Programmazione Funzionale e Parallela

Corso di Laurea in Ingegneria Informatica e Automatica - A.A. 2019-2020

Esercitazione del 4 maggio 2020

Istruzioni per l'esercitazione:

- Aprite il form di consegna in un browser e loggatevi con le vostre credenziali `uniroma1`.
- Scaricate e decomprimete sulla scrivania il codice dell'esercitazione. Vi sarà una sotto-directory separata per ciascun esercizio di programmazione. Non modificate in alcun modo i programmi di test `E*Main.scala`.
- Rinominare la directory chiamandola `cognome.nome`. Sulle postazioni del laboratorio sarà `/home/studente/Desktop/cognome.nome/`.
- È possibile consultare appunti/libri e il materiale didattico online.
- Rispondete alle domande online sul modulo di consegna.
- **Finiti gli esercizi**, e non oltre le 23:59:
  - **zippate la directory di lavoro** in `cognome.nome.zip` (`zip -r cognome.nome.zip cognome.nome/`).
- **Per consegnare**:
  - inserite nel form di consegna come autovalutazione il punteggio di ciascuno dei test forniti (inserite zero se l'esercizio non è stato svolto, non compila, o dà errore di esecuzione).
  - fate **upload** del file `cognome.nome.zip`.
- **Se avete domande** accedete a Google Meet all'indirizzo meet.google.com/sph-eiax-fsv durante orario 14:00-16:00 stabilito per l'esercitazione accedendo con la vostra **mail istituzionale**. Troverete online il docente e il tutor del corso. In alternativa, scrivete via email.

Per maggiori informazioni fate riferimento al regolamento delle esercitazioni.

Esercizio 1 (ciclo interno prodotto di matrici con vettorizzazione)

Si scriva una versione vettorizzata SSE o AVX `add_prod` della seguente funzione `add_prod_seq` inclusa nel file `E3/e3.c` scaricato:

```
static void add_prod_seq(const short* src, short* dst, short x, int n) {
    int j;
    for (j=0; j < n; ++j) dst[j] += x * src[j];
}
```

La funzione realizza il ciclo più interno in un codice di prodotto di matrici di `short`.

Suggerimento. Se si opta per SSE, usare in particolare:

- `_mm_set1_epi16` per costruire un `__m128i` con tutti i suoi `short` inizializzati a un valore specifico;
- `_mm_mullo_epi16` per effettuare il prodotto elemento a elemento di due `__m128i` che contengono ciascuno 8 `short`.

Si veda il sito Intel e la dispensa sul parallelismo per la documentazione sugli intrinsic SSE/AVX.

Usare il main di prova nella directory di lavoro `E1` digitando `make` per compilare e `./es1` per eseguire il programma. Testare inoltre il codice con `valgrind ./e1`. Non modificare alcun file tranne `e1.c` e nessuna funzione in `e1.c` tranne `add_prod`.

Esercizio 2 (numero di occorrenze in array con vettorizzazione)

Si scriva una versione vettorizzata SSE o AVX `count_occ` della seguente funzione `count_occ_seq` inclusa nel file `E4/e4.c` scaricato:

```
int count_occ_seq(const char* v, int n, char x) {
    int i, cnt = 0;
    for (i=0; i < n; ++i) cnt += v[i] == x;
    return cnt;
}
```

La funzione conta il numero di occorrenze di `x` nell'array `v` di lunghezza `n`.

Suggerimento. Se si opta per SSE, usare in particolare:

- `_mm_set1_epi8` per costruire un `__m128i` con tutti i 16 byte inizializzati a un valore `char` specifico;
- `_mm_setzero_si128` per costruire un `__m128i` con tutti i 128 bit a zero;
- `_mm_cmpeq_epi8(a,b)` che restituisce un `__m128i` in cui l'i-esimo byte è `0xFF` se gli i-esimi byte di `a` e `b` sono uguali, e `0x00` altrimenti.

Si veda per ispirazione la soluzione dell'esercizio `cdiff` discussa in dettaglio nella lezione del 27 aprile 2020.

Usare il main di prova nella directory di lavoro `E2` digitando `make` per compilare e `./es2` per eseguire il programma. Testare inoltre il codice con `valgrind ./e2`. Non modificare alcun file tranne `e4.c` e nessuna funzione in `e2.c` tranne `count_occ`.