



# OPENCL

Open Computing Language

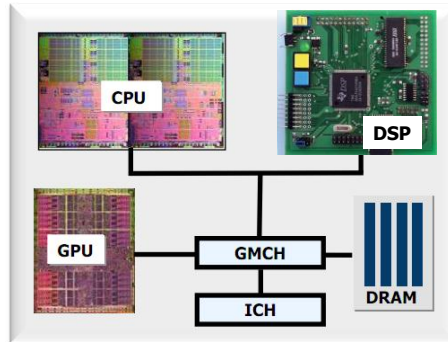
**Giuseppe Tagliavini**  
giuseppe.tagliavini@unibo.it

## OUTLINE

- **Introduction**
- OpenCL models
- Kernel Programming
- Host Programming
- Data/Task parallelism
- Case study
- Roadmap

## COMPUTING PLATFORMS

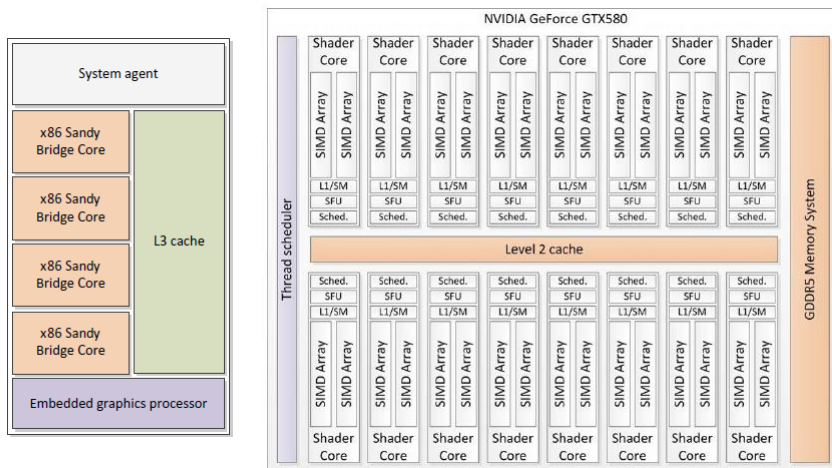
- A modern computing platform includes:
  - One or more **CPUs**
  - One or more **GPUs**
  - **Hw accelerators**



- **OpenCL (Open Computing Language)** is a framework for writing programs that uses all the resources of a heterogeneous platform

**Compute device abstraction**

## COMPUTING PLATFORMS: CPU AND GPU



**Different mappings between model and hw**

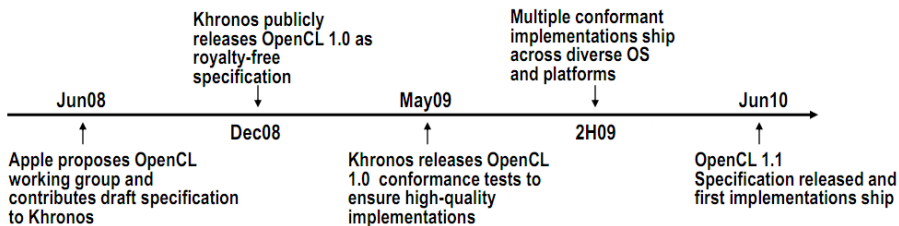
## OPENCL WORKING GROUP

- OpenCL is an **open standard** managed by the consortium *Khronos Group*.
  - **Diverse industry participation**, from processor vendors to application developers.
- **Apple made initial proposal and is very active in the working group.**



## OPENCL TIMELINE

- **6 months** from proposal to released OpenCL 1.0 specification
  - **Strong initial proposal**
  - **Shared commercial incentive**
- **18 months** between OpenCL 1.0 and OpenCL 1.1

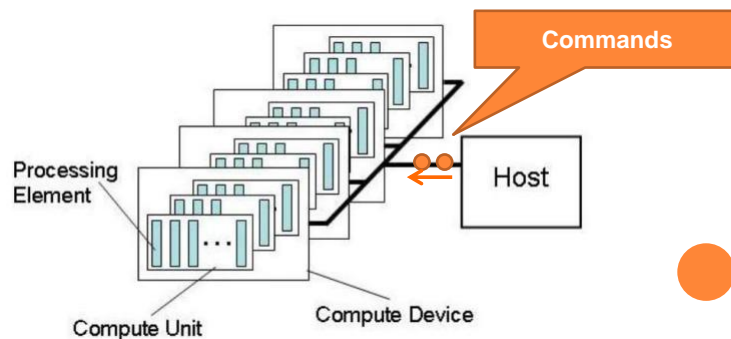


## OUTLINE

- Introduction
- **OpenCL Models**
- Kernel Programming
- Host Programming
- Data/Task parallelism
- Case study
- Roadmap

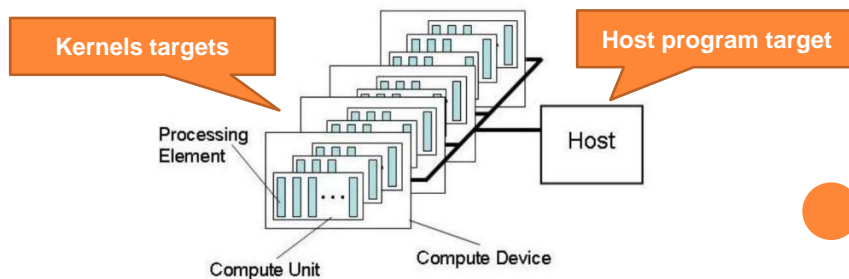
## PLATFORM MODEL

- An OpenCL application submits **commands** from the **host** to the **processing elements** within a **compute unit**.
- A *compute unit* executes a **single stream of instructions** (*no preemption*).



