Programmazione Funzionale e Parallela (A.A. 2018-19)
Corso di Laurea in Ingegneria Informatica e Automatica
Sapienza Università di Roma

Esame del 28/01/2019 – Durata 1h 45'

Inserire nome, cognome e matricola nel file `studente.txt`.

Esercizio 1 (Scala)

Si vuole dotare il seguente tipo di dato albero binario di elementi di tipo generico `T` con un metodo `def map[S](f:T=>S):Tree[S]` che crea una albero con la stessa struttura, ma gli elementi nei nodi modificati in base a una certa funzione `f` data. La definizione del tipo albero è contenuta nel file `E1/E1.scala`, che si chiede di completare:

```
sealed abstract class Tree[T]
case class E[T]() extends Tree[T]
case class L[T](e:T) extends Tree[T]
case class N[T](l:Tree[T], e:T, r:Tree[T]) extends Tree[T]
```

La sottoclasse concreta `E()` denota un albero vuoto, `L(e)` denota un albero formato dalla sola radice `e`, mentre `N(l,e,r)` denota un albero con radice `e` avente come sottoalbero sinistro `l` e come sottoalbero destro `r`.

Dopo l'aggiunta del metodo `map` alla classe `Tree`, deve essere possibile ad esempio eseguire il seguente frammento di codice:

```
val t = N(L(12),2,N(E(),49,L(6)))
val s = t.map(x=>x+"|"+x)
println(s) // stampa: N(L(12|12),2|2,N(E(),49|49,L(6|6)))
```

Per testare a fondo la soluzione, usare il main di prova contenuto nel file `E1/E1Main.scala` riportato insieme al compito. La soluzione non deve usare alcun costrutto della programmazione imperativa e in particolare alcuna variabile `var`. Non toccare nessun file tranne `E1.scala`.

Esercizio 2 (Scala)

Si vuole scrivere nel file `E2/E2.scala` un metodo Scala `def extract[T](l:Seq[T], idx:Set[Int]):Seq[T]` che, data una sequenza `l` e un insieme di indici `idx`, restituisce una nuova sequenza che contiene, nello stesso ordine, tutti gli elementi di `l` che hanno indici contenuti in `idx`. Si assuma che gli indici di l vadano da `0` a `l.size-1`.

Scrivere la soluzione in un in modo che sia possibile compilare ed eseguire correttamente il programma di prova `E2/E2Main.scala` riportato insieme al compito. La soluzione non deve usare alcun costrutto della programmazione imperativa e in particolare alcuna variabile `var`. Non toccare nessun file tranne `E2.scala`.

Esercizio 3 (Vettorizzazione SSE)

Si vuole realizzare una variante vettorizzata mediante SSE della seguente funzione C fornita nel file `E3/minmax.c` che calcola simultaneamente il minimo e il massimo di un array non vuoto di `n` valori `unsigned char`:

```
void minmax(const unsigned char* v, int n,
            unsigned char* pmin, unsigned char* pmax){
    int i;
    *pmax = 0x00; // minimo valore unsigned char
```

```
    *pmin = 0xFF; // massimo valore unsigned char
    for (i=0; i<n; ++i) {
        if (v[i] > *pmax) *pmax = v[i];
        if (v[i] < *pmin) *pmin = v[i];
    }
}
```

La soluzione deve essere scritta nel file `E3/minmax_sse.c`. Compilare il programma di prova con `make` ed eseguirlo con `./minmax`. Non toccare nessun file tranne `minmax_sse.c`.

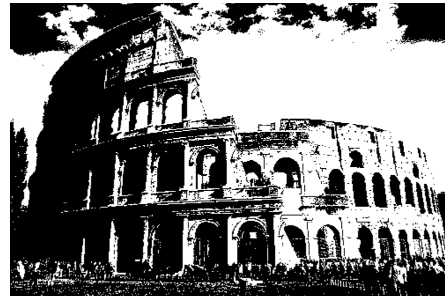
Suggerimento. È possibile usare i seguenti intrinsic SSE:

- `__m128i _mm_set1_epi8(char a)`: restituisce un packed 128-bit integer con tutti i 16 byte inizializzati con il byte fornito in `a`.
- `__m128i _mm_min_epu8(__m128i a, __m128i b)`: calcola il minimo byte a byte senza segno dei packed 128-bit integer `a` e `b` (vedi anche dispensa allegata in `docs`). La versione con `max` calcola il massimo.

Esercizio 4 (OpenCL)

Lo scopo dell'esercizio è quello di scrivere un modulo C basato su OpenCL che, data in input un'immagine a 256 toni di grigio di dimensione $w \times h$, crei una nuova immagine delle stesse dimensioni ottenuta rendendo "binaria" l'immagine in modo che ogni pixel x venga portato a 0 (nero) se strettamente inferiore a una certa soglia data come parametro e a 255 (bianco) altrimenti. Esempio:

(a) Immagine originale


(b) Immagine binarizzata con soglia 128

Si completi nel file `E4/bw.c` la funzione `bw` con il seguente prototipo:

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

dove:

- `in`: puntatore a un buffer di dimensione `w*h*sizeof(unsigned char)` byte che contiene l'immagine di input in formato row-major[1];
- `out`: puntatore a un buffer di dimensione `w*h*sizeof(unsigned char)` byte che contiene l'immagine di output in formato row-major;
- `h`: altezza di `in` e `out` in pixel (numero di righe della matrice di pixel);
- `w`: larghezza di `in` e `out` in pixel (numero di colonne della matrice di pixel);
- `threshold`: soglia per la binarizzazione (< porta a nero, ≥ porta a bianco);
- `t`: puntatore a `double` in cui scrivere il tempo di esecuzione in secondi richiesto dall'esecuzione del kernel.

---

[1] Cioè con le righe disposte consecutivamente in memoria.

Per compilare usare il comando `make`. Lanciare il programma con `./bw`. Verrà prodotta l'immagine di output `results/colosseo-bw.pgm`. Non toccare nessun file tranne `bw.c` e `bw.cl`.

Nota: è normale che in un ambiente virtualizzato come VirtualBox con immagini di piccole dimensioni il tempo di esecuzione della versione sequenziale sia inferiore a quello della versione OpenCL.