OpenCL

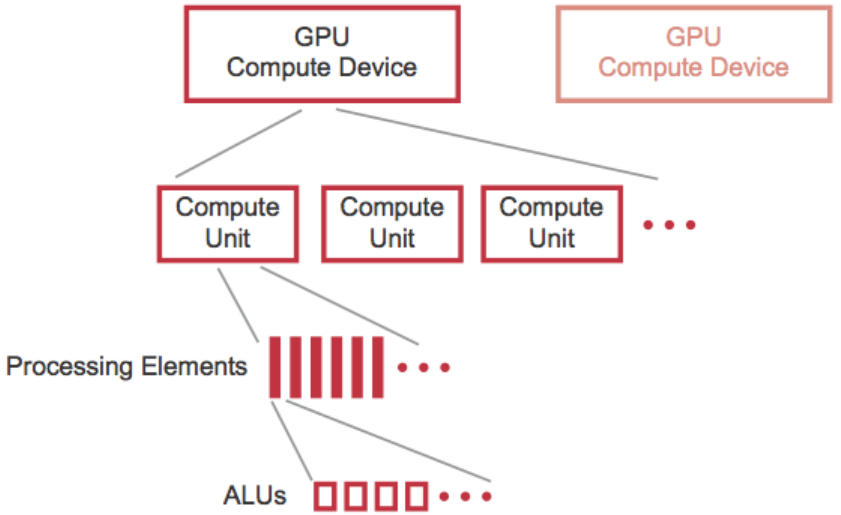
- Arquitectura estándar OpenCL
- Work groups y work items
- Tipos de memoria
- Creación, carga y compilación de kernels
- Creación de buffers y transferencia de datos
- Lanzamiento de kernels

OpenCL

- Estándar de programación de dispositivos con arquitectura multicore (GPUs, CPUs, etc.)
- Propuesto por Apple inicialmente y adoptado por la mayoría de fabricantes en la actualidad
- Soportado por GPUs Nvidia y AMD/ATI

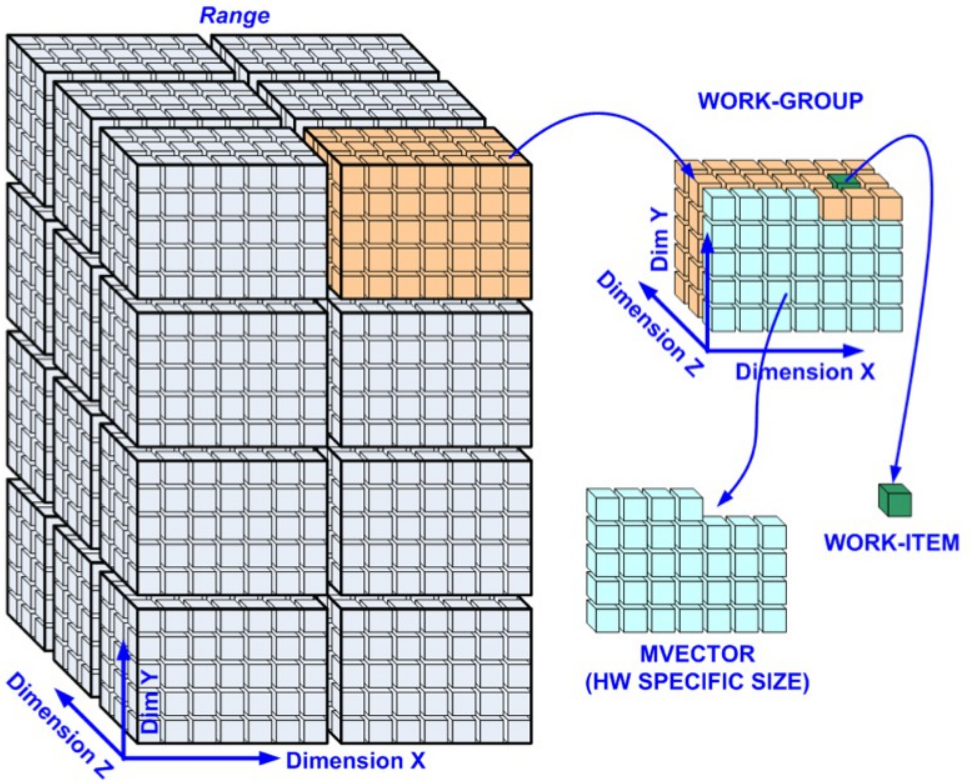
Arquitectura OpenCL

- Compute devices → GPUs, CPUs, etc.
- Compute Units → multiprocesadores MIMD
- Processing elements → procesadores SIMD