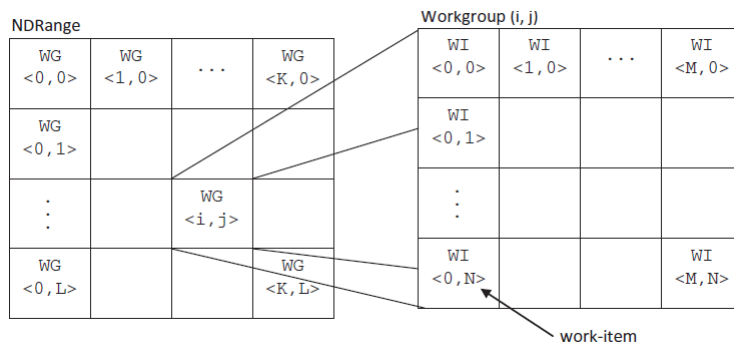
## EXECUTION MODEL (1/2)

- Execution of an OpenCL program occurs in two parts:
  - One or more **kernels** execute on one or more OpenCL devices.
  - a **host program** executes on the host environment.
- The host program defines the *context* for the kernels and manages their execution.



## EXECUTION MODEL (2/2)

- When a kernel is submitted for execution (by the host program), an **index space** is defined (called **NDRange**)
  - A kernel instance is called a **work-item**
  - Work-items are organized into **work-groups**

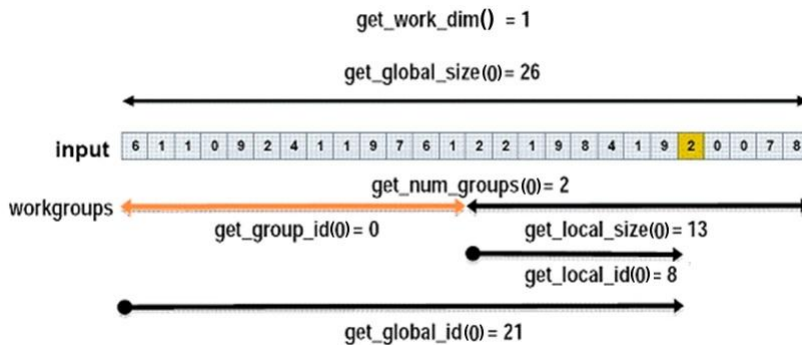


## EXECUTION MODEL: KERNEL API

- In OpenCL each thread has a **unique local index** and a **unique global index**.

<code>get_work_dim()</code>	Returns the number of dimensions in use
<code>get_num_groups(d)</code>	Number of work-groups for dim $d$
<code>get_group_id(d)</code>	Group index for dim $d$
<code>get_local_size(d)</code>	Local size for dim $d$
<code>get_global_size(d)</code>	Global size for dim $d$
<code>get_local_id(d)</code>	Local index for dim $d$
<code>get_global_id(d)</code>	Global index for dim $d$

