OpenCL
Open Computing Language

Flavio Vella

”Introduction to many/multicore parallel computing”

Corso Sistemi Operativi e Reti

Introduzione

OpenCL (Open Computing Language)

è uno standard open royalty-free per la programmazione parallela general-purpose su architetture eterogenee : CPU, GPU, DSP, CELL...

Introduzione

Chi ?

Khronos Group è un consorzio fondato da i maggiori vendors del campo HiTech (Apple, AMD, Nvidia, Intel ...)

Goal

Unificare la programmazione su architetture diverse (cross vendors) fornendo ai developers un'unica interfaccia di programmazione (top layer abstraction)

Vendors duty

Fornire e sviluppare driver che mappano le specifiche archetetturali sullo standard OpenCL

Architettura OpenCL

Necessità di astrazione: gerarchia di modelli

Platform model: è il modello che definisce e astrae le unita' di calcolo. A questo livello vengono definiti uno o piu' **OpenCL device** (Compute Device) connessi e gestiti da un host (CPU).

Execution model: definisce l'insieme di istruzioni che devono essere eseguite dai OpenCL device (kernel), e l'insieme di istruzioni che inizializzano e controllano l'esecuzione (host program) dei kernel stessi.

Memory model: definisce gli oggetti memoria, le tipologie di memoria e le relative modalità di accesso dall'host e dagli OpenCL device.

Programming model: definisce la logica di esecuzione parallela delle unità di calcolo: sui dati o sui task.

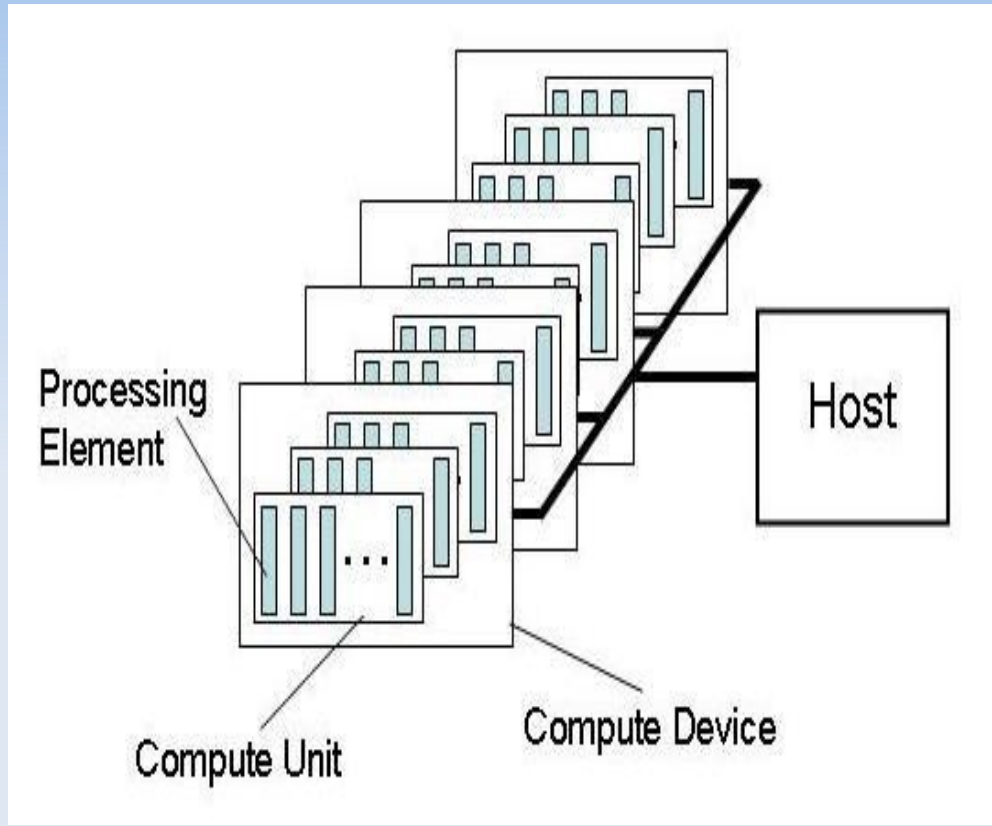
Framework model: insieme di API e estensioni di C99 per implementare kernel e host program

