Alma Mater Studiorum · Università di Bologna

**SCUOLA DI SCIENZE**

**Corso di Laurea in Ingegneria e Science Informatiche**

# CONVERSIONE A OPENCL DI UN SIMULATORE CARDIACO PARALLELO

**Relatore:**
**Chiar.mo Prof.**
**Moreno Marzolla**

**Presentata da:**
**Roberto Montalti**

**III Sessione di laurea**
**Anno Accademico 2021/2022**

# Sommario

In questa tesi discuteremo come è possibile effettuare la traduzione di un software parallelo scritto in linguaggio CUDA ad uno in linguaggio OpenCL. Tratteremo le tecnologie utilizzate per lo sviluppo di un simulatore cardiaco parallelo e discuteremo in particolar modo come derivare da queste una versione che ne permetta l'esecuzione su schede video e processori arbitrari. Questa versione verrà messa poi a confronto con quelle già esistenti, per analizzarne prestazioni ed eventuali cambiamenti strutturali del codice.

Quanto affermato sopra è stato possibile in gran parte grazie ad un *wrapper* chiamato SimpleCL pensato per rendere la programmazione OpenCL simile a quella in ambiente CUDA. OpenCL permette di operare con le unità di calcolo in maniera molto astratta, ricordando vagamente i concetti di astrazione di memoria e processori della controparte NVIDIA. Ragionevolmente SimpleCL fornisce solamente una interfaccia che ricorda chiamate CUDA, mantenendo il flusso sottostante fedele a quello che si aspetterebbe OpenCL.

# Indice

# Elenco delle figure

# Capitolo 1

# API e piattaforme per lo sviluppo di applicazioni parallele

## 1.1 Significato

Quando parliamo di API (Application Program Interaface) intendiamo un insieme di procedure pensate per l'utilizzo di un particolare servizio. Il termine piattaforma invece è rivolto a sistemi più complessi che spesso comprendono una propria API per poterci interagire. Un esempio di piattaforma può includere CUDA la quale oltre una interfaccia di programmazione, fornisce anche servizi di più basso livello quali compilatori, ambienti di esecuzione del codice CUDA e driver per i dispositivi di calcolo.

## 1.2 OpenMP

Una delle API per lo sviluppo parallelo su CPU più famose al momento si chiama OpenMP [2] (Open Multiprocessing), è compatibile con i principali sistemi operativi disponibili nel mercato e permette lo sviluppo di applicazioni parallele su sistemi a memoria condivisa.

Prevede l'utilizzo di direttive a compilatore, chiamate di libreria e variabili d'ambiente che ne dettano il funzionamento. OpenMP supporta alcuni dei linguaggi più famosi e utilizzati degli ultimi vent'anni quali C, C++ e Fortran.

OpenMP promette la creazione e gestione automatica di pool di thread, assegnando ad ognuno di questi una porzione di codice specificata da una direttiva a compilatore. Il funzionamento della libreria presuppone la conoscenza del modello di esecuzione *fork-join* dove il processo *master* (quello che viene avviato non appena viene eseguito il programma) crea la *struttura parallela* formata da un insieme di thread lavoratori.

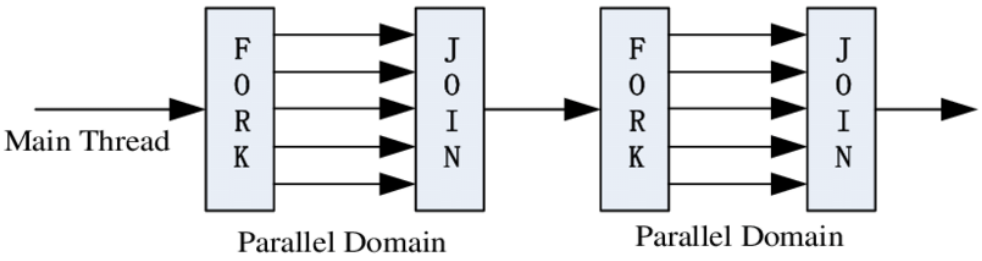


Figura 1.1: OpenMP execution model

Questo breve cenno serve solamente lo scopo di mettere a conoscenza il lettore dell'esistenza di API di questo tipo, tuttavia non tratteremo con particolare riguardo OpenMP. Infatti gran parte del testo tratterà da un punto di vista pratico quella che è stata la traduzione di un programma da codice CUDA ad OpenCL. OpemMP seppur un importante tecnologia, utilizzata anche nel contesto del simulatore cardiaco che analizzeremo, non permette un aumento di performance tale da poter competere con tecnologie e framework che fanno ausilio di GPU. A tal riguardo i prossimi capitoli tratteranno esclusivamente introduzioni ed esempi ai concetti necessari per raggiungere lo scopo prefissato in questa tesi, riscrivere un simulatore in OpenCL.

## 1.3 Introduzione a CUDA

CUDA è una piattaforma per il calcolo parallelo e un linguaggio di programmazione simile al C++ creato da NVIDIA [7]. La piattaforma porta con se un insieme di componenti essenziali al suo funzionamento quali driver, compilatori e ambienti di esecuzione. Ad oggi CUDA è una tecnologia in continuo sviluppo utilizzata in una miriade di contesti tra cui *machine learning*, sviluppo di applicazioni 3D in tempo reale e *ray tracing*. Purtroppo CUDA è pensato solo per dispositivi NVIDIA e utenti che non dispongono di dispositivi del marchio saranno costretti ad utilizzare tecnologie alternative.

### 1.3.1 Il modello di programmazione CUDA

Lo sviluppo di codice CUDA presuppone la conoscenza di alcuni concetti, c'è una chiara distinzione tra codice GPU e codice CPU. Infatti in base al tipo di codice che si scrive è fondamentale ragionare in un contesto astratto definito dalla piattaforma.

CUDA descrive una GPU allo sviluppatore come una griglia di gruppi di thread, chiamati *blocchi* [6]. Un blocco non è altro che che un insieme di thread che può essere eseguito in ordine arbitrario dalla scheda grafica.

I thread vengono messi in attesa di esecuzione (*scheduling*) in gruppi da 32, tali gruppi vengono chiamati CUDA warp e solo la minima unità di scheduling della piattaforma. I warp vengono messi in esecuzione dai *warp scheduler* di questi ce ne possono essere più di uno, dando la possibilitá di mettere in esecuzione più di un warp alla volta.

### 1.3.2 Un esempio di codice CUDA

Come citato in precedenza, la piattaforma crea una linea ben marcata tra codice CPU e codice GPU. Infatti CUDA mette a disposizione del programmatore una serie di simboli che permettono di creare funzioni GPU, i cosiddetti *kernel*.