## EXECUTION MODEL: 1D SPACE EXAMPLE



## MEMORY MODEL

### Private memory

- Per work-item

### Local memory

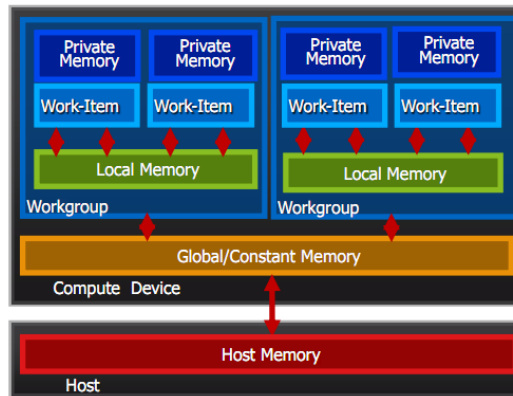
- Shared within a work-group

### Global/Constant memory

- Visible to all work-groups

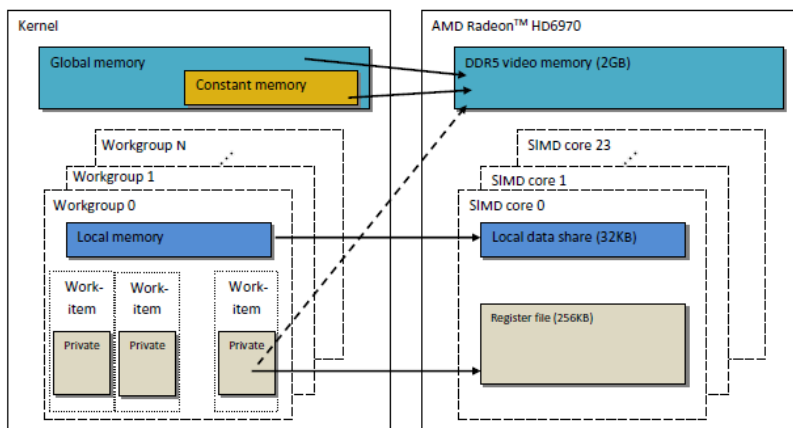
### Host memory

- System memory of the host environment



**Memory management is explicit**

## MEMORY MODEL MAPPING ON GPU



## MEMORY CONSISTENCY

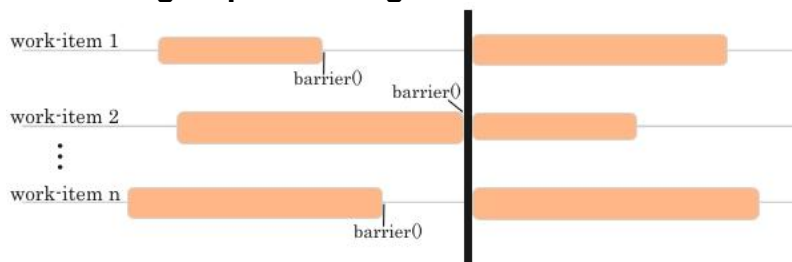
- Within a work-item, memory has **load / store consistency**.
- Local memory is consistent across work-items in a single work-group at a **work-group barrier**.
- Global memory is consistent across work-items in a single work-group at a **work-group barrier**.
- **There are no guarantees of memory consistency between different work-groups executing a kernel.**
  - *Data dependencies can also be defined and satisfied via the work queues and atomic operations.*

## MEMORY CONSISTENCY: BARRIER

- `barrier()`

All work-items in a work-group executing the kernel must execute this function before any are allowed to continue execution beyond the barrier.

**This function must be encountered by all work-items in a work-group executing the kernel.**



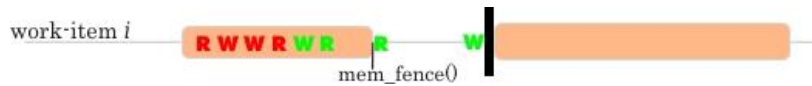
- **The barrier function also queues a memory fence (reads and writes) to ensure correct ordering of memory operations to local or global memory.**

## MEMORY CONSISTENCY: MEMORY FENCE

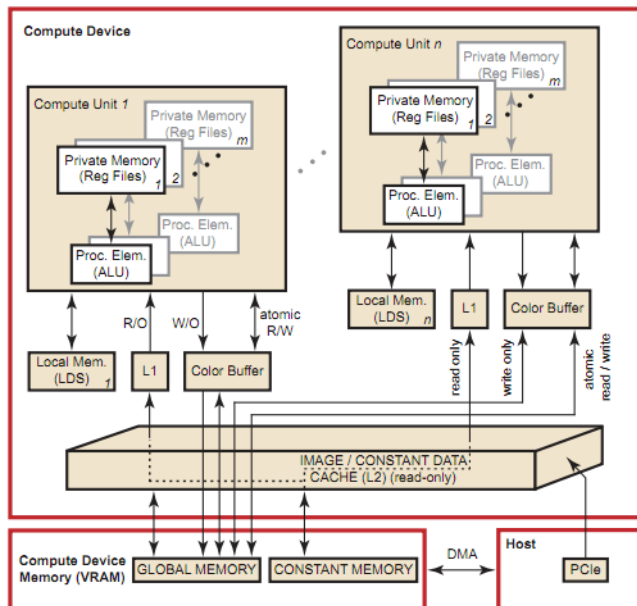
- `mem_fence()`
  - `read_mem_fence()`
  - `write_mem_fence()`

Loads and stores of a work-item executing a kernel are **ordered**: loads/stores *preceding* the `mem_fence()` will be committed to memory before any loads/stores *following* it.

**A fence may affect local memory, global memory, or both.**



## OPENCL MAPPING ON GPU ARCHITECTURE



## OUTLINE

- Introduction
- OpenCL Models
- **Kernel Programming**
- Host Programming
- Data/Task parallelism
- Case study
- Roadmap

## KERNEL PROGRAMMING: LANGUAGE

- **OpenCL C Language**
  - It is a **subset of C99**...
  - ... But without some features (standard C99 headers, function pointers, recursion, variable length arrays, and bit fields)
- Extensions to C language:
  - *Work-items and workgroups*
  - *Vector types*
  - *Synchronization*
  - *Address space qualifiers*
- Built-in functions
  - *Image manipulation*
  - *Work-item manipulation,*
  - *Specialized math routines*
  - ...