Platform model

Hardware abstraction: ad un **HOST** sono connessi uno o più OpenCL device detti **Compute Device**. Un **CD** è suddiviso in una o più **Compute Unit (CU)** che a loro volta contengono i **Processing Element (PE).**

Concetti di base: dall'host vengono lanciate istruzione da eseguire sui PE. Per eseguire questa operazione è necessario definire l'ambiente di esecuzione: il **context**.

Primitive: interrogazioni a basso livello verso i device, creazione e gestione dei context.

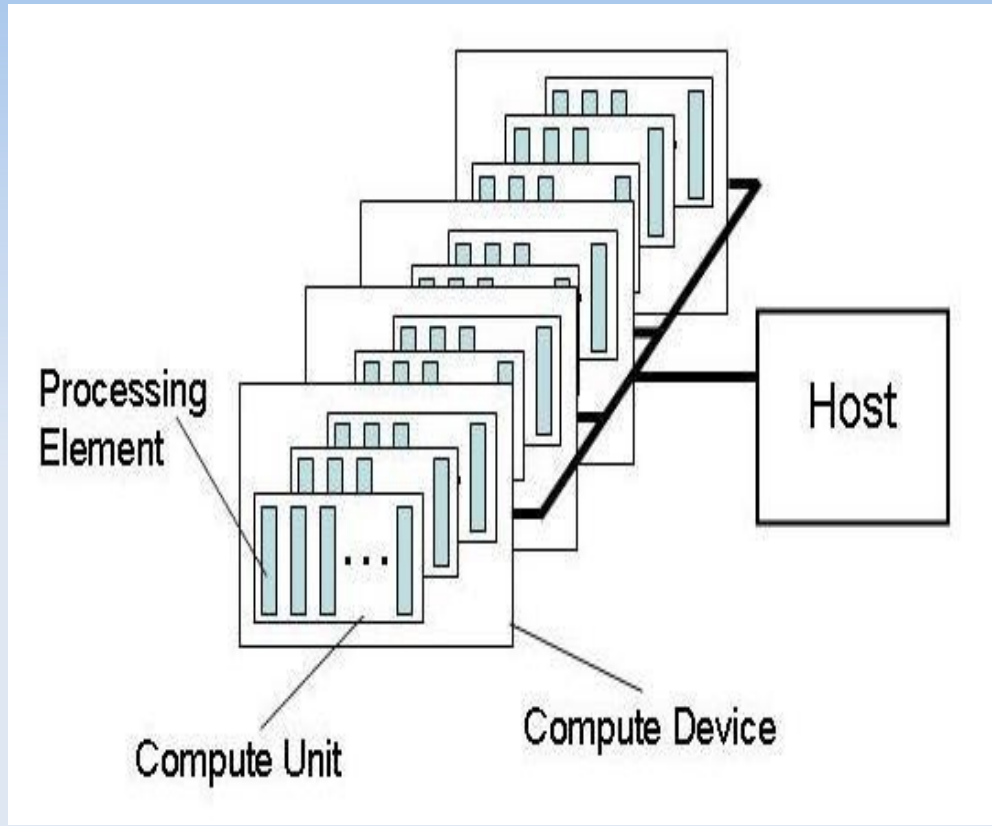


Platform model

Hardware abstraction: ad un **HOST** sono connessi uno o più OpenCL device. Un oclDevice è suddiviso in una o più **Compute Unit (CU)** che a loro volta contengono i **Processing Element (PE).**

Concetti di base: dall'host vengono lanciate istruzione da eseguire sui PE tramite gli oclDevice. Per eseguire questa operazione è necessario definire l'ambiente di esecuzione: il **context**.

Primitive: interrogazioni a basso livello verso i device, creazione e gestione dei context



Execution model

Host program : insieme di istruzioni che inizializzano e gestiscono l'ambiente di esecuzione dei Compute Device.