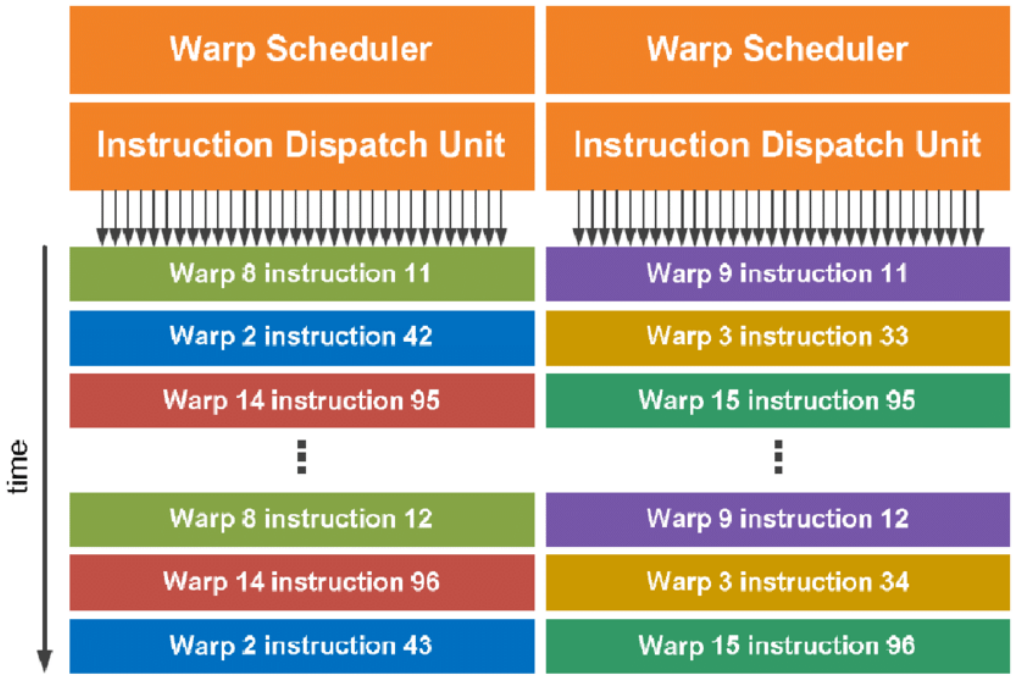


Figura 1.2: CUDA Warps scheduling

Un *kernel* non è altro che una sezione di codice che verrà eseguita da ogni singolo thread CUDA. Il numero di thread che verranno lanciati è dato da dei parametri passati al *kernel*. Vediamo di seguito la struttura di una chiamata *kernel* per capire meglio di cosa stiamo parlando.

```
#define N 256

__global__ void vector_add(float *out, float *a, float *b)
{
    size_t i = threadIdx.x;
    out[i] = a[i] + b[i]
}

void main()
{
```

```
    float *a, *b, *out;
    float *d_a, *d_b, *d_out;

    a = (float*)malloc(sizeof(float) * N);
    b = (float*)malloc(sizeof(float) * N);
    out = (float*)malloc(sizeof(float) * N);

    // Allocate device memory for a
    cudaMalloc((void**)&d_a, sizeof(float) * N);
    cudaMalloc((void**)&d_b, sizeof(float) * N);
    cudaMalloc((void**)&d_out, sizeof(float) * N);

    // Transfer data from host to device memory
    cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, sizeof(float) * N, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, sizeof(float) * N, cudaMemcpyHostToDevice);

    vector_add<<<1,N>>>(d_out, d_a, d_b, N);

    cudaMemcpy(out, d_out, sizeof(float) * N, cudaMemcpyDeviceToHost);

    // Cleanup after kernel execution
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_out);
    free(a);
}
```

La chiamata al *kernel* `vector_add` comprende una sintassi insolita, cioè `<<<N_BLOCKS, N_THREADS_PER_BLOCK>>>`. I valori passati all'interno delle triple parentesi angolari sono esattamente: il numero di blocchi da lanciare e quanti thread ogni blocco ospita al suo interno.

Le funzioni *kernel* inoltre, sono in grado di accedere a informazioni riguardanti il thread che le sta eseguendo, quindi l'id interno al blocco del thread, l'id del blocco o la dimensione del blocco a cui appartiene. Con ulteriore attenzione possiamo notare la presenza di un'altra parola chiave: `__global__`, che specifica a CUDA di che tipo è la funzione che stiamo dichiarando. Questo implica l'esistenza di altre parole chiave che elencheremo di seguito in una tabella:

| Parola chiave | invocabile da | eseguita su | tipo di ritorno |
|---|---|---|---|
| `__global__` | device e host | device | void |
| `__device__` | device | device | qualsiasi |
| `__host__` | host | host | qualsiasi |

Dalla tabella notiamo che ogni parola chiave distingue la funzione in termini di, dove può essere eseguita e da chi può essere chiamata oltre al tipo di ritorno. Nel caso dell'esempio che abbiamo preso come riferimento, `vector_add` è una funzione *kernel* che chiamiamo da *host* ed eseguiamo su *device* che ha tipo di ritorno void, perché abbiamo dichiarato la funzione come `__global__`.

### 1.3.3 Gestione della memoria in CUDA

Utilizzando CUDA è facile pensare che le aree di memoria che vengono allocate siano accessibili sia da CPU che da GPU. In realtà una GPU ha uno spazio di memoria indipendente da quello di una CPU e condividere uno spazio di memoria porterebbe scarsi risultati in termini di prestazioni. In CUDA occorre copiare i dati da memoria host a memoria device e viceversa.

Questo lo abbiamo visto nell'esempio fatto poco fa, infatti per accedere al contenuto dei vettori `a`, `b` e `out` è stato necessario fare una invocazione a `cudaMalloc(...)` e `cudaMemCpy(...)` per allocare e copiare memoria sul dispositivo.

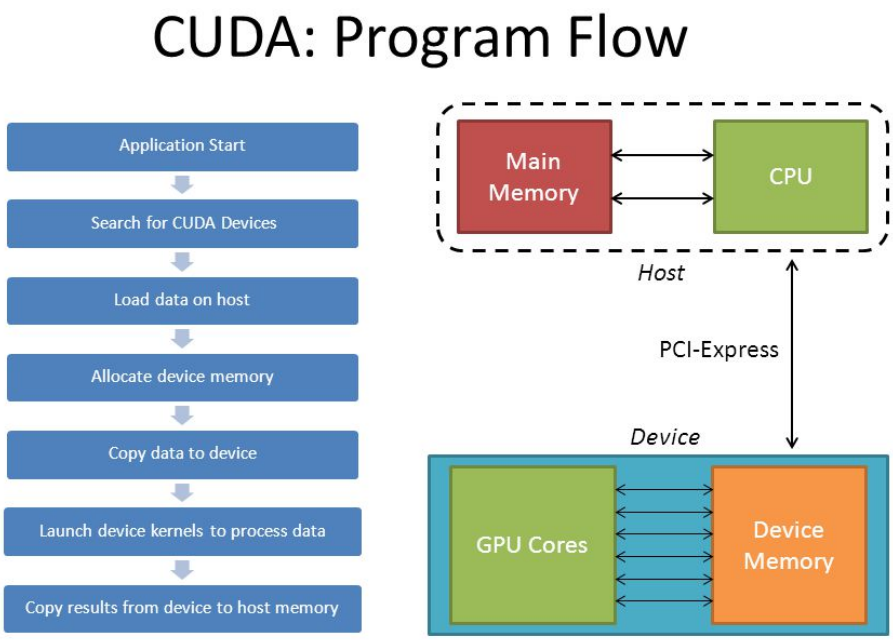


Figura 1.3: Flusso di un programma CUDA

## 1.4 Introduzione a OpenCL

OpenCL (Open Computing Language) è un *framework* per lo sviluppo di applicazioni su piattaforme arbitrarie che possono essere CPU, GPU o FPGA. OpenCL è un progetto avviato da Apple Inc. nel 2008, con collaborazioni di squadre tecniche di aziende quali AMD, IBM, NVIDIA, Intel e Qualcomm. Apple ha in seguito deciso di fare una prima presentazione del prototipo di OpenCL al Khronos Group che porta avanti e mantiene ancora oggi il progetto.

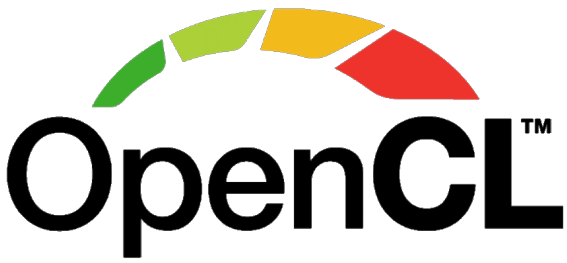


Figura 1.4: Logo OpenCL

Lo standard OpenCL prevede l'utilizzo di un linguaggio basato sul C99, C++14 e C++17 con l'aggiunta di alcune *keyword* strettamente legate alla natura del framework. Utilizzare OpenCL non implica l'utilizzo di alcun linguaggio specifico, infatti è sufficiente avere un binding alla libreria nativa OpenCL per poterne utilizzare le procedure. Questo rende l'utilizzo di OpenCL (a utenti non familiari con linguaggi della famiglia C) più semplice.

