Ingegneria degli Algoritmi (A.A. 2012-2013)

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

Sapienza Università di Roma

Appello del 12/07/2013 – Prova al calcolatore – Durata 2h 15'

Preliminari (aula 17)

1. Fare login in Linux (Scientific Linux 5.1) con le credenziali fornite in aula
2. La directory `/local/studente/Desktop/esame` contiene:
   a. `resize`: directory di lavoro per l'esercizio 1;
   b. `bintree`: directory di lavoro per gli esercizi 2 e 3;
   c. `esempi`: esempi di programmi OpenCL da cui trarre ispirazione;
   d. `docs`: documentazione utile su OpenCL;
   e. `studente`: file in cui inserire nome, cognome e matricola.

Esercizio 1: ridimensionamento di un'immagine (16 punti su 32)

L'esercizio richiede di scrivere una funzione che, data un'immagine di input a 256 toni di grigio di larghezza `w` e altezza `h`, genera un'immagine di output di larghezza `w/2` e altezza `h/2` ottenuta rimpicciolendo quella di partenza dimezzandone i lati. La funzione, da definire nel file `resize_device.c` deve creare l'immagine rimpicciolita di output mediante un opportuno kernel OpenCL, da definire nel file `resize.cl`, e deve avere il seguente prototipo:

```
void resize_device(unsigned char* in, unsigned char* out,
                   int w, int h, clut_device* dev, double* t)
```

Dove:

1. `in`: immagine di input di dimensione `w` × `h` (row-major);
2. `out`: immagine di output di dimensione `(w/2)×(h/2)` (row-major);
3. `w`: larghezza in pixel dell'immagine di input;
4. `h`: altezza in pixel dell'immagine di input;
5. `dev`: contesto di esecuzione sul device;
6. `t`: puntatore a un buffer double in cui scrivere la durata in secondi dell'esecuzione del kernel sul device.

La directory `esempi` fornisce esempi di programmi OpenCL da cui trarre ispirazione. Si veda in particolare `convolution`, dato come esercitazione il 28/5/2013.

Algoritmo di ridimensionamento.

Ogni pixel di coordinate (i,j) dell'immagine di output, dove i è la coordinata verticale (indice di riga) e j quella orizzontale (indice di colonna), deve essere ottenuta come media dei toni di grigio dei quattro pixel dell'immagine di input di coordinate (2i, 2j),

(2i, 2j+1), (2i+1, 2j) e (2i+1, 2j+1). Non usare alcun valore `float` o `double` per effettuare i calcoli.

Compilazione e test.

- *Directory di lavoro*: `resize/`
- *Compilazione programma*: dare il comando `make`, che genera il file eseguibile `resize`;
- *Compilazione ed esecuzione*: dare il comando `make test`.

Nelle directory `img_device` e `img_host` verranno generate le versioni rimpicciolite di quelle presenti nella directory `img`:

- `img`: immagini di input (fornite);
- `img_host`: immagini generate dal codice host (fornito in forma binaria);
- `img_device`: immagini generate dal codice device da scrivere come esercizio (a un'ispezione visuale devono essere uguali a quelle in `img_host`).

Le immagini, salvate in formato PGM (Portable Graymap Format), possono essere visualizzate con il programma Gimp.

Esercizio 2: deallocazione di un albero binario (8 punti su 32)

Si richiede di scrivere una funzione C che deallochi un albero binario i cui nodi sono definiti come segue (si veda `bintree/bintree.h`):

```
typedef struct bintree bintree;
struct bintree {
    void* elem;
    bintree *left, *right;
};
```

Dove `elem` è un puntatore al contenuto informativo associato al nodo, mentre `left` e `right` sono i puntatori al figlio sinistro e destro nell'albero, rispettivamente (o `NULL` se i figli sono assenti). La funzione, da definire nel file `bintree/bintree.c`, deve avere il seguente prototipo:

```
void bintree_delete(bintree* root)
```

Dove `root` è la radice dell'albero binario da deallocare.

Compilazione e test.

- *Directory di lavoro*: `bintree/`
- *Compilazione programma*: dare il comando `make bintree1`, che genera il file eseguibile `bintree1`;
- *Test funzionamento*: dare il comando `make test1`;
- *Test uso corretto della memoria*: dare il comando `make test1-valgrind`. Verificare che valgrind non riporti errori di accesso a memoria e memory leak.

Esercizio 3: ricerca in un albero binario di ricerca (8 punti su 32)

Si richiede di scrivere una funzione C che cerchi se una determinata chiave è presente in un albero binario di ricerca[1] i cui nodi sono definiti come nell'esercizio 2. La funzione, da definire nel file `bintree/bintree.c`, deve restituire 1 se la chiave appare nell'albero, e 0 altrimenti. La funzione deve poter essere **invocata** come segue (si veda la funzione `check` in `bintree/main.c`):

```
bintree_search(root, key, comparator)
```

dove:

1. `root` è il puntatore alla radice dell'albero di ricerca;
2. `key` è un puntatore `void*` a una chiave;
3. `comparator` è una funzione di confronto fra chiavi con il seguente prototipo: `int comparator(void* a, void* b)` che, dati i puntatori a due chiavi `a` e `b`, restituisce:
    - un intero negativo se la chiave `a` è minore della chiave `b`;
    - zero se le chiavi `a` e `b` sono uguali;
    - un intero positivo se la chiave `a` è maggiore della chiave `b`.

Compilazione e test.

- *Directory di lavoro*: `bintree/`
- *Compilazione programma*: dare il comando `make bintree2`, che genera il file eseguibile `bintree2`
- *Test funzionamento*: dare il comando `make test2`.
- *Test uso corretto della memoria*: dare il comando `make test2-valgrind`. Verificare che valgrind non riporti errori di accesso a memoria e memory leak.

[1] Si ricordi che in un **albero binario di ricerca**, se x è la chiave contenuta in un qualsiasi nodo v, tutte le chiavi nel sottoalbero sinistro di v saranno minori o uguali a x, e tutte le chiavi contenute nel sottoalbero destro di v saranno maggiori o uguali a x.