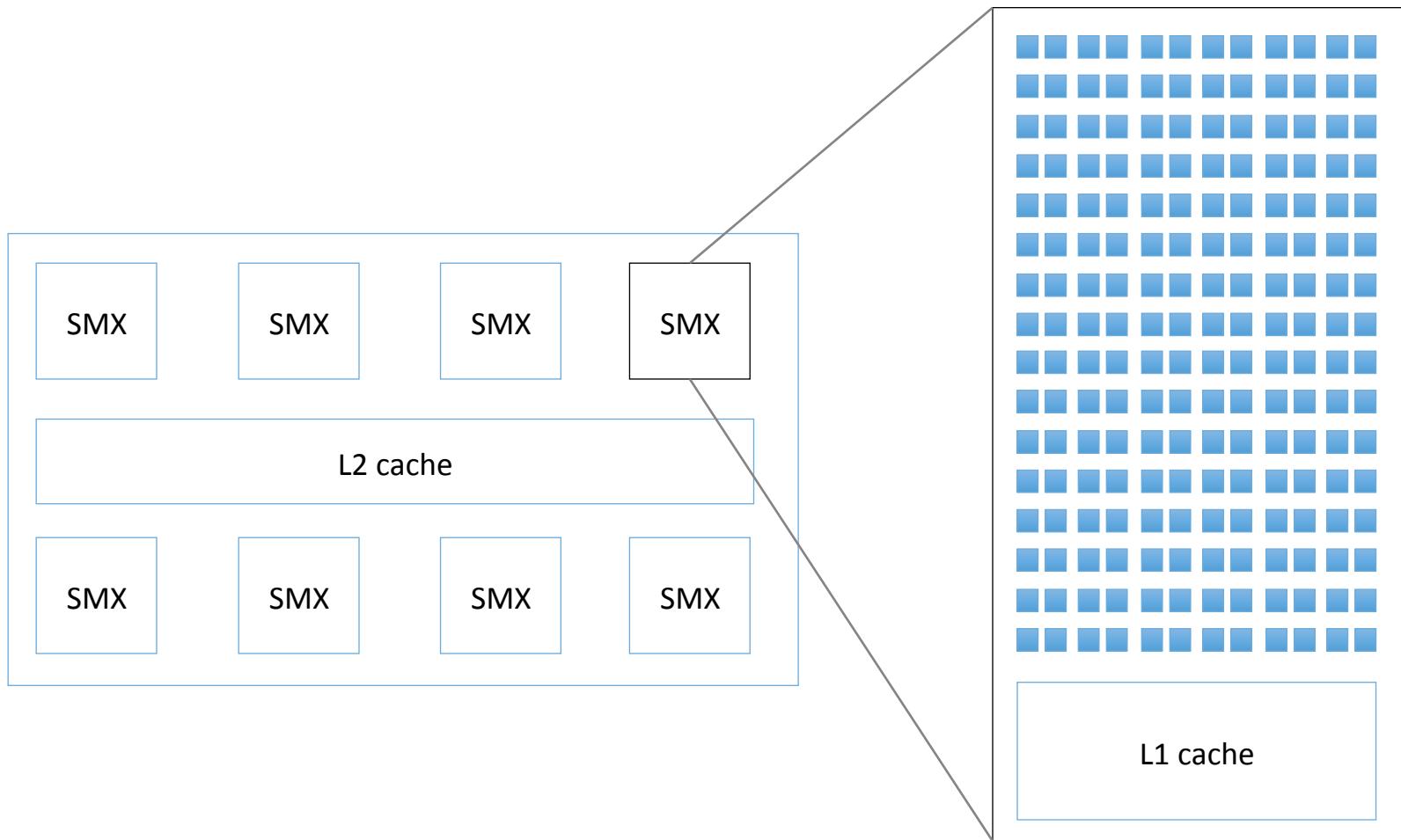
Sistemas con coprocesadores



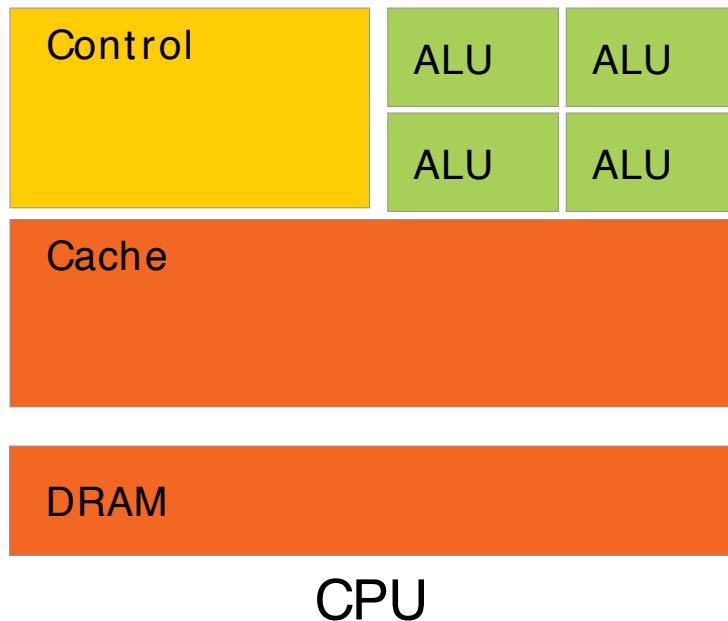
Tarjetas gráficas para la computación científica

- Las características principales de las tarjetas gráficas
 - un gran cantidad de núcleos
 - ejecución simultánea de hilos (SIMD)
 - registros privados por cada hilo
 - un tamaño limitado de memoria
 - requiere comunicación con un *host* CPU
- Implicancias para computación en GPU
 - require un alto grado de paralelización
 - una granularidad en el nivel de operaciones
 - intensidad aritmética alta

Arquitectura de tarjetas GPU (NVIDIA Kepler)



Programación de GPU



ALU - *Arithmetic Logit Unit*

Figura de “CUDA programming guide.”

Programación de GPU

- Interfaces de programación especiales son necesarias para la computación GPU
 - CUDA (*Compute Unified Device Architecture*) es una extensión C / C++ para GPUs de Nvidia
 - OpenCL (*Open Computing Language*) es un estándar de programación abierto para arquitectura heterogénea
 - OpenACC (*Open Accelerators*) es un estándar de programación para computación CPU / GPU con directivas de compilación

Programación de GPU

- Se distingue entre el *host* y *device*
 - el *host* controla las instrucciones y es normalmente la CPU
 - el *device* ejecuta las instrucciones y podría ser el coprocesador o la CPU
- El hilo maestro en el *host* realiza lo siguiente:
 1. Inicializar dispositivo de cómputo
 2. Definir dominio del problema
 3. Asignar memoria en *host* y dispositivo
 4. Copiar datos del *host* al dispositivo
 5. Iniciar el *kernel* de ejecución en el dispositivo
 6. Copiar datos del dispositivo al *host*
 7. Desasignar toda la memoria