Il *Computing Language* rimane sempre sconnesso dal linguaggio *host* dato che il codice *device* viene compilato dalla libreria OpenCL e non da un compilatore esterno che si occupa di compilare sia codice *host* che *device*, come nel caso di CUDA.

### 1.4.1 Pensare in OpenCL

Un host è connesso ad uno o più dispositivi di computazione supportati da OpenCL, dove ognuno di questi comprende delle untià di computazione. Ogni unità di computazione ha al suo interno degli *elementi di elaborazione* che eseguono codice parallelo in modo SIMD (Single Instruction Multiple Data).

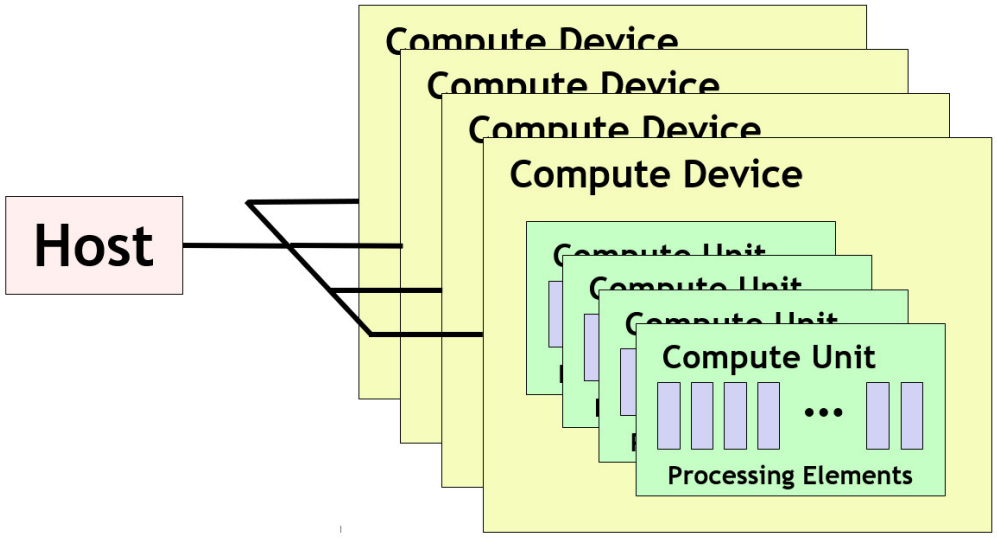


Figura 1.5: Modello della piattaforma OpenCL [3]

OpenCL consente di avviare un programma in parallelo attraverso una singola funzione, detta *kernel*. Un *kernel* opera in una griglia di nodi divisi in gruppi di lavoro. Un gruppo di lavoro condivide risorse e funzioni utili a rendere più efficiente l'interoperabilità tra nodi di uno stesso gruppo, come ad esempio una memoria locale condivisa, funzioni di sincronizzazione o trasferimento dati.

### 1.4.2 Esempio di codice OpenCL

Scrivere codice in OpenCL risulta prolisso e laborioso per la parte di configurazione dell'applicazione, soprattutto se messo a confronto con CU-

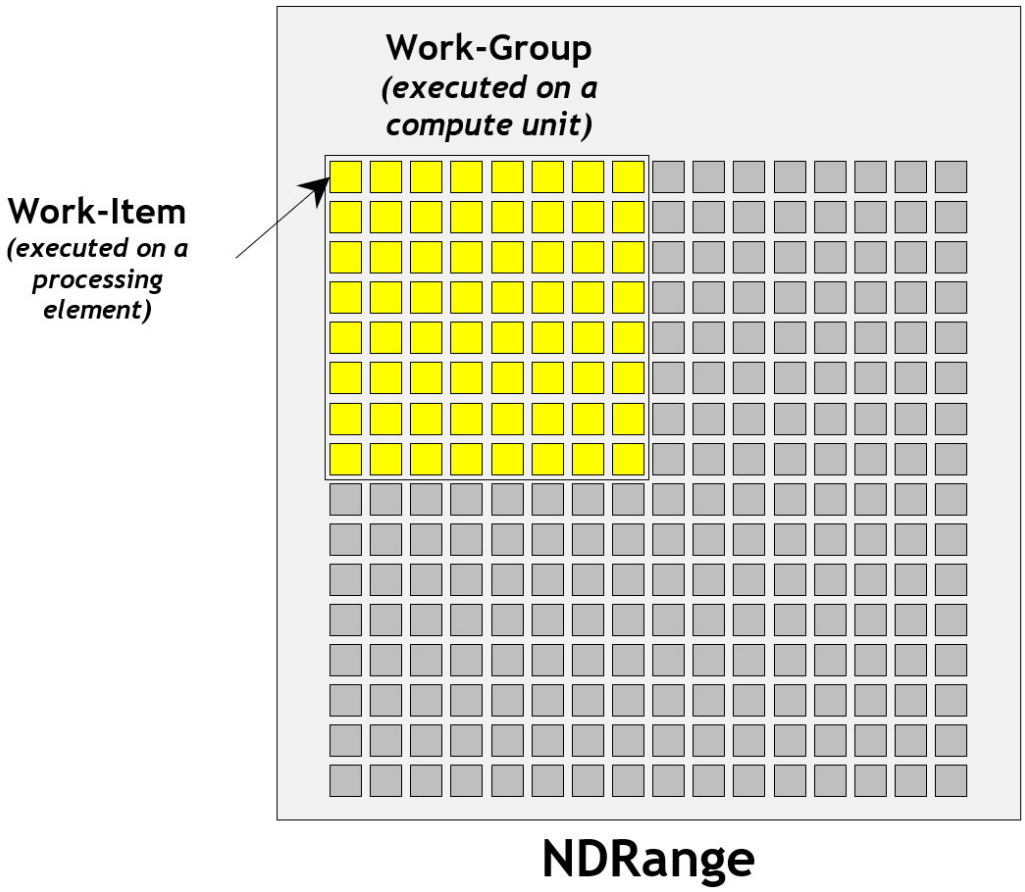


Figura 1.6: Griglia di computazione OpenCL

DA. Partiamo subito con un esempio che dimostra la stessa cosa fatta in precedenza con il *kernel* `vector_add`.

```
#include <string.h>
#include <stdlib.h>
#include "CL/cl.h"

#define N 256

const char* cl_source = "
__kernel void vector_add(__global float *a, __global float *b,
    __global float *out)\
{\
    int i = get_global_id(0);\
    out[i] = a[i] + b[i];\
}";

void main()
{
    const size_t cl_source_len = strlen(cl_source);

    float *a, *b, *out;

    a = (float*)malloc(sizeof(float) * N);
    b = (float*)malloc(sizeof(float) * N);
    out = (float*)malloc(sizeof(float) * N);

    cl_mem d_a, d_b, d_out;

    cl_platform_id platform_id = NULL;
    cl_device_id device_id = NULL;
    cl_uint ret_num_devices;
    cl_uint ret_num_platforms;
```

```
cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 1, &device_id,
    &ret_num_devices);

cl_context context = clCreateContext( NULL, 1, &device_id, NULL,
    NULL,&ret);
cl_command_queue command_queue =
    clCreateCommandQueue(context,device_id,NULL, &ret);

d_a = clCreateBuffer(context, CL_MEM_READ_ONLY, N * sizeof(float),
    NULL, &ret);
d_b = clCreateBuffer(context, CL_MEM_READ_ONLY, N * sizeof(float),
    NULL, &ret);
d_out = clCreateBuffer(context, CL_MEM_READ_WRITE, N *
    sizeof(float), NULL, &ret);

ret = clEnqueueWriteBuffer(command_queue, d_a, CL_TRUE, 0, N *
    sizeof(float), a, 0, NULL, NULL);
ret = clEnqueueWriteBuffer(command_queue, d_b, CL_TRUE, 0, N *
    sizeof(float), b, 0, NULL, NULL);

cl_program program = clCreateProgramWithSource(context, 1,
    &cl_source, &cl_source_len, &ret);
clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

cl_kernel kernel = clCreateKernel(program, "vector_add", &ret);

clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&d_a);
clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&d_b);
clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&d_out);

size_t global_item_size = N;
size_t local_item_size = 1;
```

```
    cl_event event;
    clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
        &global_item_size, &local_item_size, 0, NULL, NULL);
    clFinish(command_queue);
    clEnqueueReadBuffer(command_queue, d_out, CL_TRUE, 0, N *
        sizeof(float), out, 0, NULL, NULL);

    clFlush(command_queue);
    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseMemObject(d_a);
    clReleaseMemObject(d_b);
    clReleaseMemObject(d_out);
    clReleaseCommandQueue(command_queue);
    clReleaseContext(context);
    free(a);
    free(b);
    free(out);
}
```

Nonostante siano stati trascurati la gestione degli errori e la visualizzazione dei risultati, il codice risulta verboso. Come in CUDA vengono gestiti i trasferimenti di memoria tra *host* e *device* olte che alle allocazione di memoria GPU. Una differenza sostanziale la vediamo in `clCreateProgramWithSources(...)` e `clBUildProgram(...)`, infatti il compilatore per il *Computing Language* si trova all'interno della libreria, perciò dobbiamo invocarne l'esecuzione da codice passandogli i sorgenti che vogliamo compilare.