## KERNEL PROGRAMMING: DATA TYPES

### ○ Scalar data types

- char , uchar, short, ushort, int, uint, long, ulong, float
- bool, intptr\_t, ptrdiff\_t, size\_t, uintptr\_t, void, half (storage)

### ○ Image types

- image2d\_t, image3d\_t, sampler\_t

### ○ Vector data types

- Vector lengths 2, 4, 8, & 16 (char2, char4, char8, char16, float2, ...)
- Vector operations & built-in functions

```

int4 vi0 = (int4) -7;
int4 vi1 = (int4) (0, 1, 2, 3);
vi0.lo = vi1.hi;
int8 v8 = (int8) (vi0, vi1.s01, vi1.odd);

```

## EXAMPLE: VECTOR ADDITION

### ○ Plain C version:

```

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

```

## EXAMPLE: VECTOR ADDITION

- OpenCL kernel:

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

Global memory

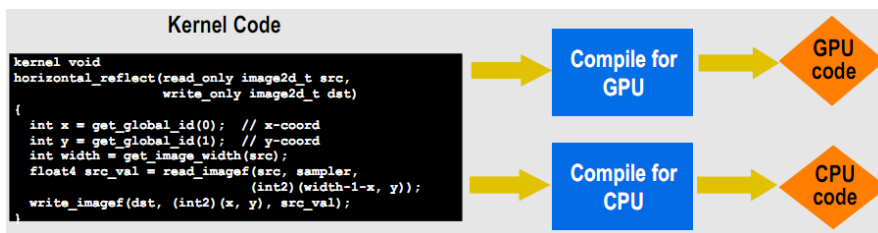
Private memory

Index-space test

**Data parallel code:** this kernel is executed n times, one for each vector element

## KERNEL BUILDING PROCESS (1/2)

- OpenCL framework creates a **target specific executable** using kernel sources.

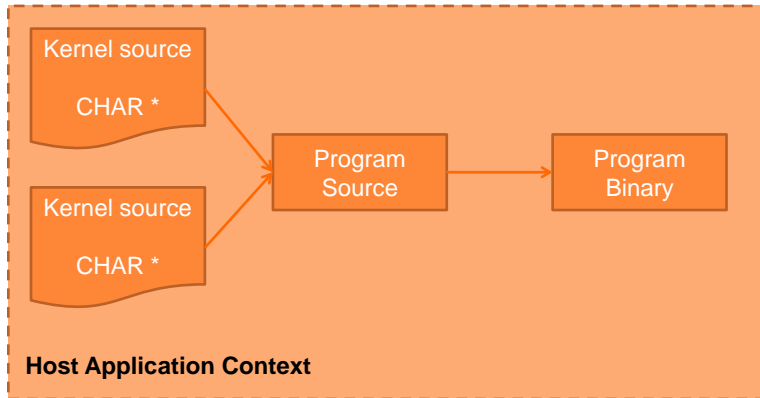


- OpenCL applications can:
  - use a pre-built binary (**offline approach**)
  - load and compile kernel sources (**online approach**)

Using Applications can load and build program executables online on its first instance for appropriate OpenCL devices in the system. These executables can now be queried and cached by the application

## KERNEL BUILDING PROCESS (2/2)

- Using an online approach, applications can load and build program executables **just-in-time** on its first instance for appropriate OpenCL devices in the system.
  - These executables can later be queried and cached by the application



## OUTLINE

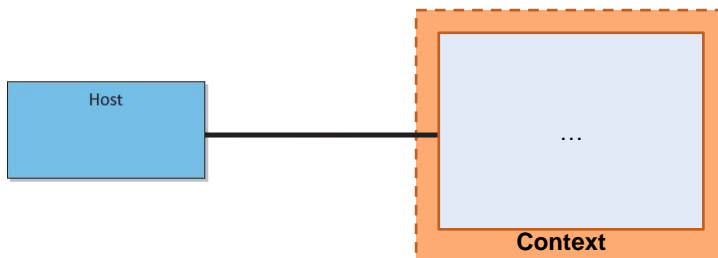
- Introduction
- OpenCL Models
- Kernel Programming
- **Host Programming**
- Data/Task parallelism
- Case study
- Roadmap

## HOST PROGRAMMING (1/10)

1. Create an **OpenCL context** for later operations

```
cl_context context;
context = clCreateContextFromType(0,
    CL_DEVICE_TYPE_GPU, 0, 0, 0);
```

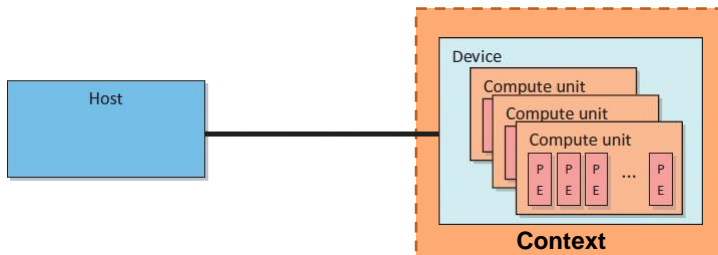
Create a context for GPU devices



## HOST PROGRAMMING (2/10)

2. Query all **devices** available to the context

```
size_t nContextDescriptorSize;
clGetContextInfo(context, CL_CONTEXT_DEVICES,
    0, 0, &nContextDescriptorSize);
cl_device_id aDevices =
    malloc(nContextDescriptorSize);
clGetContextInfo(context, CL_CONTEXT_DEVICES,
    nContextDescriptorSize, aDevices, 0);
```