Programar en OpenCL

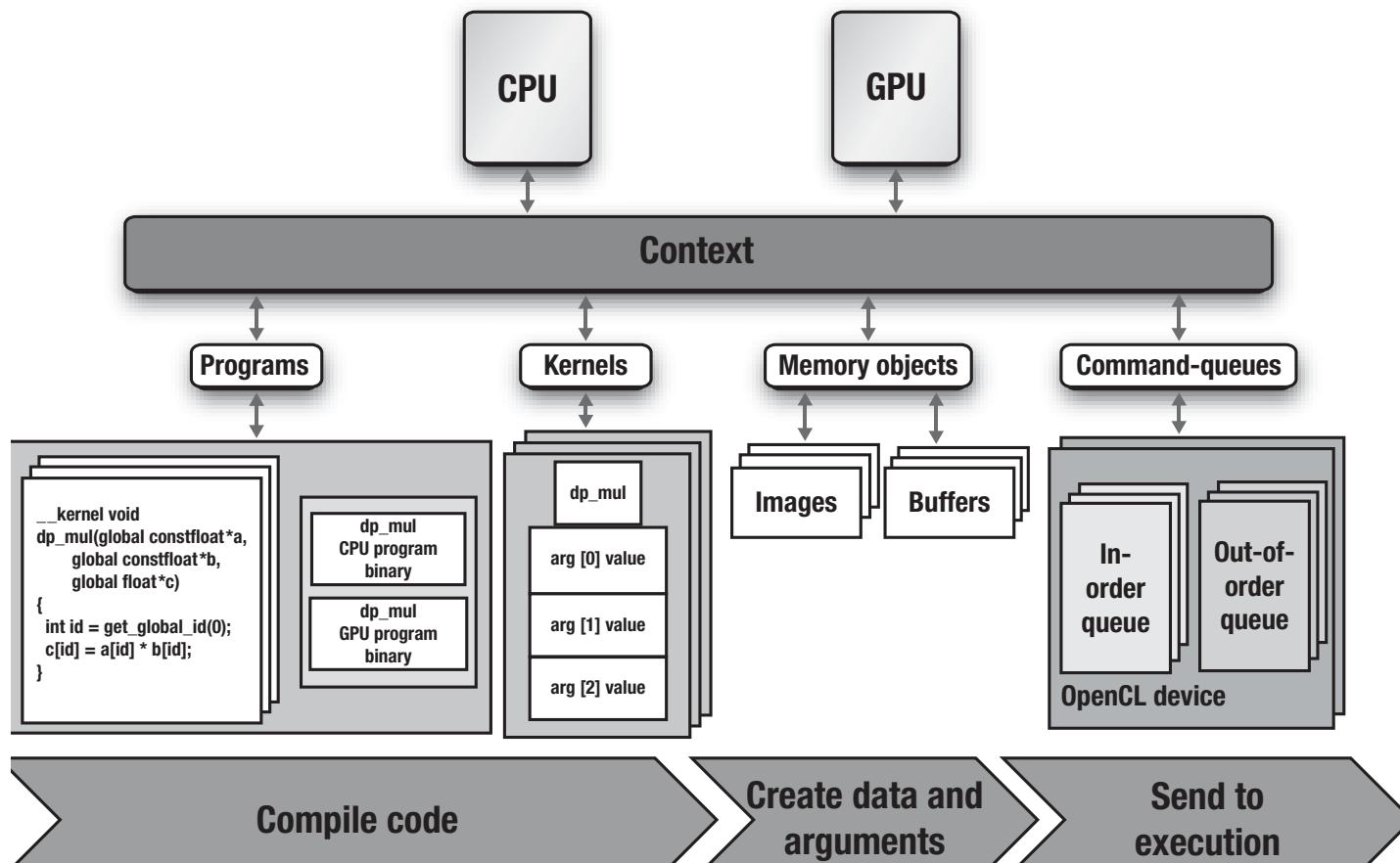


Figura de “OpenCL programming guide.”

Programar en OpenCL

- Primera parte de la programación OpenCL: “*boilerplate*”
- En este parte se tiene que inicializar y configurar los dispositivos
- Parece complicado pero es similar para todos los programas
 - un poco como “import numpy as np” en Python

Programar en OpenCL

1. Definir la plataforma

- obtener plataforma: `clGetPlatformIDs()`
- ver los dispositivos dentro de la plataforma: `clGetDeviceIDs()`
- crear un contexto para el dispositivo: `clCreateContext()`
- crear una cola de comandos para alimentar el dispositivo: `clCreateCommandQueue()`

Programar en OpenCL

- Puedes usar diferentes procesadores como dispositivo
 - la CPU es el anfitrión (*host*)
 - el dispositivo (*device*) podría ser la GPU, la CPU u otros coprocesadores
- En OpenCL, puedes elegir el dispositivo
 - `clGetDeviceIDs(plataforma, CL_DEVICE_TYPE_DEFAULT, 1, y dispositivo, NULL);`
 - `clGetDeviceIDs(plataforma, CL_DEVICE_TYPE_CPU, 1, y dispositivo, NULL);`
 - `clGetDeviceIDs(plataforma, CL_DEVICE_TYPE_GPU, 1, y dispositivo, NULL);`

Programar en OpenCL

- El código para los *kernel*s debe especificarse con cadenas de texto
 - incomodo de escribir
 - también se puede escribir el código en un archivo distinto
2. Crear y construir el programa
- generar objeto de programa:
clCreateProgramWithSource()
 - compilar el programa para construir la biblioteca de núcleos: **clBuildProgram()**

Programar en OpenCL

- Todos los objetos de memoria deben configurarse en el *host*
3. Configurar objetos de memoria
- Asignación e inicialización de vectores de entrada
 - Definir objetos de memoria OpenCL: `clCreateBuffer()`