Per quanto riguarda `cl_source`, ci sono alcune osservazioni da fare. Vediamo inanzitutto una parola chiave `__kernel` che serve a notificare OpenCL che la funzione è invocabile da host ed eseguibile su device (per intenderci, un equivalente a `__global__` in CUDA). Esistono anche le cosiddette funzioni

ausiliare, che possono solo essere invocate ed eseguite su device. Notiamo inoltre una keyword `__global` che precede il tipo dei parametri in ingresso al *kernel*. Serve a definire uno spazio di memoria per i buffer, OpenCL ha bisogno di sapere dove sono collocati. Ci sono diversi tipi di identificatori per gli spazi di memoria, il numero può variare in base al tipo di dispositivo che si ha sottomano. In linea generale ci sono sempre almeno quattro identificatori standard. Riassumiamoli quindi in una tabella.

| ID | visibilità | condivisa con | R/W |
|---|---|---|---|
| `__private` | un thread | solo se stesso | R/W |
| `__local` | thread di un gruppo | gruppo | R/W |
| `__global` | tutti i thread | tutti i thread | R/W |
| `__constant` | tutti i thread | tutti i thread | R |

Con la keyword `__constant` è importante sapere che è esclusivamente compito dell'host aver popolato dati in quell'area di memoria, cosa che deve essere fatta prima dell'esecuzione di un *kernel*. La keyword `__private` invece identifica memoria molto veloce, privata al contesto di un nodo, possiamo quindi pensare a registri di memoria per valori di grandezza relativamente piccola o a memorie *cache* in caso contrario, spesso più veloce di memoria di tipo `__local`. Tuttavia questi accorgimenti sono poco affidabili, infatti molte casi in cui un tipo di memoria diventa più lento o più veloce nell'utilizzo, dipende strettamente dal dispositivo che si ha sotto mano. OpenCL fornisce comunque un modo di utilizzare queste funzionalità quando possibile. Tornando su `__local` questa in molti casi garantisce memoria più veloce di quella identificata come `__global`, è consigliata in casi di lettura ridondante in specifiche aree di memoria globale. Architetturalmente occupa spazi di memoria di prestazioni simili a quelle di una *cache* di livello 3. In finale con `__global` e `__constant` andiamo a utilizzare memoria di tipo RAM e VRAM, viene spesso utilizzata per dati di grandi dimensioni quindi immagini, matrici o campioni audio [1].

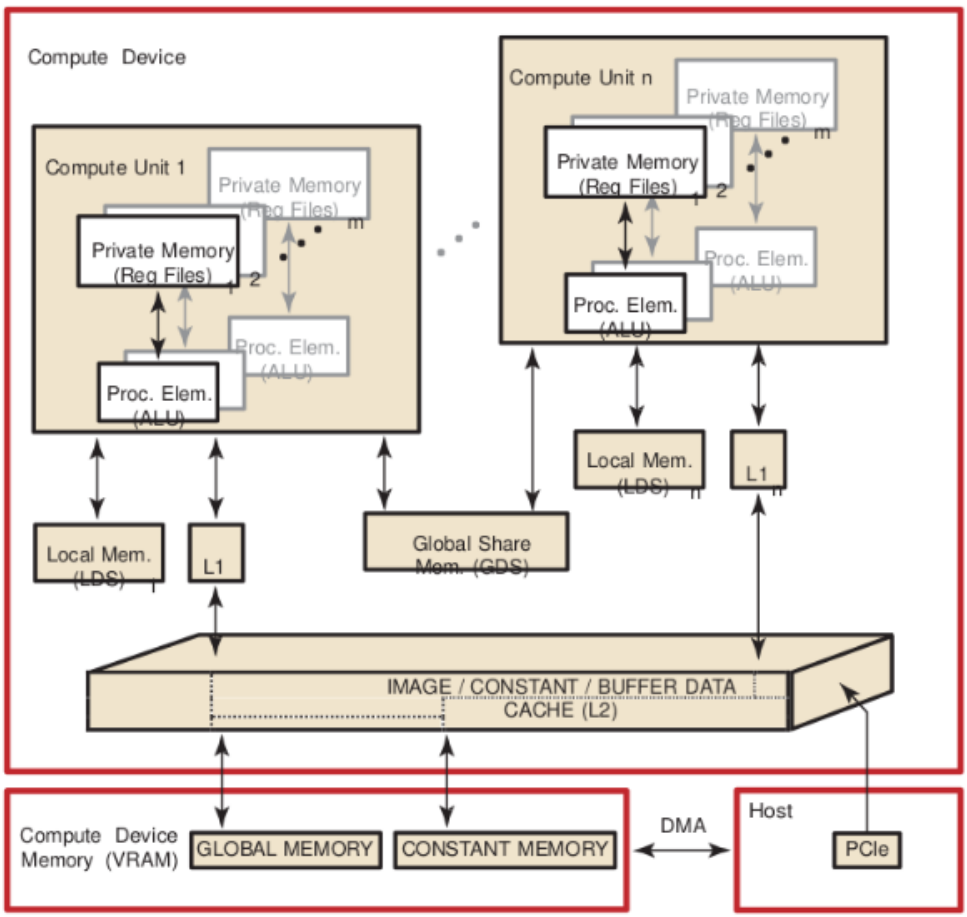


Figura 1.7: Memoria GPU secondo OpenCL

Le prestazioni di memoria globale dipendono strettamente dal corretto utilizzo di questa, bisogna utilizzare in maniera intelligente la *cache* e la prossimità dei dati in memoria. Dove possibile è consigliato utilizzare memoria locale per aumentare le prestazioni.

# Capitolo 2

# Introduzione a SimpleCL

Abbiamo visto con un esempio come il codice OpenCL possa risultare molto verboso anche per problemi semplici. Nonostante questo le due tecnologie hanno sintassi divergenti, una più semplice e immediata, l'altra con una curva di apprendimento leggermente più ripida. Sarebbe ottimo avere un compromesso tra le due o quantomeno un modo di velocizzare i primi approcci ad OpenCL.

SimpleCL nasce proprio per questo, rendere la programmazione OpenCL simile alla programmazione CUDA mantenendosi fedele ai vincoli imposti dal framework. SimpleCL è un *wrapper* sviluppato e mantenuto dal Prof. Moreno Marzolla. Procedure comuni di configurazione o trasferimento dati vengono "impacchettate" in chiamate simili a quelle che uno sviluppatore CUDA si aspetterebbe.

Introdurremo in questo capitoli alcuni spunti di codice scritto con SimpleCL, vedremo come interagire con la GPU e metteremo a confronto li tutto con l'equivalente CUDA.