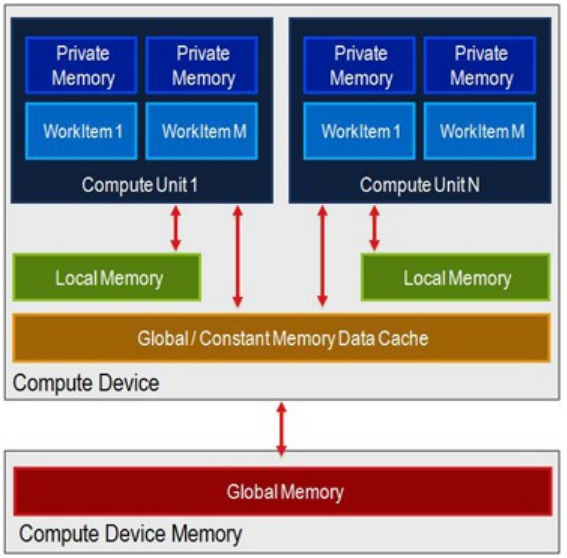
Modelo de programación

- Grid CUDA → ND Range
- Bloque CUDA → work group
- Hebra CUDA → work item



Tipos de memoria

- Privada → asociada a un work item
- Local → compartida por un work group
- Global → accesible desde todos los work items y desde el host



Programación de kernels

- Similares a los de CUDA
- Proporciona tipos vectoriales y una biblioteca de funciones muy rica
- Funciones C con el prefijo **kernel**

```
kernel void threadSumaGPU(global float *op1, global float
*op2, global float *res)
{
     // Calcular la posición que le corresponde al thread
     int pos = get_global_id(0);

     if (pos < get_global_size(0))
          res[pos] = op1[pos] + op2[pos];
}
```

Cualificadores de memoria

- **global** delante de punteros para indicar punteros a memoria global
- **local** delante de punteros o variables locales para variables en memoria local
- Sin cualificador: variable privada del kernel

```
kernel void threadSumaGPU(global float *memGlobal, local
float *memLocal)
{
     int float variablePrivada;
     local float bufferLocal[16];

}
```

Información del work item

- Numerosas funciones para obtener información del work group y work item:
  - get_work_dim() → número de dimensiones del ND range
  - get_global_size() → número total de work items
  - get_local_size() → número de work items por work group
  - get_num_groups() → número de work groups
  - get_global_id() → identificador global del work item
  - get_local_id() → identificador local del work item
  - get_group_id() → identificador del work group

Tipos vectoriales

- Tipos vectoriales (n puede ser 2, 3, 4, 8):
  - charn
  - ucharn
  - shortn
  - ushortn
  - intn
  - longn
  - ulongn
  - floatn
- Soporta operaciones aritméticas vectoriales y swizzle (v.x / v.s0, v.xy / v.s01, etc.)

Sincronización de hebras

- La función *barrier(flags)* sincroniza las hebras en un work group para garantizar un correcto acceso a memoria
- “flags” puede valer:
  - CLK_LOCAL_MEM_FENCE → sincroniza acceso a memoria local
  - CLK_GLOBAL_MEM_FENCE → sincroniza acceso a memoria global

Acceso atómico

- atomic_add()
- atomic_sub()
- atomic_xchg()
- atomic_inc()
- atomic_dec()
- atomic_min()
- atomic_max()
- etc.

Funciones predefinidas

- Multitud de funciones predefinidas
  - Información del work item
  - Matemáticas
  - Geométricas
  - Relacionales
  - Atómicas
  - Procesamiento de imágenes
- Usar la hoja de referencia:
  https://www.khronos.org/files/opencl-1-2-quick-reference-card.pdf

Procesamiento en el host

- Bastante más laborioso que CUDA
- Los pasos a seguir son los siguientes:
  - Obtención del dispositivo
  - Creación del contexto
  - Creación de la cola de comandos
  - Compilación y carga del kernel
  - Creación de buffers y transferencia de datos
  - Establecer argumentos del kernel
  - Lanzar kernel
  - Leer resultados

Obtención de la plataforma OpenCL

- Obtiene la implementación de OpenCL disponible localmente

```
cl_platform_id platform_id;
cl_uint ret_num_platforms;

cl_int err = clGetPlatformIDs(1, &platform_id,
    &ret_num_platforms);

if (err != CL_SUCCESS) {
    // No existe una implementación OpenCL disponible!
}
```

Iniciación de OpenCL

- Obtención los dispositivos existentes
- Pueden obtenerse todos los dispositivos registrados en la plataforma:
  - CL_DEVICE_TYPE_GPU
  - CL_DEVICE_TYPE_CPU
  - CL_DEVICE_TYPE_ALL

```
cl_device_id deviceIds[1]; // Array para los dispositivos obtenidos

// Obtener un único dispositivo de tipo GPU
err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU,
    1, deviceIds, NULL);

if (err != CL_SUCCESS) {
    // No existe un dispositivo compatible OpenCL disponible
}
```

