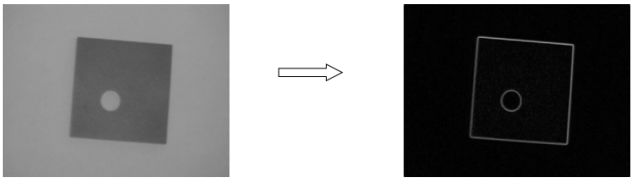
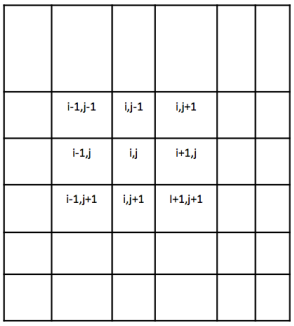
Esercizi

Domanda 1. Il riconoscimento dei contorni (edge detection) è utilizzato allo scopo di marcare i punti di un'immagine digitale in cui l'intensità luminosa cambia bruscamente, ed è uno strumento fondamentale in visione artificiale. Un esempio è mostrato di seguito:



Per calcolare l'intensità del pixel (i, j) si usa frequentemente l'operatore di Sobel definito dalle seguenti equazioni, dove $s(i, j)$ denota l'intensità del pixel nell'immagine originale e $d(i, j)$ l'intensità nella nuova immagine (ovvero quella ottenuta dopo aver effettuato il riconoscimento dei contorni):

$$
\begin{aligned}
G_x = & -s(i-1,j-1) - 2s(i-1,j) - s(i-1,j+1) \\
& +s(i+1,j-1) + 2s(i+1,j) + s(i+1,j+1) \ , \\
G_y = & -s(i-1,j-1) - 2s(i,j-1) - s(i+1,j-1) \\
& +s(i-1,j+1) + 2s(i,j+1) + s(i+1,j+1) \ , \\
& d(i,j) = \sqrt{{G_x}^2 + {G_y}^2} \quad .
\end{aligned}
$$



Il vicinato del pixel (i, j) consiste in generale di 8 pixel, come mostrato nella matrice. Per trattare pixel sul bordo della matrice, si assuma che l'intensità al di fuori dell'immagine sia 0. La massima intensità ammissibile è 255.

Si richiede di scrivere un *kernel* OpenCL che data una matrice di input IN di dimensioni $w \times h$ che rappresenta un'immagine e una matrice di output OUT delle stesse dimensioni, calcoli OUT applicando l'operatore di Sobel all'immagine IN. Assumete un NDRange di dimensioni $[0, w-1] \times [0, h-1]$, e fate riferimento alla documentazione OpenCL distribuita sul sito del corso. L'esercizio non richiede di scrivere il codice host (che vi consiglio comunque di implementare per testare il kernel proposto).

Domanda 2. Analizzare il seguente codice OpenCL:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#ifdef __APPLE__
#include <OpenCL/opencl.h>
#include <unistd.h>
#else
#include <CL/cl.h>
#endif
```

```
#include "err_code.h"

//pick up device type from compiler command line or from the default type
#ifndef DEVICE
#define DEVICE CL_DEVICE_TYPE_DEFAULT
#endif

extern int output_device_info(cl_device_id );

#define LENGTH (1024)    // length of vectors a, b, c and d

//------------------------------------------------------------------------------
// kernel:  vadd
// Purpose: Compute the elementwise sum d = a+b+c
// input: a, b and c float vectors of length count
// output: d float vector of length count holding the sum a + b + c

const char *KernelSource = "\n"                              \
"__kernel void vadd(                                    \n" \
                       [FILL IN THIS PART]
"}                                                      \n" \
"\n";