## 2.1 Esempio di codice SimpleCL

SimpleCL si presenta snello e semplice con una interfaccia che ricorda in modo positivo l'API CUDA. Vediamo ora un esempio di codice che si rifà agli esempi fatti in precedenza, la somma di due vettori di stessa dimensione.

```
#include "simpleCL.h"
#include "stdlib.h"
#include "string.h"

#define N 256

const char* cl_source = "__kernel void vector_add(__global float *a,
    __global float *b, __global float *out) {\
    int i = get_global_id(0);\
    out[i] = a[i] + b[i];\
}";

int main(void)
{
    float* a, *b, *out;
    cl_mem d_a, d_b, d_out;

    sclInitFromString(cl_source);

    a = (float*)malloc(sizeof(float) * N);
    b = (float*)malloc(sizeof(float) * N);
    out = (float*)malloc(sizeof(float) * N);
    d_a = sclMallocCopy(sizeof(float) * N, a, CL_MEM_READ_ONLY);
    d_b = sclMallocCopy(sizeof(float) * N, b, CL_MEM_READ_ONLY);
    d_out = sclMalloc(sizeof(float) * N, CL_MEM_WRITE_ONLY);

    sclDim global_work_size = DIM1(N);
```

```
    sclDim local_work_size = DIM1(1);

    sclKernel kernel = sclCreateKernel("vector_add");

    sclSetArgsEnqueueLernel(kernel,
        global_work_size,
        local_work_size,
        ":b :b :b",
        d_a, d_b, d_out);

    sclMemcpyDeviceToHost(out, d_out, sizeof(float) * N);

    free(a); free(b); free(out);
    sclFree(d_a); sclFree(d_b); sclFree(d_out);
    return 0;
}
```

Analizziamo il sorgente e capiamo cosa sta succedendo. In cima al file troviamo le dichiarazione dei buffer che verrà utilizzato per effettuare la somma (notare come i buffer che risiedono nella memoria del device rimangono di tipo `cl_mem`). Successivamente andiamo a riservare della memoria per i buffer, SimpleCL permette di utilizzare una funzione di allocazione e copia tutta in una chiamata `sclMallocCopy(...)` che si occuperà dietro le quinte di creare buffer, controllare errori e copiare i dati dal buffer che viene passato come parametro.

Prima di fare questo è importante effettuare una chiamata a `sclInitFromString(...)` o `sclInitFromFile(...)` con il sorgente che contiene la funzione *cl* che vogliamo invocare. Questo perché SimpleCL ha bisogno di impostare il suo ambiente prima che qualsiasi operazione possa avvenire. Andando oltre, passiamo alla creazione del *kernel*, specificando a `sclCreateKernel(...)` il nome della funzione che andremo a chiamare. Non resta che effettuare la

chiamata, o meglio, mettere la funzione in coda per essere eseguita.

Questo passo richiede di dare una serie di valori a `sclSetArgsEnqueueKernel(...)` quali la dimensione della griglia, la dimensione di ogni gruppo di lavoro e i tipi dei valori che passeremo codificati in una stringa, il resto dei parametri sono i parametri che OpenCL passerà alla funzione che abbiamo selezionato come *kernel*.

## 2.2 SimpleCL e CUDA a confronto

Sappiamo per definizione che codice scritto con SimpleCL ricorda al programmatore codice CUDA. Nonostante sia già ben visibile nella parte introduttiva di questo capitolo, è bene fare qualche appunto sulle somiglianze e dettagli della libreria.

Negli esempi a seguire considereremo delle sezioni di codice comuni in CUDA e ne vedremo la corrispettiva versione SimpleCL, faremo uso di funzioni immaginarie (ad esempio `init(...)`) per dare un minimo di coerenza al codice.

### 2.2.1 Creazione, allocazione e copia di un buffer device

Consideriamo ora la creazione, allocazione e copia di un buffer in memoria device.

```
// CUDA
void main()
{
    int* some_buffer = (int*)malloc(sizeof(int) * N);
    int* d_some_buffer;
    cudaMalloc((void**)d_some_buffer, sizeof(int) * N);
    cudaMemcpy(d_some_buffer, some_buffer, sizeof(int) * N,
        cudaMemcpyHostToDevice);
}
```

```
// SimpleCL mappando 1:1
void main()
{
    int* some_buffer = (int*)malloc(sizeof(int) * N);
    init(some_buffer);
    cl_mem d_some_buffer;
    d_some_buffer = sclMalloc(sizeof(int) * N, CL_MEM_READ_ONLY);
    sclMemcpyHostToDevice(d_some_buffer, some_buffer, sizeof(int) * N);
}
```

```
// SimpleCL in forma breve
void main()
{
    int* some_buffer = (int*)malloc(sizeof(int) * N);
    init(some_buffer);
    cl_mem d_some_buffer = sclMallocCopy(sizeof(int) * N, some_buffer,
        CL_MEM_READ_ONLY);
}
```

È scontato dire che SimpleCL fa un ottimo lavoro nell'imitare la sintassi CUDA, riesce anche a fornire una sintassi più snella tramite funzioni che incorporano chiamate spesso fatte insieme, questo è il caso di `sclMallocCopy(...)`. Una sottile differenza la vediamo in come la libreria restituisce i puntatori alla memoria device (in realtà in OpenCL sono numeri, identificatori di memoria e non puntatori), infatti vengono forniti sotto forma di valore restituito da funzione. In CUDA per operazioni tipo `cudaMalloc(...)` viene richiesto di passare un puntatore a puntatore, dove verrà salvato l'indirizzo della memoria GPU che stiamo allocando.

### 2.2.2 Chiamata di una funzione kernel

Creare ed invocare un *kernel* presenta alcune differenze che, come intuibile dal capitolo precedente, derivano dalla differenza in cui OpenCL e CUDA decidono di astrarre le unità di lavoro. Vediamo un semplice esempio.

```
// CUDA
void main()
{
    my_kernel<<<N_BLOCKS, BLOCK_DIM>>>(...);
}
```

```
// SimpleCL
void main()
{
    sclKernel kernel = sclCreatekernel("my_kernel");
    sclSetArgsEnqueueKernel(kernel,
        global_work_size,
        local_work_size,
        <param types here>,
        ...);
}
```

Notiamo subito una differenza particolare tra i due metodi, SimpleCL richiede la specifica di una grandezza globale per la griglia di lavoro, CUDA richiede un numero di blocchi. L'approccio SimpleCL (in realtà OpenCL) è più sicuro, in particolar modo nell'utilizzo di identificatori globali per un nodo lavoratore. Per ottenere un identificatore globale in un kernel usando CUDA si può fare nel seguente modo:

```
unsigned long int threadId = blockIdx.x * blockDim.x + threadIdx.x;
```

Questa è una operazione pericolosa, si rischia di andare fuori dai limiti del nostro insieme di dati, causando errori. Il programmatore è forzato a fare

controllo di errori prima di utilizzare un dato identificatore. In questi casi si fa uso di apposite trasformazioni di arrotondamento per il numero di blocchi e thread per blocco.

Tuttavia OpenCL propone la strada alternativa: si specifica direttamente una grandezza per la griglia di lavoro, sarà poi il framework attraverso `get_global_id(0)` a fornire l'identificatore globale per il nostro nodo di lavoro. Il programmatore non deve fare alcun controllo, a patto che le dimensioni fornite al *kernel* siano conformi con i buffer che si stanno processando. Visualizziamo ora quanto detto attraverso una immagine.