Programar en OpenCL

- Los *kernel*s son las funciones reales del programa que se pueden ejecutar en los dispositivos

4. Definir los *kernel*s

- Crear objeto *kernel* desde el programa:
`clCreateKernel()`
- Adjuntar argumentos al *kernel*: `clSetKernelArg()`

Programar en OpenCL

- El *host* iniciará la ejecución de los *kernels* en el dispositivo
 - el dispositivo puede ser el mismo procesador que el *host*
5. Enviar comandos
- escribir buffers del *host* en la memoria global:
`clEnqueueWriteBuffer()`
 - poner en cola el núcleo para la ejecución:
`clEnqueueNDRangeKernel()`
 - leer el resultado: `clEnqueueReadBuffer()`

Programar en OpenCL

- Los *kernel*s son ejecutados por muchas instancias de elementos de trabajo, que tienen acceso a
 - algunas variables pasan como argumentos
 - búfers en la memoria global o local
 - constantes en la memoria global
 - memoria local y registros privados
 - algunas funciones especiales, por ejemplo,
 - `get_global_id()`: índice en el dominio
 - `get_local_id()`: índice en el grupo de trabajo
 - `get_group_id()`: índice del grupo de trabajo
 - `get_local_size()`: tamaño del bloque

Programar en OpenCL

- Los prefijos `h_` y `d_` para las variables en el *host* y el dispositivo no son obligatorios sino útiles
- Las rutinas del *kernel* se declaran con el prefijo `__kernel`
- Las rutinas del *kernel* se escriben desde el punto de vista de un solo hilo
 - similar a MPI, no como OpenMP

Programar en OpenCL

- Los *kernel*s se ejecuta por cada hilo

- Por ejemplo, un bucle tradicional

```
for (int i; i=0, i++) {  
    c[i] = a[i] + b[i];  
}
```

está programado con *kernel*s como

```
int id = get_global_id(0);  
c[id] = a[id] + b[id];
```

Programar en OpenCL

- Las variables declaradas como `_global` viven en la memoria global, compartida por todos los hilos
- Las variables declaradas como `_local` viven en la memoria local de un grupo de trabajo, compartido por todos los elementos de trabajo en ese grupo
 - no se puede comunicar información local a otros grupos
 - esto reduce el uso de memoria porque todos los hilos en el grupo de trabajo pueden usar la misma variable
 - sin embargo, los elementos de trabajo se ejecutan en orden arbitrario, por lo que necesita sincronización para usar la memoria local
 - usar una barrera: `barrier(CLK_LOCAL_MEM_FENCE);`

Programar en OpenCL

- Se debe especificar el tamaño de un grupo de trabajo
 - programar tal que el algoritmo no dependa del tamaño del grupo de trabajo
 - probar diferentes valores del tamaño del grupo de trabajo para optimizar el rendimiento
 - la mejor opción depende tanto del algoritmo como del hardware

Divergencia de elementos de trabajo

- Las GPUs conforman al metodología SIMD
- Recuerde la diferencia entre ‘grupo de trabajo’ y *warp*
 - grupo de trabajo: una colección de hilos definidos por el programador.
 - *warp*: el número de hilos ejecutados simultáneamente por un multiprocesador de transmisión

Divergencia de elementos de trabajo

- ¿Qué sucede con las ramas ('if-else') en la GPU?
- Supongamos que parte de los elementos de trabajo en un grupo toman la rama TRUE y otros la rama FALSE
 - todos los elementos de trabajo en un *warp* necesitan ejecutar las mismas instrucciones
 - efectivamente, ambas ramas se ejecutarán para todos los elementos de trabajo
- Se llama divergencia de elementos de trabajo (*work-item divergence*)

Divergencia de elementos de trabajo

- Ejemplo: condiciones de borde para métodos de diferencias finitas
 - intente ejecutar las condiciones de borde en grupos de trabajo separados
- En el peor de los casos, uno tiene una pérdida de rendimiento de un factor de 32
 - un elemento va a la rama cara
 - los otros elementos están inactivos