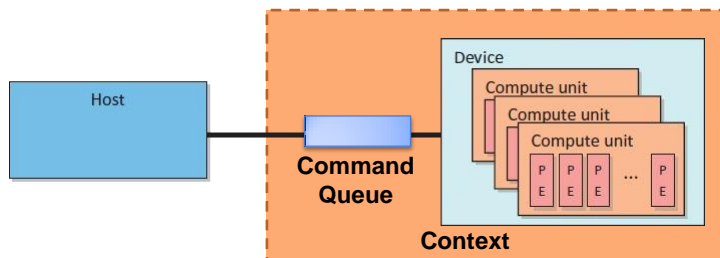


## HOST PROGRAMMING (3/10)

3. Create **command queues** for the required devices

```
cl_command_queue queue;
queue = clCreateCommandQueue(context,
    aDevices[0], 0, 0);
```

Create a command queue for the first device



## HOST PROGRAMMING (4/10)

4. Create and compile **program objects**

```
cl_program program;
program = clCreateProgramWithSource(context,
    1, source, 0, 0);
```

Kernel sources are provided as a string

```
clBuildProgram(program, 0, 0, 0, 0, 0);
```

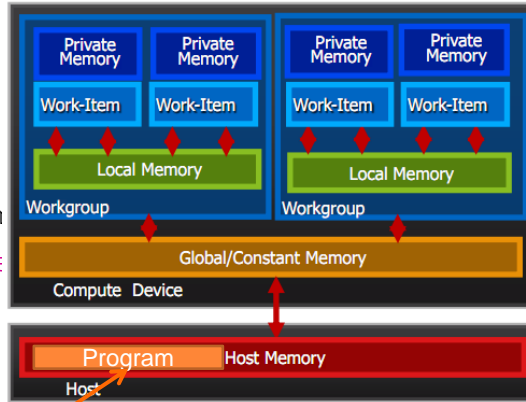
A program contains one or more kernels

## HOST PROGRAMMING (4/10)

### 4. Create and compile

```
cl_program program;
program = clCreateProgramWithSource(
    1, source, 0,
```

```
clBuildProgram(program, 0, 0, 0, 0, 0);
```



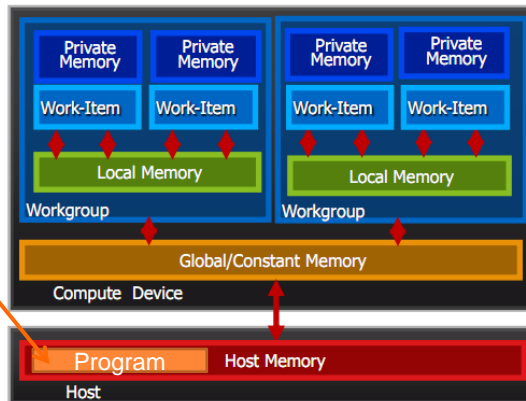
A program contains one or more kernels

## HOST PROGRAMMING (5/10)

### 5. Create kernel objects from program

```
cl_kernel kernel;
hKernel = clCreateKernel(program, "vecAdd", 0);
```

Specify kernel name

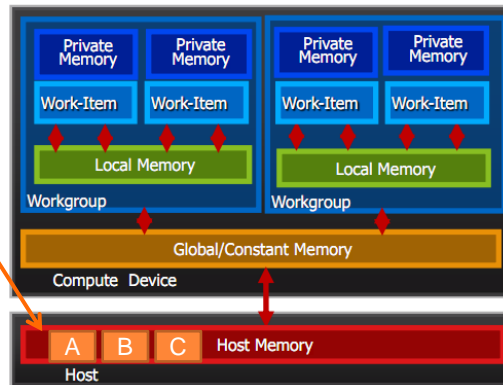


## HOST PROGRAMMING (6/10)

### 6. Allocate and initialize host memory data

```
float *pA = new float[n];
float *pB = new float[n];
float *pC = new float[n];
```

```
init(pA, n);
init(pB, n);
```