Kernel : insieme di istruzioni eseguite dal Compute Device.

HOST PROGRAM

Context : include un insieme di Compute Device, la memoria accessibile a questi ultimi e una o piu' command-queue usate per schedulare i kernel.

Command Queue : schedula l'esecuzione di un set di istruzioni sul device (uno o più kernel, trasferimenti in memoria etc..).

Memory object : aree di memoria nella quale il Compute Device può eseguire I/O.

Program : oggetto che associa il context e il sorgente del kernel.

Lancia e gestisce l'esecuzione dei kernel.

KERNEL PROGRAM

Quando il Kernel è sottomesso al CD per l'esecuzione dall'Host viene definito uno **spazio di indici**. Un'istanza è eseguita per ogni e punto dello spazio degli indici. Questa istanza è chiamata **work-item** ed è identificata da un punto (un id detto **Global-ID**) nello spazio degli indici. Ogni work-item esegue quindi lo stesso codice su un dato diverso.

Memory model (1/2)

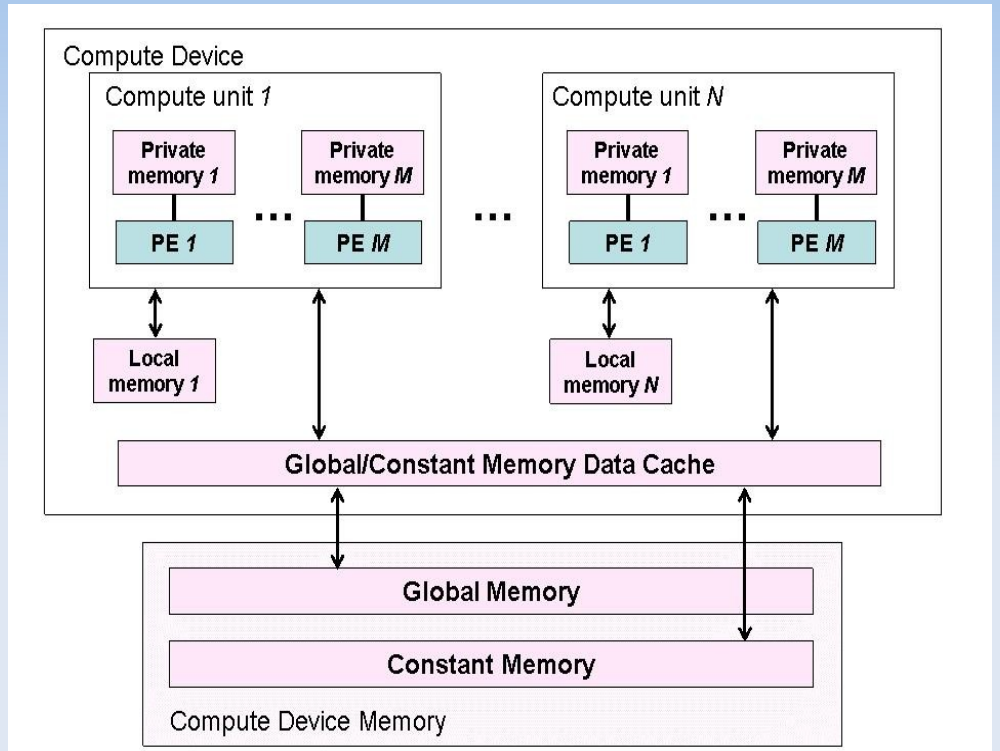
In OpenCL vengono distinte 4 tipologie di memoria accessibili dai work-item:

Global memory: regione di memoria condivisa e accessibile (r/w) da tutti i work item appartenenti a tutti i workgroup del medesimo oclDevice.

Constant memory: è una regione della global memory accessibile read only da tutti i work-item di tutti work-group.

Local memory: è una regione di memoria locale ai work-group.