Divergencia de elementos de trabajo

- OpenCL tiene una función de barrera
 - `barrier(CLK_LOCAL_MEM_FENCE)`
 - para todos los elementos de trabajo en un grupo de trabajo
 - `barrier(CLK_GLOBAL_MEM_FENCE)`
 - para todos los elementos de trabajo en el kernel
- Asegúrese de que todos los elementos de trabajo lleguen a la barrera
 - no es evidente si hay ramas
 - de lo contrario, se producirá un *deadlock*

Reducción

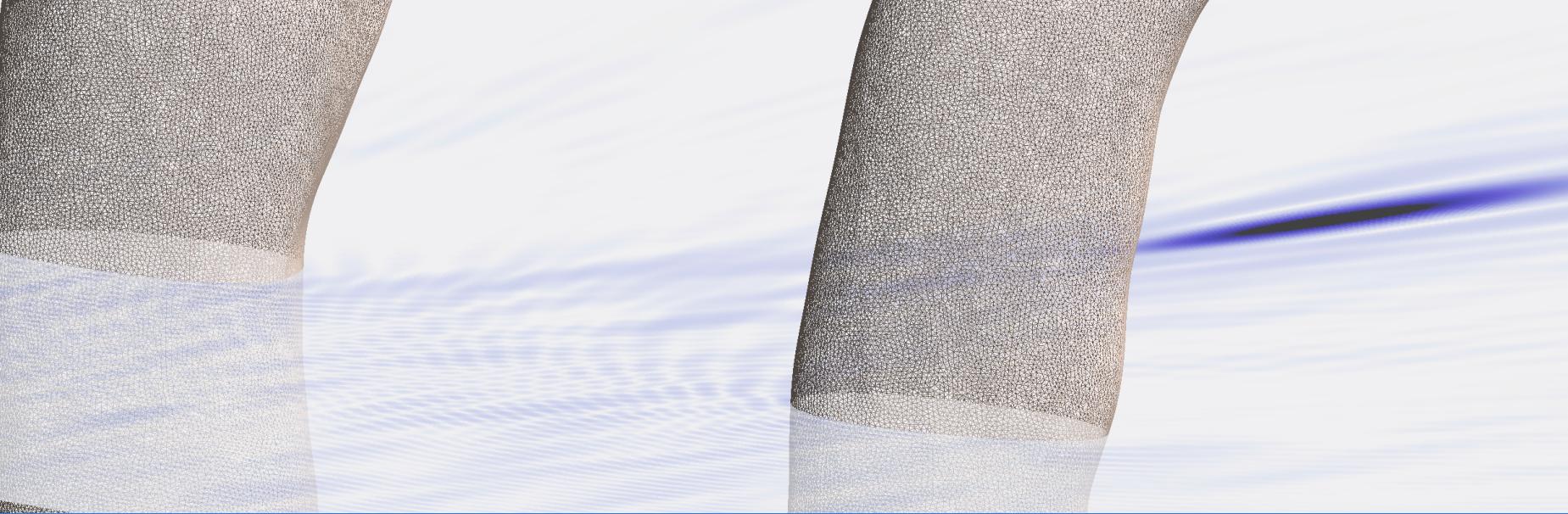
- Ejemplo: calcular la suma de un arreglo
- Aprovechar paralelización en la GPU
 - localmente, calcule el valor de cada elemento de trabajo
 - declare las variables localmente:
`__local float* local_values`
 - cada grupo de trabajo necesita calcular la suma de sus ítems
 - declare las variables globalmente:
`__global float* partial_sums`
 - calcule la suma global en el *host*
 - alternativamente, use el mismo algoritmo de sumas locales nuevamente

Arreglos bidimensionales

- El almacenamiento de matrices o mallas rectangulares se realiza de manera más eficiente con arreglos bidimensionales
- Puede usar particiones bidimensionales de matrices con OpenCL
 - `size_t localSizes [2] = {16, 8};`
 - `size_t globalSizes [2] = {512, 512};`
 - `clEnqueueNDRangeKernel (myqueue, mykernel, 2, NULL, globalSizes, localSizes, 0, NULL, NULL);`

Resumen

- Programar con OpenCL



IMT2112 - Algoritmos Paralelos en Computación Científica

Programar con OpenCL

Elwin van 't Wout

28 de noviembre de 2019



PONTIFICIA
UNIVERSIDAD
CATÓLICA
DE CHILE

Facultad de Matemáticas • Escuela de Ingeniería

imc.uc.cl