## HOST PROGRAMMING (7/10)

### 7. Create input and output memory objects

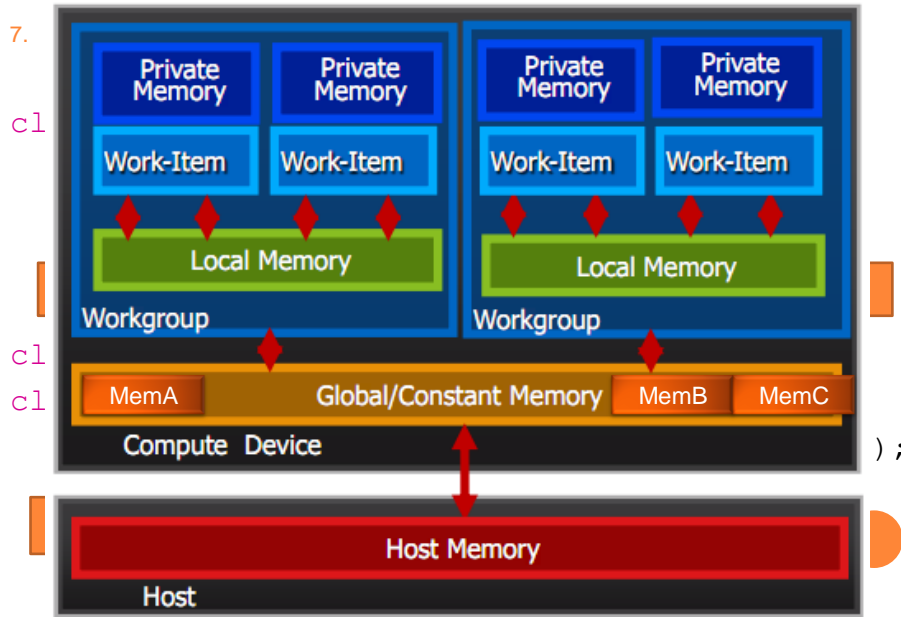
```
cl_mem deviceMemA = clCreateBuffer(context,
    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    n * sizeof(cl_float), pA, 0);
```

Create a read-only buffer for kernel input, pinned to host memory

```
cl_mem deviceMemB = /* ... */
cl_mem deviceMemC = clCreateBuffer(context,
    CL_MEM_WRITE_ONLY, n*sizeof(cl_float), 0, 0);
```

Create a write-only buffer for kernel input

## HOST PROGRAMMING (7/10)



## HOST PROGRAMMING (8/10)

### 8. Setup **kernel parameters** and **execute kernel**

```

clSetKernelArg(kernel, 0, sizeof(cl_mem),
    (void *)&deviceMemA);
clSetKernelArg(kernel, 1, sizeof(cl_mem),
    (void *)&deviceMemB);
clSetKernelArg(kernel, 2, sizeof(cl_mem),
    (void *)&deviceMemC);

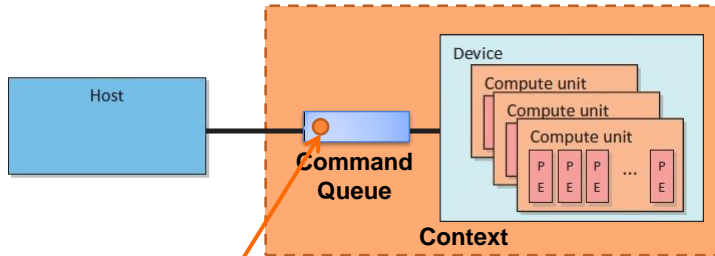
clEnqueueNDRangeKernel(queue, kernel, 1, 0,
    &n, 0, 0, 0, 0);

```

Global and local space size (0 = automatic sizing)

## HOST PROGRAMMING (8/10)

### 8. Setup **kernel parameters** and **execute kernel**



```
clEnqueueNDRangeKernel(queue, kernel, 1, 0,
    &n, 0, 0, 0, 0);
```

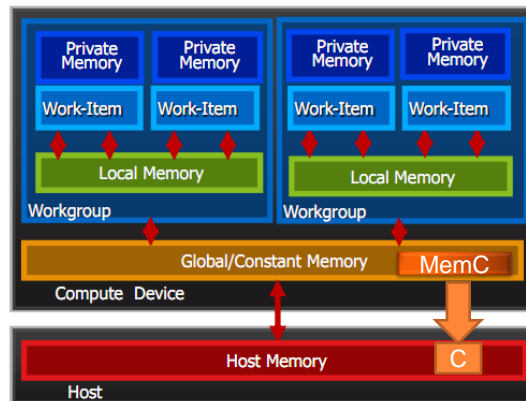
Global and local space size (0 = automatic sizing)

## HOST PROGRAMMING (9/10)

### 9. Copy results back to host

```
clEnqueueReadBuffer(context, deviceMemC,
    CL_TRUE, 0, n*sizeof(cl_float), pC, 0, 0, 0);
```