Private memory: regione di memoria privata accessibile (r/w) solo ai work-item appartenenti a singoli work-group



	Global	Costant	Local	Private
Host	Dynamic allocation r/w access	Dynamic allocation r/w access	Dynamic allocation No access	No allocation No access
Kernel	No allocation r/w access	Static allocation r/w access	Static allocation r/w access	Static allocation r/w access

Memory model (2/2)

Tipologie di Memory object :

- Buffer object : memorizza una collezione di elementi unidimensionali
- Image object : memorizza una collezione di elementi bidimensionali o tridimensionali come texture, frame buffer o image.

Trasferimenti Host → Device :

- Copia esplicita : l'host schedula i comandi per il trasferimento dei dati e li esegue serialmente.
- Mapping/unmapping : consente di mappare un regione di memoria dall'host in quella del device.

Nota : i memory object sono descritti dal tipo cl_mem object (vedi reference) . Il kernel manipola questi oggetti.

Nota : le memoria nella global memory può essere organizzata pageable o pinned.

Programming model

Data parallel programming model

- Definisce il dominio computazionale del kernel : work-item o gruppi di work-item (work-groups) vengono associati agli elementi dei memory object tramite una relazione one-to-one.

Task parallel programming model (non ancora implementato)

- Definisce un modello nel quale una singola istanza del kernel viene eseguita indipendentemente sullo spazio degli indici.
- Logicamente definisce un singolo work-item per work-group.

Data-parallel: vengono eseguite le instruzione su tutti i dati associati a gruppi di thread

Task-parallel: ogni gruppo di thread esegue istruzioni su porzioni di dati diverse (no one-to-one)

Applicazioni OpenCL

Dichiararazione variabili.

Dichiarazione e inizializzazione dei dati da associare ai memory object (memObj).

Dichiarazione del context e discovery device.

Dichiarazione della command queue.

Inizializzazione dei memory object.

Dichiarazione e build del program.

Dichiarazione del kernel e setup dei parametri.

Esecuzione del kernel.

Get output.

Release degli object (memObj, program, kernel etc..).

Framework model

Estensione allo standard C99 sono presenti nuovi tipi di dato...

```
cl_context my_context;                  // OpenCL context
cl_command_queue my_command_queue;      // OpenCL command queue
cl_device_id* devices;                  // OpenCL device list
cl_program my_program;                  // OpenCL program
cl_kernel my_kernel;                    // OpenCL kernel
cl_mem input_buffer;                    // OpenCL memory object
cl_event device_execution;              // OpenCL event object
```

….
Vedi manuale...

Dichiarazione del context

Ci sono due modalità per la dichiarazione dei context :

Context creation from deviceID.

Context creation from device type.

Esempio → my_context = clCreateContext(0, device_number, *devices , NULL, &ci_error);

Esempio → my_context = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, NULL, NULL, &ci_error);

Nel primo caso è necessario ottenere il device number e la device list prima di dichiarare il context tramite la funzione clGetDeviceIDs.

Nel secondo caso invece il context è associato direttamente ad un device di tipo **CL_DEVICE_TYPE**, se esistente. Le informazioni sul device possono essere estratte con la funzione clGetContexInfo.

Dichiarezione Command-queue

La commad-queue consente di schedulare un insieme di operazioni che il CD deve eseguire: kernel, trasferimenti in/da memoria.

Prototipo → clCreateCommandQueue(context, devices_id, properties, err_code_return);

Esempio → my_command_queue = clCreateCommandQueue(my_context, devices[0], CL_QUEUE_PROFILING_ENABLE, &ci_error);

E' necessario specificare il context e il deviceID (*).

E' opzionale Command-queue-properties (terzo argomento passato alla funzione in esempio) che può essere specificata eventualmente successivamente (clSetCommandQueueProperty).

Questo parmetro permette di abilitare il profiling dei commandi e la modalità di esecuzione di quest'ultimi (order, out-of-order, blocked etc..)

(*) il deviceID deve corrispondere a quello specificato nel context.

Dichiarazione memory object

Creazione di un cl_mem buffer → clCreateBuffer

permette di allocare uno spazio nella memoria di un device.