//------------------------------------------------------------------------------
int main(int argc, char** argv) {
    cl_int          err;       // error code returned from OpenCL calls

    size_t dataSize = sizeof(float) * LENGTH;
    float*       h_a = (float *)malloc(dataSize);       // a vector
    float*       h_b = (float *)malloc(dataSize);       // b vector
    float*       h_c = (float *)malloc(dataSize);       // c vector
    float*       h_d = (float *)malloc(dataSize);       // d vector (result)
    unsigned int correct;           // number of correct results

    size_t global;                  // global domain size

    cl_device_id     device_id;     // compute device id
    cl_context       context;       // compute context
    cl_command_queue commands;      // compute command queue
    cl_program       program;       // compute program
    cl_kernel        ko_vadd;       // compute kernel

    cl_mem d_a;                     // device memory used for the input  a vector
    cl_mem d_b;                     // device memory used for the input  b vector
    cl_mem d_c;                     // device memory used for the input c vector
    cl_mem d_d;                     // device memory used for the output d vector

    // Fill vectors a and b with random float values
    int i = 0;
    for(i = 0; i < LENGTH; i++) {
        h_a[i] = rand() / (float)RAND_MAX;
        h_b[i] = rand() / (float)RAND_MAX;
        h_c[i] = rand() / (float)RAND_MAX;
    }

    cl_uint numPlatforms;

    err = clGetPlatformIDs(0, NULL, &numPlatforms);
    checkError(err, "Finding platforms");
    if (numPlatforms == 0) {
        printf("Found 0 platforms!\n");
```

```
    return EXIT_FAILURE;
}

cl_platform_id Platform[numPlatforms];
err = clGetPlatformIDs(numPlatforms, Platform, NULL);
checkError(err, "Getting platforms");

for (i = 0; i < numPlatforms; i++) {
    err = clGetDeviceIDs(Platform[i], DEVICE, 1, &device_id, NULL);
    if (err == CL_SUCCESS) { break; }
}

if (device_id == NULL)
    checkError(err, "Getting device");

err = output_device_info(device_id);
checkError(err, "Outputting device info");

context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
checkError(err, "Creating context");

commands = clCreateCommandQueue(context, device_id, 0, &err);
checkError(err, "Creating command queue");

program = clCreateProgramWithSource(context, 1, (const char **) & KernelSource, NULL, &err);
checkError(err, "Creating program");

err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if (err != CL_SUCCESS) {
    size_t len;
    char buffer[2048];

    printf("Error: Failed to build program executable!\n%s\n", err_code(err));
    clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
    printf("%s\n", buffer);
    return EXIT_FAILURE;
}

ko_vadd = clCreateKernel(program, "vadd", &err);
checkError(err, "Creating kernel");

d_a  = clCreateBuffer(context,  CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,  dataSize, h_a, &err);
checkError(err, "Creating buffer d_a");
d_b  = clCreateBuffer(context,  CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,  dataSize, h_b, &err);
checkError(err, "Creating buffer d_b");
d_c  = clCreateBuffer(context,  CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,  dataSize, h_c, &err);
checkError(err, "Creating buffer d_c");

d_d  = clCreateBuffer(context,  CL_MEM_WRITE_ONLY, dataSize, NULL, &err);
checkError(err, "Creating buffer d_d");

const int count = LENGTH;

global = count;
err = clEnqueueNDRangeKernel(commands, ko_vadd, 1, NULL, &global, NULL, 0, NULL, NULL);
checkError(err, "Enqueueing kernel 1st time");

err = clEnqueueReadBuffer( commands, d_d, CL_TRUE, 0, sizeof(float) * count, h_d, 0, NULL, NULL );
checkError(err, "Reading back d_f");

// cleanup then shutdown
```

```
    clReleaseMemObject(d_a);
    clReleaseMemObject(d_b);
    clReleaseMemObject(d_c);
    clReleaseMemObject(d_d);
    clReleaseProgram(program);
    clReleaseKernel(ko_vadd);
    clReleaseCommandQueue(commands);
    clReleaseContext(context);

    free(h_a);
    free(h_b);
    free(h_c);
    free(h_d);

    return 0;
}
```

1. Fornite il codice del kernel.
2. Descrivete sinteticamente i passi principali compiuti dal codice host.
3. Un'operazione fondamentale che dovrebbe essere compiuta dall'host manca: di quale operazione si tratta?
4. Descrivete dettagliatamente il layout di memoria e il flusso dei dati tra memoria host e device.