Blocking read from device to host



## HOST PROGRAMMING (10/10)

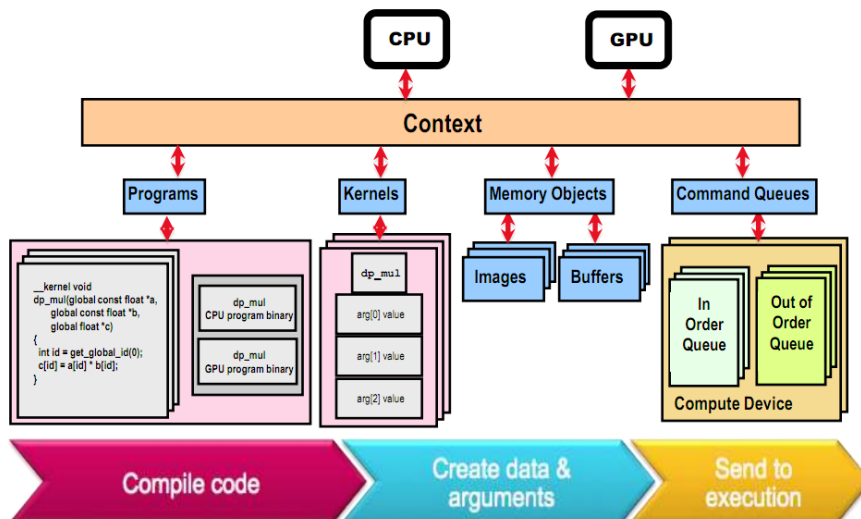
### 10. Host and device **memory cleanup**

```
delete [] pA;
delete [] pB;
delete [] pC;
```

```
clReleaseMemObj (deviceMemA);
clReleaseMemObj (deviceMemB);
clReleaseMemObj (deviceMemC);
```



## OPENCL PROGRAMMING SUMMARY

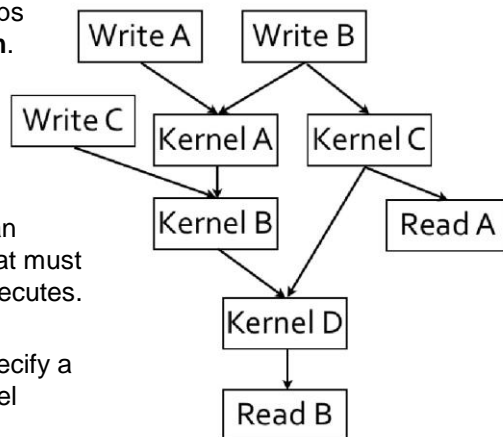


## OUTLINE

- Introduction
- OpenCL Models
- Kernel Programming
- Host Programming
- **Data/Task parallelism**
- Case study
- Roadmap

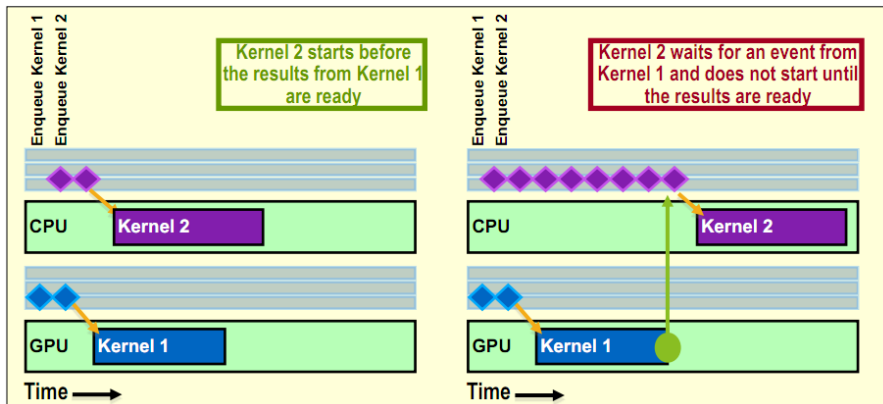
## DATA/TASK PARALLELISM

- The division of a kernel into work-items and work-groups supports **data parallelism**.
- OpenCL also supports **task parallelism**.
- Submitting a kernel, we can specify a **list of events** that must occur *before* the kernel executes.
- This feature enables to specify a **task graph** including kernel executions and memory transfers, also between different command queues .



## EVENTS

- Events are generated by **kernel completion, memory commands**, or may be **user-defined**.
- Events can be used to *synchronize kernel executions between queues*.



## OUTLINE

- Introduction
- OpenCL Models
- Kernel Programming
- Host Programming
- Data/Task parallelism
- **Case study**
- Roadmap

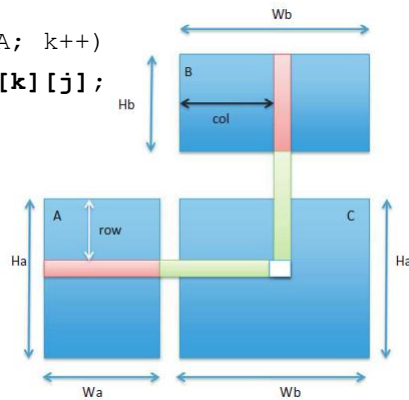


## MATRIX MULTIPLICATION

```

for(int i = 0; i < heightA; i++)
{
    for(int j = 0; j < widthB; j++)
    {
        C[i][j] = 0;
        for(int k = 0; k < widthA; k++)
            C[i][j] += A[i][k] * B[k][j];
    }
}

```



## MATRIX MULTIPLICATION: KERNEL 1

```

kernel void simpleMultiply(global float* outputC,
                           int widthA, int heightA,
                           int widthB, int heightB,
                           global float* inputA,
                           global float* inputB)
{
    int row = get_global_id(1);
    int col = get_global_id(0);
    float sum = 0.0f;
    for(int i = 0; i < widthA; i++)
        sum += inputA[row*widthA+i] * inputB[i*widthB+col];
    outputC[row*widthB+col] = sum;
}