è necessario specificare il context dove il memory object è utilizzabile e la modalità di accesso (read only o write only o read/write).

Read and Write dei buffer → clEnqueueReadBuffer, clEnqueueWriteBuffer

consente di trasferire i dati dalla memoria dell'host a quella del device (analogamente la clEnqueueReadBuffer).

è necessario specificare la command-queue e l'indirizzo di memory dell'host dove sono presenti i dati (input_data_buffer).

Esempio:

```
cl_input_buffer = clCreateBuffer (my_context, CL_MEM_READ_WRITE, sizeof (unsigned float) * dim, NULL,
&ci_err1);

ci_err1 = clEnqueueWriteBuffer(my_command_queue, cl_input_buffer, CL_TRUE, 0, sizeof(unsigned float) * dim ,
 (void *)input_data_buffer,0, NULL, &event);
```

Nota la clCreateBuffer puo' allocare e trasferire dati dalla memoria dell'host verso quella del device. Tuttavia l'esecuzione del trasfermento non avverrebbe tramite command-queue con la conseguente perdita delle command-queue-properties (profiling, out of order-execution).

Dichiarazione di program object

La funzione clCreateProgramWithSource permette di definire un program object a partire dal sorgente del kernel.

Prototipo → clCreateProgramWithSource(context, count , string_source, lenghts, err_code_ret);

Esempio → my_program = clCreateProgramWithSource(my_context, 1,(const char **) &c_source_CL, 0, 0);

my_context rappresenta il contesto di esecuzione del kernel

c_source_cl rappresenta il puntatore alla stringa che contiene il sorgente del kernel. (è comodo definirsi una funzione che legga il sorgente del kernel da file).

Costruzione del program object

E' necessario compilare e linkare un program sul device.

Prototipo

clBuildProgram(my_program, deviceID, device list, options_of_build, pfn_notify, arg_of_notify);

Esempio → clBuildProgram(my_program, 0, 0, 0, 0, 0);

Sorgente kernel

Le funzioni kernel iniziano con il qualificatore __kernel.

Accettano come argomenti SOLO indirizzi relativi ai memory object.

__global o __local : identificano l'indirizzo rispettivamente in global memory e local memory.

Gli argomenti di una funzione kernel vengono settati per mezzo di una funzione specifica nell'host program : clSetKernelArg.

Le istruzioni ammesse sono: dichiarazioni di variabili in local e private memory. Cicli, costrutti condizionali e funzioni definite nello standard (per lo più matematiche).

```
__kernel void kernelname (__global  type *
var_name, __local type * var_name2 )
{
istruzioni;
}
```

Dichiarazione kernel

Prima di poter eseguire un kernel sul device è necessario dichiarare il kernel object e definire i suoi parametri formali rispetivamente tramite le funzioni **clCreateKernel** e **clSetKernelArg**.

Prototipo clCreateKernel → clCreateKernel(program, *kernelname, *errcode_ret)

Dove program è un Program object costruito con successo.

Kernel_name è il nome della funzione definita nel sorgente del kernel dichiarata con il qualificatore __kernel.

Prototipo clSetKernelArg → clSetKernelArg(kernel, arg_index, arg_size, arg_value).

```
my_kernel = clCreateKernel (my_program, "kernelName", ci_kernel);

 //Set kernel first arg
 clSetKernelArg(my_kernel, 0, sizeof(cl_mem), (void *)&input_buffer);
// Set kernel second arg
 clSetKernelArg(my_kernel, 1, sizeof(cl_mem), (void *)&other_cl_mem_obj);
```

Esecuzione del kernel

Per eseguire un kernel e' necessario inserirlo come command in una command-queue. Tale operazione viene eseguita dalla funzione ocl

ClEnqueueNDRangeKernel(command-queue, kernel, work-dimension, offset, global-work-size, local-work-size,event-num-in-wait-list, event-wait-list, event)

Dove:

Command-queue : la comand-queue dove il kernel verrà in codato per l'esecuzione sul device.

Kernel: è il kernel object dichiarato in precedenza.