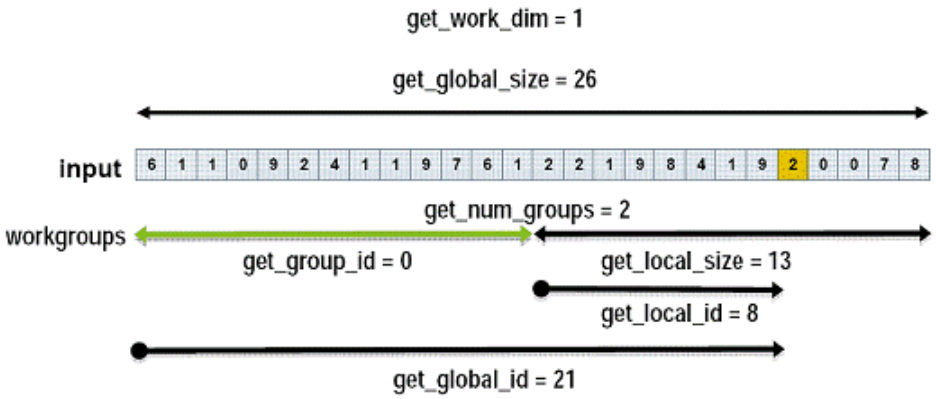


Figura 2.1: Sistema di identificatori OpenCL

### 2.2.3 Riassumendo

Oltre agli esempi fatti, non ci sono particolari differenze a livello di sintassi per applicazioni relativamente semplici. Uno sviluppatore CUDA può tranquillamente iniziare a scrivere codice con SimpleCL praticamente da subito, sicuramente si perdono alcuni vantaggi dell'utilizzare OpenCL puro, cioè il poter personalizzare al meglio l'esperienza d'uso. In tal caso probabilmente SimpleCL non è consigliato, ma non dovrebbe nemmeno dato che non è stato pensato per questo.

# Capitolo 3

# Passaggio da CUDA a SimpleCL di un simulatore cardiaco

In questo capitolo verrà descritto e analizzato un simulatore cardiaco dal punto di vista computazionale. Il simulatore prende il nome di *2D SAN Atrium*, è stato sviluppato da Stefano Severi e Eugenio Ricci in collaborazione con Moreno Marzolla. Il simulatore ha visto diverse versioni tra cui una MATLAB, C++11 con OpenMP e successivamente una CUDA, rinominandolo a *2D Parallel SAN Atrium*. Verrà presentata una versione OpenCL derivata dalla versione CUDA tramite SimpleCL, vedremo in particolare gli accorgimenti che sono stati fatti a questo fine.

## 3.1 Introduzione all'utilizzo del simulatore

*2D SAN Atrium* si presenta sotto forma di una serie di directory, ognuna contenente le versioni precedentemente citate e alcune directory dedicate a file contenenti parametri in input al programma. Ogni versione include un `Makefile` con alcune direttive per compilare il programma ed eseguire dei

test. Per semplicità e chiarezza riportiamo di seguito una rappresentazione testuale del *directory tree* del simulatore.

```
2D_parallel_SAN_Atrium
├── OpenMP/
├── CUDA/
├── Init_cond/
├── OpenCL
│   ├── CLSources/
│   ├── Makefile
│   └── source files...
└── m_scripts/
    └── Sim_Param_folder/
```

Per eseguire il programma è sufficiente avere a disposizione lo strumento `make` nel proprio sistema ed eseguire il comando `make test`. Il comando compilerà ed eseguirà il programma in automatico, utilizzando dei parametri preimpostati. Visto come compilare ed eseguire il programma è bene fare un passo indietro e chiarire alcuni aspetti. Nella prossima sezione vedremo come raggiungere l'obbiettivo che ci siamo prefissati ad inizio capitolo, ovvero convertire un simulatore cardiaco da CUDA a OpenCL. Facendolo vedremo come creare un `Makefile` adatto e convertire chiamate e *kernel* CUDA ad OpenCL.

## 3.2 Conversione da CUDA a SimpleCL

### 3.2.1 Creare un Makefile per OpenCL

Creare un `Makefile` per OpenCL diventa un'operazione relativamente semplice, per renderci le cose ancora più semplici possiamo fissare come punto di partenza la versione CUDA e dal lì fare le giuste modifiche.

Consideriamo quindi il Makefile CUDA.

---

```
EXE:=2D_parallel_SAN_atrium
SRC:=$(wildcard *.cpp) $(wildcard *.cu)
INC:=$(wildcard *.hpp)
OBJ:=${SRC:%.cpp=%.o}
OBJ:=${OBJ:%.cu=%.o}
NVCC:=nvcc
NVCFLAGS+=-DNO_CUDA_CHECK_ERROR

## compiler flags
CXXFLAGS += -std=c++11 -Wall -Wpedantic
LDLIBS += -L/usr/local/cuda/lib64/ -lm -lcudart

.PHONY: test clean

$(EXE): $(OBJ)
   $(CXX) $(LDFLAGS) $^ $(LDLIBS) -o $@

%.o : %.cu
   $(NVCC) $(NVCFLAGS) -c $<

test: $(EXE)
   cd ../m_scripts/Sim_Param_folder/ && ../../CUDA/$(EXE)

clean:
```

```
   \rm -f $(OBJ) $(EXE) *~ gmon.out

distclean: clean
```

Come potevamo aspettarci vediamo un compilatore diverso dal solito `g++` o `clang++`, in questo caso viene tirato in ballo `nvcc` il compilatore per file sorgenti `.cu`, i file utilizzati dalla piattaforma CUDA. Il compilatore `nvcc` è un normalissimo compilatore C++ con parecchie modifiche per soddisfare i bisogni della piattaforma. Tuttavia in OpenCL sicuramente non ne avremo bisogno e possiamo tranquillamente farne a meno. Oltre a questo, notiamo delle regole `make` per associare file `.cu` a file oggetto `.o`, output del compilatore, anche questi non serviranno. Ovviamente, anche le librerie che vengono passate al `linker` non serviranno, infatti le andremo a sostituire con le librerie OpenCL.

Fatti questi accorgimenti, direi che siamo più che pronti per introdurre un `Makefile` OpenCL e questo è quello che otteniamo.

```
EXE:=2D_parallel_SAN_atrium
SRC:=$(wildcard *.cpp *.c)
INC:=$(wildcard *.hpp)
OBJ:=${SRC:%.cpp%.c=%.o}

## compiler flags
CXXFLAGS += -std=c++17 -Wall -Wpedantic
CLLIBS += -lOpenCL

.PHONY: test clean

$(EXE): $(OBJ)
   $(CXX) $(CXXFLAGS) $(LDFLAGS) $^ $(CLLIBS) -o $@

test: $(EXE)
```

```
   #cd ../m_scripts/Sim_Param_folder/ && ../../OpenCL/$(EXE)
   ./$(EXE) ../m_scripts/Sim_Param_folder/

clean:
   rm -f *.o $(EXE)

distclean: clean
```

Oltre ad avere cambiato lo standard C++ per supportare funzionalità più moderne, la wildcard per la variabile `SRC` è stata cambiata; allo scopo di includere file scritti in linguaggio C, proprio il linguaggio utilizzato dal *wrapper* SimpleCL. Da notare che, essendo il C++ un sovrainsieme del linguaggio C, un tradizionale compilatore C++ non dovrebbe avere alcun problema a processare file C, ad esclusione di alcuni casi estremi.

### 3.2.2 Sostituire chiamate CUDA con chiamate SimpleCL

Convertire codice CUDA a codice SimpleCL abbiamo visto essere piuttosto immediato. A questo punto basta prendere ogni file `.cu`, rinominarlo a `.cpp` e convertire le chiamiate CUDA a chiamate SimpleCL. Per quanto riguarda le funzioni device, è consigliato spostarle in appositi file `.cl` che andremo a leggere una volta che avremo bisogno di invocare tali funzioni. Procediamo quindi con i primi step illustrando alcune delle procedure che sono state tradotte nel simulatore. Il file su cui vogliamo concentrarci al momento si chiama `euler_solve.cu`, questo nella pratica contiene il punto di entra delle funzioni device, infatti contiene la sola funzione in tutto il programma che esegue l'invocazione di un *kernel* CUDA.