```



## MATRIX MULTIPLICATION: KERNEL 2

```
kernel void coalescedMultiply(global float* outputC,
                             int widthA, int heightA,
                             int widthB, int heightB,
                             global float* inputA,
                             global float* inputB)
{
    local float aTile[BLOCK_SIZE][widthA];
    int row = get_global_id(1); int col = get_global_id(0);
    float sum = 0.0f;
    int x = get_local_id(0); int y = get_local_id(1);
    aTile[y][x] = a[row*widthA+x];
    barrier(CLK_LOCAL_MEM_FENCE);
    for(int i = 0; i < widthA; i++)
        sum += aTile[y][i] * inputB[i*widthB+col];
    outputC[row*widthB+col] = sum;
}
```



## MATRIX MULTIPLICATION: KERNEL 2

```
kernel void coalescedMultiply(global float* outputC,
                             int widthA, int heightA,
                             int widthB, int heightB,
                             global float* inputA,
                             global float* inputB)
{
    local float aTile[BLOCK_SIZE][widthA];
    int row = get_global_id(1); int col = get_global_id(0);
    float sum = 0.0f;
    int x = get_local_id(0); int y = get_local_id(1);
    aTile[y][x] = a[row*widthA+x];
    barrier(CLK_LOCAL_MEM_FENCE);
    for(int i = 0; i < widthA; i++)
        sum += aTile[y][i] * inputB[i*widthB+col];
    outputC[row*widthB+col] = sum;
}
```

1. Local memory access is FASTER
2. Transfers of adjacent memory addresses are COALESCED
3. Work-group size: (widthA, BLOCK\_SIZE)

## MATRIX MULTIPLICATION: KERNEL 3

```
kernel void coalescedMultiply(global float* outputC,
                             int widthA, int heightA,
                             int widthB, int heightB,
                             global float* inputA,
                             global float* inputB)
{
    local float aTile[BLOCK_SIZE][BLOCK_SIZE];
    ...
    for(int m = 0; m < widthA/BLOCK_SIZE; m++)
    {
        aTile[y][x] = a[row*widthA+m*BLOCK_SIZE+x];
        barrier(CLK_LOCAL_MEM_FENCE);
        for(int i = 0; i < BLOCK_SIZE; i++)
            sum += aTile[y][i] * inputB[i*widthB+col];
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    ...
}
```

## MATRIX MULTIPLICATION: KERNEL 3

```
kernel void coalescedMultiply(global float* outputC,
                             int widthA, int heightA,
                             int widthB, int heightB,
                             global float* inputA,
                             global float* inputB)
{
    local float aTile[BLOCK_SIZE][BLOCK_SIZE];
    ...
    for(int m = 0; m < widthA/BLOCK_SIZE; m++)
    {
        aTile[y][x] = a[row*widthA+m*BLOCK_SIZE+x];
        barrier(CLK_LOCAL_MEM_FENCE);
        for(int i = 0; i < BLOCK_SIZE; i++)
            sum += aTile[y][i] * inputB[i*widthB+col];
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    ...
}
```

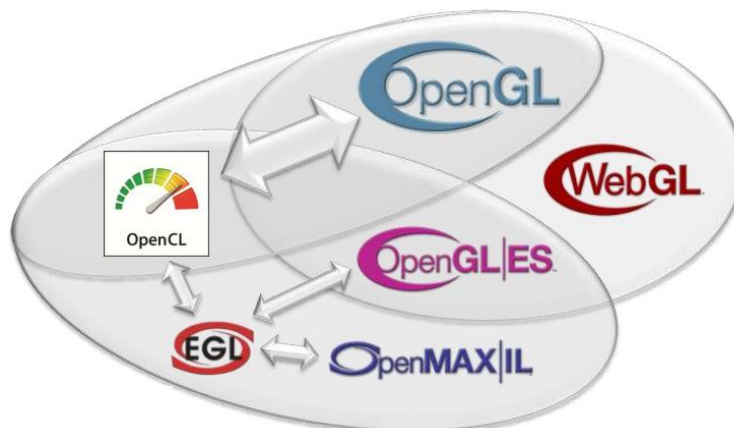
Local memory usage is limited

Work-group size: (BLOCK\_SIZE, BLOCK\_SIZE)

## OUTLINE

- Introduction
- OpenCL Models
- Kernel Programming
- Host Programming
- Data/Task parallelism
- Case study
- **Roadmap**

## OPENCL AND OPENGL ECOSYSTEM



## OPENCL ROADMAP

- **June, 2010** ► first OpenCL implementation for the ARM processor.
  - **September, 2010** ► Intel released details of their first OpenCL implementation for the Sandy Bridge chip architecture.
  - **March, 2011** ► Khronos Group announces the formation of the WebCL working group to explore defining a JavaScript binding to OpenCL.
  - **May, 2011** ► Nokia Research releases an open source **WebCL extension for the Firefox web browser**
  - **July, 2011** ► Samsung Electronics releases an open source prototype implementation of **WebCL for WebKit**
  - .....
- 