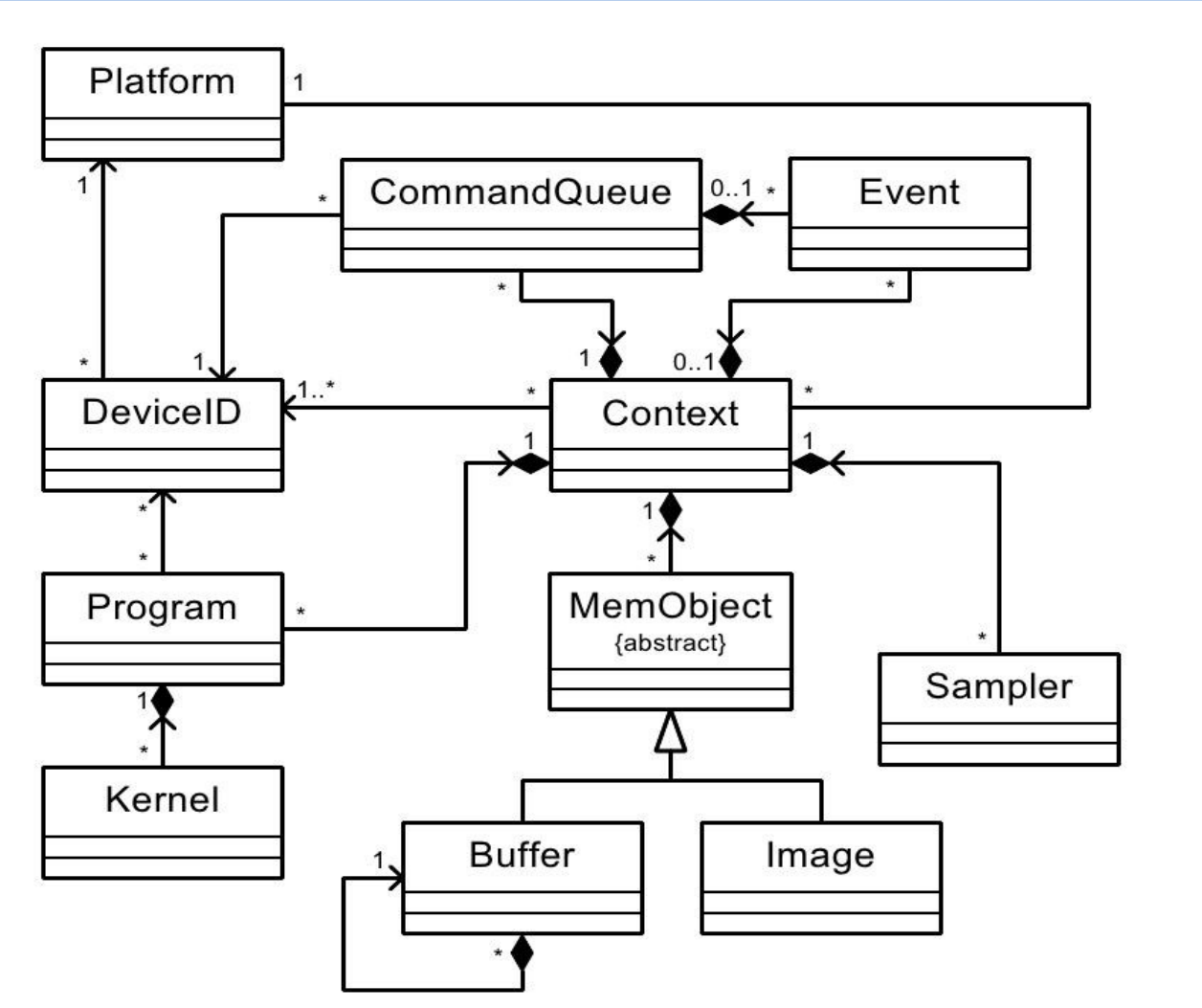
Work-dimension : rappresenta il numero di dimensioni dello spazio logico degli indici da associare ai work-group (mono-bi-tri dimensionale).

