Ingegneria degli Algoritmi (A.A. 2013-2014)

Corsi di Laurea in Ingegneria Informatica e Automatica, Ingegneria dei Sistemi Informatici, e Laurea Magistrale in Ingegneria Informatica

Sapienza Università di Roma

Esercitazione di laboratorio (17/06/2014) – Durata 2h 30'

Preliminari (aula 17)

1. Fare login in Linux (Scientific Linux 5.1) con le credenziali fornite in aula.
2. Creare sulla scrivania (`/local/studente/Desktop`) una directory chiamata: `ida2014`
3. Entrare nella directory `/local/studente/Desktop/ida2014`
4. Copiare il testo e il codice fornito per l'esercitazione nella directory di lavoro: `cp -rf /home/distrib/demetres-sw/esame140617 .`
5. La directory `/local/studente/Desktop/ida2014/esame140617` contiene:
    a. `resize2x`: directory di lavoro per l'esercizio 1 (OpenCL)
    b. `stack`: directory di lavoro per l'esercizio 2 (C)
    c. `esempi`: esempi di programmi OpenCL da cui trarre ispirazione
    d. `docs`: documentazione utile su OpenCL

Esercizio 1: raddoppiamento delle dimensioni di un'immagine (11 punti)

Lo scopo dell'esercizio è quella di scrivere un modulo C che, data in input una immagine a toni 256 di grigio di dimensione $w \times h$, crei una nuova immagine allocata dinamicamente ottenuta da quella di input raddoppiandone altezza e larghezza, come nell'esempio sotto.

(a) Immagine originale a 256 toni di grigio di dimensione 500×334

(b) Immagine raddoppiata di dimensione 1000×668

Si vada nella directory di lavoro `resize2x` e si definisca nel file `resize2x.c` la funzione `resize2x` con il seguente prototipo:

```
void resize2x(unsigned char* in, int w, int h,
              unsigned char** out, int* ow, int* oh,
              clut_device* dev, double* td);
```

dove:

- `in`: puntatore a un buffer di dimensione `w*h*sizeof(unsigned char)` byte nella memoria dell'host che contiene l'immagine di input in formato row-major;
- `w`: larghezza di `in` in pixel (numero di colonne della matrice di pixel);
- `h`: altezza di `in` in pixel (numero di righe della matrice di pixel);
- `out`: parametro in cui restituire il puntatore all'immagine di output in formato row-major, **che la funzione deve allocare dinamicamente al suo interno**;
- `ow`: parametro in cui restituire la larghezza dell'immagine di output in pixel;
- `oh`: parametro in cui restituire l'altezza dell'immagine di output in pixel;
- `dev`: ambiente di esecuzione della GPU (si veda `clut.h`);
- `td`: parametro in cui restituire la durata dell'esecuzione del kernel.

Scrivere un opportuno kernel OpenCL nel file `resize2x.cl`.

Suggerimento: eseguire il kernel su un NDRange a due dimensioni dove la dimensione 0 corrisponde all'asse orizzontale (x) e la dimensione 1 corrisponde all'asse verticale (y).

L'immagine deve essere riscalata mediante interpolazione lineare facendo la media dei toni di grigio di pixel adiacenti come nel seguente esempio, dove la matrice di input è 3×3 e quella di output è 6×6 (sui bordi si usino i valori di grigio originali):

a	**b**	**c**
d	**e**	**f**
g	**h**	**i**

Immagine input

a	$(a+b)/2$	**b**	$(b+c)/2$	**c**	c
$(a+d)/2$	$(a+e)/2$	$(b+e)/2$	$(b+f)/2$	$(c+f)/2$	c
d	$(d+e)/2$	**e**	$(e+f)/2$	**f**	f
$(d+g)/2$	$(d+h)/2$	$(e+h)/2$	$(e+i)/2$	$(f+i)/2$	f
g	$(g+h)/2$	**h**	$(h+i)/2$	**i**	i
g	g	h	h	i	i

Immagine output

Compilazione e test.

Directory di lavoro: `~/Desktop/ida2014/esame140617/resize2x/`

1. Compilazione programma di test (*una tantum*): dare il comando `make`, che genera il file eseguibile `resize2x`
2. Compilazione ed esecuzione kernel:
   a. `make test1`: test su immagine 500×334 a 256 toni di grigio del Colosseo;
   b. `make test2`: test su una versione 512×512 a 256 toni di grigio del classico bechmark "Lena" usato in computer graphics.

Nella directory `results` verranno generate varie immagini ottenute aggiungendo diverse cornici alle immagini date in input. Le immagini, salvate in formato PGM (Portable Graymap Format), possono essere visualizzate con il programma Gimp.

Esercizio 2: operazioni su una pila con lista collegata (11 punti)

Si richiede di completare le funzioni push, pop e top di una pila relizzata mediante lista collegata in C, con i seguenti prototipi dichiarati nella header `stack.h`:

```
void stack_push(stack** s, int elem);
```

Aggiunge un elemento in cima alla pila:

1. `s`: indirizzo di un oggetto contenente il puntatore al nodo in cima alla pila;
2. `elem`: intero da inserire in cima alla pila.

```
int stack_pop(stack** s, int* err);
```

Toglie un elemento dalla cima alla pila:

1. `s`: indirizzo di un oggetto contenente il puntatore al nodo in cima alla pila;
2. `err`: se diverso da `NULL`, la funzione deve restiture nel parametro il valore 1 se la pila è vuota, e 0 altrimenti;
3. la funzione deve restituire l'elemento rimosso dalla cima della pila, o 0 se la pila è vuota.

```
int stack_top(const stack* s, int* err);
```

Restituisce l'elemento in cima alla pila senza toglierlo (`const` sta ad indicare che la pila non viene modificata):

1. `s`: puntatore al nodo in cima alla pila;
2. `err`: se diverso da `NULL`, la funzione deve restiture nel parametro il valore 1 se la pila è vuota, e 0 altrimenti;
3. la funzione deve restituire l'elemento in cima della pila, o 0 se la pila è vuota.

Scrivere le funzioni nel file `stack.c`. Il modulo contiene la definizione del tipo `stack` che rappresenta la struttura di un nodo della pila. E' utile esaminare il main di prova `main.c` per vedere un esempio di operazioni su una pila.

Compilazione e test.

- *Directory di lavoro*: `~/Desktop/ida2014/esame140617/stack/`
- *Compilazione programma di test*: dare il comando `make`, che genera il file eseguibile `stack` contenente un programma di test
- *Test correttezza*: eseguire il programma di test `./stack`.

E' consigliabile modificare il main per fare ulteriori test e verificare la correttezza delle funzioni scritte, purché non si modifichino i prototipi delle funzioni dichiarate in `stack.h`.

Nota bene: testare la correttezza dell'uso della memoria usando il tool Valgrind:

```
valgrind ./stack
```