Iniciación de OpenCL

- Creación del contexto con el dispositivo obtenido

```
cl_context context = clCreateContext(0, 1, &deviceIds[0],
      NULL, NULL, &err);

if (!context) {
  // Error de iniciación del dispositivo
}
```

- Creación de una cola de comandos para el contexto

```
commands = clCreateCommandQueue(context, deviceIds[0], 0, &err);

if (!commands) {
    // Error de creación de contexto
}
```

Carga del fuente del programa

- En el caso de OpenCL, el código del kernel no está integrado junto al código C/C++ del código host
- El código fuente debe obtenerse de un buffer de caracteres y compilarse como en el caso de los shaders

```
// Función propia para leer el programa desde un fichero
char *fuentePrograma = leerFuentePrograma("threadSumaGPU.cl");

cl_program program = clCreateProgramWithSource(context, 1,
    (const char**) &fuentePrograma, NULL, NULL);

if (!program) {
    // Error al crear programa desde fuente
}
```

Compilación del código fuente

- La compilación indicará si existen errores en el programa

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

if (err != CL_SUCCESS) {
    size_t len;
     char buffer[2048];

     cout << "Error: Failed to build program executable!” << endl;
     clGetProgramBuildInfo(program,  deviceIds[0],
          CL_PROGRAM_BUILD_LOG,
        sizeof(buffer),
          buffer, &len);
     cout << buffer << endl;

     exit(1);
}
```

Creación del kernel

- Una vez compilado se crea un kernel asociado al programa

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

if (!kernel || err != CL_SUCCESS) {
    cout << "Error: Failed to create compute kernel!” << endl;
    exit(1);
}
```

Creación de buffers

- Se crean los buffer necesarios y indicando si son de sólo lectura, sólo escritura o ambos:

```
cl_mem gpuA = clCreateBuffer(context, CL_MEM_READ_ONLY,
            sizeof(float) * tam, NULL, NULL);

cl_mem gpuB = clCreateBuffer(context, CL_MEM_READ_ONLY,
            sizeof(float) * tam, NULL, NULL);

cl_mem gpuRes = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
            sizeof(float) * tam, NULL, NULL);

if (!gpuA || !gpuB || !gpuRes) {
  // Error al crear buffer de datos
}
```

Transferencia de datos

- Se transfieren los datos desde memoria del host a los buffers creados en la GPU

```
// Copiar datos de entrada
err = clEnqueueWriteBuffer(commands, gpuA, CL_TRUE, 0,
    sizeof(float) * tam, a, 0, NULL, NULL);

if (err != CL_SUCCESS) {
    // Error al copiar datos de op1
}

err = clEnqueueWriteBuffer(commands, gpuB, CL_TRUE, 0,
    sizeof(float) * tam, b, 0, NULL, NULL);

if (err != CL_SUCCESS) {
    // Error al copiar datos de op2
}
```

Lanzamiento de kernel

- Cada argumento del kernel debe indicarse con la función *clSetKernelArg()*
- A continuación puede lanzarse el kernel mediante *clEnqueueNDRangeKernel()*
- Indicar el número global y local de work items
- Ojo!: **El número local de work items debe ser divisor del número global**

Lanzamiento de kernel

```
err = 0;
err |= clSetKernelArg(kernel, 0, sizeof(cl_mem), &gpuA);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &gpuB);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &gpuRes);

if (err != CL_SUCCESS) {
    // Error al establecer parámetros
}

dim = 1; // Número de dimensiones del NDRange
local = 1; // Número de work items locales
global = tam; // Número de work items globales
err = clEnqueueNDRangeKernel(commands, kernel, dim, NULL,
      &global, &local, 0, NULL, NULL);

if (err != CL_SUCCESS) {
  // Error al ejecutar kernel
}
```

Sincronización básica de la cola de comandos

- En la cola de comandos pueden introducirse varias ejecuciones sucesivas
- Si una ejecución requiere que las anteriores hayan finalizado, puede añadirse una barrera

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

// Esperar a que finalicen todos los comandos en la cola
clEnqueueBarrier(commands);

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

Lectura de resultados y liberación de objetos

- Tras finalizar el cálculo se obtienen los resultados del mismo añadiendo a la cola de comandos una lectura de buffer

```
// Terminar computación
clFinish(commands);


// Lectura de los datos al buffer res
err = clEnqueueReadBuffer(commands, gpuRes, CL_TRUE, 0,
        sizeof(float) * tam, res, 0, NULL, NULL);

if (!err) {
  // Error al leer buffer
}

clReleaseMemObject(gpuA);
clReleaseMemObject(gpuB);
clReleaseMemObject(gpuRes);
clReleaseProgram(program);
clReleaseKernel(kernel);
clReleaseCommandQueue(command);
clReleaseContext(context);
```