Global-work size : che definisce la dimensione dell'index space di esecuzione del kernel. Mapping dati-thread

Local-work size : che descrive il numero totale di work-item definite in un work-group

L'esecuzione del kernel è a runtime

Struttura di una applicazione OpenCL



Logica dell'index space 1/2

L'index space mappa il mem obj in spazi mono, bi o tri dimensionali e li associa ai work-item.

La dimensione dell'index space è data dalla global-work-size che viene definita in clEnqueueNDRangeKernel.

I work-item possono essere identificati tramite un indice univoco detto global_id.

Per esempio se volessimo sommare 2 array di 1024 elementi la global-work-size associata all' index space sarà 1024 e il global_id sarà definito da 0 a 1023.

```
__kernel void kernelname (__global  float * a, __global float
* b,__global float *c )
{
 unsigned int gid = get_global_id(0);
a[gid] = b[gid] + c[gid]
}
```

Per eseguire il kernel in esempio...

```
ci_error = clEnqueueNDRangeKernel(my_command_queue, my_kernel,1, 0 ,globalsize, 0, 0, 0, &device_execution);
```

Logica dell'index space 2/2

I **work-item** possono essere indentificati,non solo dal global-id, ma anche dalla coppia (indice di gruppo,indice locale) detti rispettivamente **group-id** e **local-id**. In questo caso il numero di gruppi di work-item in un work-group è dato dalla work-local-size che viene definita in clEnqueueNDRangeKernel.

Per esempio : vogliamo sommare 2 array di 1024 elementi: la global-work-size associata all'index space sara' 1024 e il global-id sara' definito da 0 a 1023.

Se utilizziamo la coppia (group-id,local-id) e definiamo la local-work-size pari a 128 si avrà:

Local-id definito da 0 a 127 che mapperà 128 work-item in un work-group.

Lo spazio del group-id sara' definito dalla global-work-size/local-work-size → 0 a 7

In conclusione ogni work-item sarà identificato dalla coppia (group-id,local-id)

```
__kernel void kernelname (__global  float * a, __global float * b,__global float *c )
{
 unsigned int tid = get_local_id(0);
unsigned int bid = get_group_id(0);
unsigned int ng = get_num_groups(0);
unsigned int ogid = tid+bid*ng;
a[ogid] = b[ogid] + c[ogid];
}
```

```
ci_error = clEnqueueNDRangeKernel(my_command_queue, my_kernel,1, 0 ,globalsize, localsize,
0, 0, &device_execution);
```

Approfondimento: event e profiling

E' utile calcolare il tempo di esecuzione dei kernel o di altri comandi in command-queue quali la writeBuffer...

Abilitare il profiling al momento della definizione della command-queue settando la queue-properties nel modo seguente :

```
my_command_queue = clCreateCommandQueue(my_context, devices[0], CL_QUEUE_PROFILING_ENABLE, &ci_error);
```

Se si vuole misurare il tempo di esecuzione del kernel è necessario passare un oggetto event in fase di esecuzione del kernel:

```
clEnqueueNDRangeKernel(my_command_queue, my_kernel,1, 0 ,globalsize, localsize, 0, 0, &device_execution);
```

A questo punto basta passare device_execution ad una funzione del tipo :

```
double utils_execution_time(cl_event &event){
    cl_ulong start, end;
    double totalTime;
    clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &end, NULL);
    clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &start, NULL);
    totalTime = (end - start) / 1.0e6);
    return totalTime;
}
```

Conclusioni

Lo standard è alla versione 1.1 .

AMD ha fornito il supporto OpenCL 1.1 su CPU (con sse) , GPU e APU. Stream SDK2.3

Nvidia supporta OpenCL 1.1 solo su GPU.

IBM ha rilasciato i driver OpenCL per BLADE QS22 (CELL based) e JS23 (PowerPC based).

Intel supporta OpenCL 1.1 solo CPU.

References

http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

"Introduction to Parallel Computing". Grama, Gupta, Karypis, Kumar.