Il *kernel* (di nome `f_sys`), è una funzione piuttosto complessa, che richiede molti parametri e che viene invocata più volte nel corso della simulazione. Riportiamo di seguito la sezione di codice su cui interessa agire, tagliando gran parte del codice ridondante.

```
// euler_solve.cu
void euler_solve(...)
{
    // ... Inizializzazione buffer host
    double *X_arr = new double[X_ROWS * X_COLS];
    double *X_n_arr = new double[X_n_ROWS * X_n_COLS];
    ...

    // ... Creazione dei corrispettivi buffer device
    double *d_X_arr;
    const size_t d_X_arr_SIZE = X_ROWS * X_COLS * sizeof(*d_X_arr);
    double *d_X_n_arr;
    const size_t d_X_n_arr_SIZE = X_n_ROWS * X_n_COLS *
        sizeof(*d_X_n_arr);
    ...

    // ... Allocazione e copia dei buffer
    cudaSafeCall( cudaMalloc( (void**)&d_X_arr, d_X_arr_SIZE ) );
    cudaSafeCall( cudaMemcpy(d_X_arr, X_arr, d_X_arr_SIZE,
        cudaMemcpyHostToDevice) );
    ...

    const dim3 BLOCK(BLKDIM);
    const dim3 GRID((nCells + BLKDIM - 1)/BLKDIM);

    for (int steps = 0; steps < sim_steps; steps++) {
        ...
        f_sys<<< GRID, BLOCK >>>(...);
        ...
    }

}
```

Questo è tutto il codice CUDA presente lato host, l'unica cosa che dobbiamo fare è applicare i concetti che abbiamo visto nelle sezione dell'introduzione a SimpleCL, in particolare l'allocazione di buffer e le chiamate a *kernel*. Applichiamo quanto detto e procediamo come segue.

```
// euler_solve.cpp
void euler_solve(...)
{
    // ... Inizializzazione buffer host (rimane invariata)
    double *X_arr = new double[X_ROWS * X_COLS];
    double *X_n_arr = new double[X_n_ROWS * X_n_COLS];
    ...

    // ... Id di memoria OpenCL
    cl_mem d_X_arr;
    cl_mem d_X_n_arr;
    ...

    // ... Allocazione e copia dei buffer
    d_X_arr       = sclMallocCopy(d_X_arr_SIZE,    X_arr,
                          CL_MEM_READ_ONLY);
    d_gJ_arr      = sclMallocCopy(d_gJ_arr_SIZE,   gJ_arr,
                         CL_MEM_READ_ONLY);
    ...

    sclKernel kernel = sclCreateKernel("f_sys");

    size_t gws_component = sclRoundUp((size_t)pow(nCells, 1/3),
        SCL_DEFAULT_WG_SIZE3D);
    sclDim global_work_size = DIM3(gws_component, gws_component,
        gws_component);
    sclDim local_work_size = DIM3(SCL_DEFAULT_WG_SIZE3D,
```

```
        SCL_DEFAULT_WG_SIZE3D, SCL_DEFAULT_WG_SIZE3D);

    for (int steps = 0; steps < sim_steps; steps++) {
        ...
        sclSetArgsEnqueueKernel(kernel,
            global_work_size,
            local_work_size,
            ":d :b :b ...",
            nCells, d_X_arr, d_X_n_arr, ...
            );
        ...
    }
}
```

Come anticipato più volte, grazie a SimpleCL la conversione è stata una sciocchezza. Tuttavia non abbiamo ancora visto come gestire la conversione di *kernel* e funzioni device CUDA a OpenCL. Nella prossima sezione tratteremo proprio questo.

### 3.2.3 Convertire kernel CUDA in kernel OpenCL

Convertire un *kernel* CUDA ad uno OpenCL spesso significa semplicemente cambiare le keyword usate in CUDA con quelle usate in OpenCL. Altre volte ci si ritrova con problemi più seri, ad esempio garantire la precisione dei calcoli, dato che alcune piattaforme potrebbero non supportare certi tipi di dato. Tali problemi si presentano spesso con i tipi di dato a virgola mobile, infatti il tipo di dato `double` non è supportato da tutte le GPU o CPU ancora oggi in commercio. Ovviare a questi problemi non è di grandissima difficoltà, basta limitarsi ad usare dati `float` se la precisione non è importante, oppure, gestire l'uso di `double` o `float` con macro a compile-time. Fortunatamente, a noi interessa principalmente lavorare sul server `isi-raptor03` dotato di tre GPU NVIDIA GTX 1070, che supportano il tipo di dato `double` per i calcoli.

Prendiamo ora in considerazione il *kernel* CUDA `f_sys` e vediamo quali sono i cambiamenti necessari a renderlo un *kernel* OpenCL valido.

```
// f_sys.cu
__global__
void f_sys(size_t nCells,
          const double *Ymat,
          double *Y_n_mat,
          size_t Ymat_COLS,
          ...
{
        const size_t j = threadIdx.x + blockIdx.x * blockDim.x;

        ...

        const double *Y = Ymat + j*Ymat_COLS;
        double *dY = dYmat + j*dYmat_COLS;
        double *Y_n = Y_n_mat + j*Ymat_COLS;
        const double *rand_g = rand_g_mat + j*rand_g_COLS;

        ...
}
```

Ricordando le osservazioni fatte nel capitolo di introduzione alle piattaforme di computazione parallela, siamo in grado di dire che la keyword `__global__`, essendo una funzione *kernel* invocabile da host ed eseguibile su device, può diventare l'equivalente OpenCL `__kernel`. Oltre a questo, ci sono degli accorgimenti da fare riguardo alla specifica dello spazio di memoria per i buffer che trattiamo nel *kernel*. OpenCL vuole che ogni buffer processato abbia un particolare spazio di memoria, che se non specificato viene impostato a `__private`. Per quello che sappiamo tutti i buffer che andremo

a processare nei *kernel* dovranno stare in spazio `__global`. Oltre a questo possiamo sfruttare una comodità di OpenCL per rendere il codice un pò più espressivo, questo tramite l'utilizzo di `get_global_id(0)`. Infatti nella prima riga di codice del nostro *kernel* stiamo cercando di calcolare l'id globale del thread corrente sull'asse X, la prima dimensione.

Riportiamo quindi qui sotto, quale sarebbe l'equivalente OpenCL del *kernel* appena visto.

```
// f_sys.cl
__kernel
void f_sys(size_t nCells,
          const __global double *Ymat,
          __global double * Y_n_mat,
          size_t Ymat_COLS,
          ...
{
        const size_t j = get_global_id(0);

        ...

        const __global double *Y = Ymat + j*Ymat_COLS;
        __global double *dY = dYmat + j*dYmat_COLS;
        __global double *Y_n = Y_n_mat + j*Ymat_COLS;
        const __global double *rand_g = rand_g_mat + j*rand_g_COLS;

        ...
}
```

## 3.3 Analisi prestazioni del simulatore

### 3.3.1 Metodi per le stime di prestazioni

Per stimare le prestazioni delle due versioni analizzate del simulatore utilizzeremo la quantità dello *speedup*. Lo *speedup* è un numero che quantifica le prestazioni relative di due sistemi computazionali. In questo caso abbiamo applicato una semplice conversione di piattaforma software. Tuttavia è interessante verificare la presenza di perdite a livello di prestazioni ed eventualmente discuterne la causa.

Lo *speedup* è un numero che deriva da un rapporto tra tempi. Nel rapporto sono presenti rispettivamente, tempo di esecuzione del programma seriale e tempo di esecuzione del programma parallelo.

Formalmente, possiamo definire lo *speedup* come:

$$S(n) = \frac{T_1}{T(n)}$$

Dove $n$ è il numero di nodi utilizzati per la versione parallela, $T_1$ rappresenta il tempo impiegato dalla versione del programma con un solo nodo e $T(n)$ il tempo impiegato dalla versione parallela con $n$ nodi.

### 3.3.2 Impostare l'ambiente

Per analizzare le prestazioni del simulatore faremo uso del server `isi-raptor03` disponibile presso l'università di Bologna. Il server mette a disposizione tre schede video NVIDIA GTX 1070 con 8GB di VRAM ciascuna. Noi ne utilizzeremo solo una, dato che il simulatore non è ancora stato pensato per l'esecuzione su più GPU.

Per la versione CUDA, sarà la piattaforma a scegliere in automatico una delle schede grafiche disponibili. Nel caso di SimpleCL, se non viene specificato altrimenti il sistema sceglie il primo dispositivo compatibile, che potrebbe risultare la CPU anziché la GPU. Fortunatamente SimpleCL mette a

disposizione una variabile d'ambiente `SCL_DEFAULT_DEVICE`, che permette di selezionare il numero del dispositivo nel sistema che vogliamo utilizzare. Nel nostro caso eseguiremo il comando `export SCL_DEFAULT_DEVICE=1` per selezionare una delle GPU del server. Fatti questi accorgimenti tecnici possiamo passare all'analisi delle prestazioni.

### 3.3.3 Tempi di esecuzione e analisi prestazioni

Il simulatore ha visto diversi cambiamenti nel corso dello sviluppo tra una versione e l'altra. I dati che andremo a visualizzare, seppur poco coerenti con i cambiamenti discussi fino ad ora, indicano uno *speedup* notevole tra versione seriale e parallelizzata. Si può notare inoltre, una differenza sostanziosa tra versione CUDA e OpenCL, questo può sembrare strano dato che ci siamo solo dedicati ad effettuare una traduzione del software da una piattaforma all'altra.

Prenderemo come riferimento una versione seriale del programma, in particolare la versione OpenMP con un solo thread attivo. Di seguito riportiamo alcune tabelle con i rispettivi tempi medi su cinque esecuzioni per ogni versione. Prenderemo in considerazioni quattro griglie di dati di dimensioni incrementali, queste sono: 51x51, 71x71, 101x101 e 201x201. Riportiamo quindi i tempi.

| Versione | 51x51 | 71x71 | 101x101 | 201x201 |
|---|---|---|---|---|
| CUDA | 0,6362s | 0,9557s | 1,5775s | 5,4448s |
| OpenCL | 0,0972s | 0,1572s | 0,2769s | 0,9831s |
| OpenMP (1 thread) | 114,6320s | 216,1071s | 442,4563s | 1852,1758s |

Date le medie dei tempi di esecuzione vogliamo (per versioni parallele e versione seriale) trovare lo *speedup* ottenuto. Abbiamo già visto come farlo precedentemente, non resta che applicare i calcoli e ottenere il valore per ogni dimensione di griglia. Notiamo quindi uno *speedup* notevole, le versioni parallele migliorano molto le prestazioni, soprattutto la versione OpenCL.

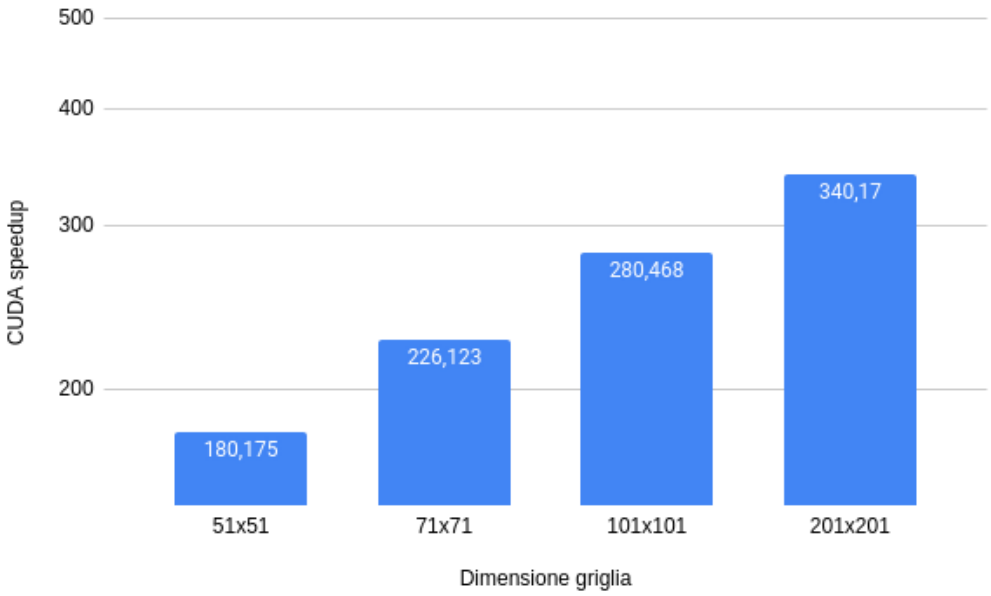


Figura 3.1: speedup CUDA vs OpenMP (1 thread)

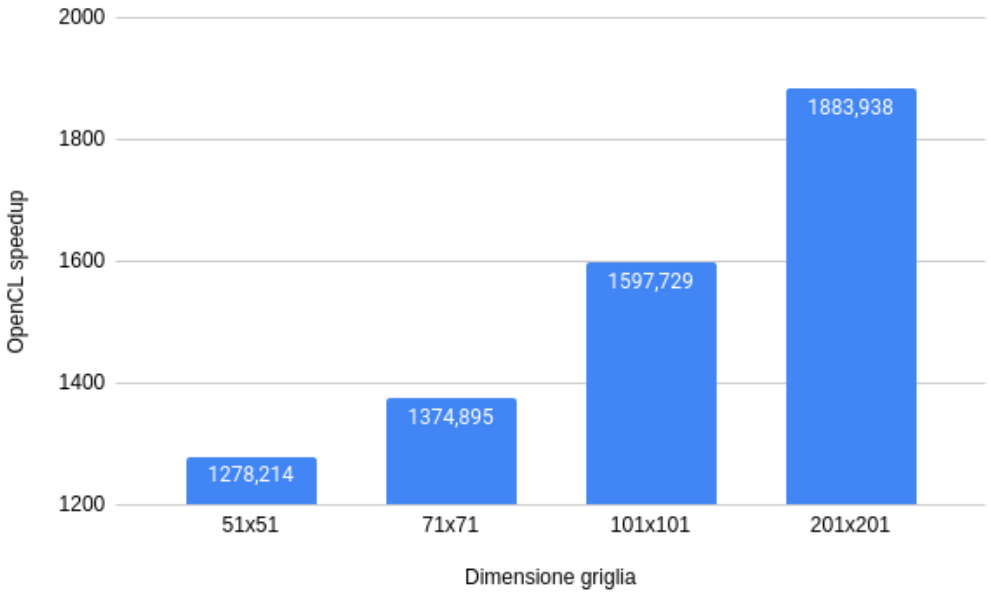


Figura 3.2: speedup OpenCL vs OpenMP (1 thread)

# Conclusioni e sviluppi futuri

In qeusta tesi è stata descritta la conversione di un programma CUDA in OpenCL usando la libreria SimpleCL. Abbiamo visto come possa essere semplice ed immediata la conversione di codice CUDA in codice OpenCL. Siamo stati in grado di applicare tali nozioni ad un simulatore cardiaco, il quale forniva in anticipo una versione CUDA e OpenMP.

Nonostante i buoni risultati raggiunti mediante la nuova versione del programma, c'è ancora molto spazio per applicare miglioramenti. Il programma non fa utilizzo di alcun tipo di memoria locale ad esempio. In programmi paralleli il vincolo più grande, una volta raggiunto un alto numero di *worker* paralleli, diventa il traffico di memoria. Leggere e scrivere memoria con molta frequenza nei posti sbagliati causa perdita di prestazioni che potrebbe essere evitata semplicemente partizionando l'insieme di dati in maniera più efficace e utilizzando le giuste aree di memoria. Sarebbe interessante inoltre per il futuro del simulatore, vedere in opera una versione *multi-GPU* abilitando il programma a processare griglie di dimensioni maggiori alle correnti.

# Bibliografia


[1] Intel Corporation. Optimizing opencl usage with intel processor graphics.

[2] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan 1998.

[3] Khronos Group. Opencl programming model, https://www.khronos.org/assets/uploads/developers/library/2012-pan-pacific-road-show-june/opencl-details-taiwan_june-2012.pdf.

[4] Khronos Group. The opencl specification, https://registry.khronos.org/opencl/specs/opencl-1.2.pdf.

[5] AMD Inc. Opencl programming guide, https://rocmdocs.amd.com/en/latest/programming_guides/opencl-programming-guide.html.

[6] NVIDIA Inc. Cuda c programming guide, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[7] David Kirk. Nvidia cuda software and gpu parallel computing architecture. volume 7, pages 103–104, 01 2007.