Corso di High Performance Computing

Esercitazione SIMD del 18/12/2018

Moreno Marzolla

Ultimo aggiornamento: 2018-12-18

Per svolgere l'esercitazione è possibile collegarsi tramite ssh al server `disi-hpc.csr.unibo.it` oppure `isi-raptor03.csr.unibo.it` (entrambi supportano le estensioni SIMD di Intel); allo scopo è possibile usare il programma putty (che dovrebbe essere presente sul desktop) usando come *username* il vostro indirizzo mail istituzionale completo, e come password la vostra password istituzionale (cioè quella che usate per accedere alla casella di posta). Ad esempio, se il vostro indirizzo mail è paolo.rossi@studio.unibo.it, allora la vostra username è `paolo.rossi@studio.unibo.it` (per intero).

0. Verifica dell'ambiente

Verificare quali estensioni SIMD sono supportate dalla CPU esaminando l'output del comando `cat /proc/cpuinfo` oppure del comando `lscpu`. Cercare nel campo `flags` la presenza delle sigle `mmx`, `sse`, `sse2`, `sse3`, `sse4_1`, `sse4_2`, `avx`, `avx2`.
Il Makefile fornito compila i programmi con i flag seguenti:

- `-march=native` abilita tutte le istruzioni supportate dalla macchina su cui si sta compilando;
- `-g -ggdb` genera informazioni di debug; utile per mostrare il codice sorgente corrispondente al codice assembly (vedi seguito).

Può risultare utile analizzare il codice assembly prodotto dal compilatore, ad esempio per vedere se vengono effettivamente usate istruzioni SIMD. Per fare ciò è sufficiente dare in comando:

```
objdump -dS nome_eseguibile
```

Per sapere quali flag del compilatore sono attivi con `-march=native` è possibile dare il comando:

```
gcc -march=native -Q --help=target
```

1. Prodotto scalare

Il file `simd-dot.c` contiene una funzione che calcola il prodotto scalare tra due array di `float`. Scopo di questo esercizio è di implementare la funzione `simd_dot()` per il calcolo del prodotto scalare usando parallelismo SIMD, procedendo secondo i passi descritti sotto.

Il programma `simd-dot.c` stampa il tempo medio di esecuzione della versione seriale e parallela della funzione di calcolo, ottenuto ripetendo il calcolo un certo numero di volte. Il calcolo del prodotto scalare è una procedura piuttosto semplice che richiede pochissimo tempo anche con array di grandi dimensioni. Di conseguenza, *potrebbero* non apparire differenze significative tra le prestazioni della versione SIMD e di quella scalare.

1. Auto-vettorizzazione. Verificare l'efficacia del compilatore nell'auto-vettorizzazione della funzione `scalar_dot()`. Si compili il programma con il seguente comando:

```
gcc -O2 -march=native -ftree-vectorize -fopt-info-vec-optimized -fopt-info-
vec-missed simd-dot.c -lm -o simd-dot 2>&1 | grep "loop vectorized"
```

I flag `-fopt-info-vec-XXX` stampano una serie di messaggi “informativi” (per modo di dire) sullo standard error indicando quali cicli sono stati vettorizzati e quali no. La riga di comando redireziona lo standard error sullo standard output e cerca la stringa “loop vectorized” che viene stampata dal compilatore quando riesce a vettorizzare un ciclo. Non dovrebbe venire stampato alcun messaggio, il che significa che la vettorizzazione automatica non ha avuto successo.

2. Auto-vettorizzazione (secondo tentativo). Esaminare i messaggi di output del compilatore (rimuovere `2>&1 …` dalla riga di comando precedente); dovrebbe essere presente un messaggio simile al seguente:

```
simd-dot.c:48:5: note: reduction: unsafe fp math optimization: r_15 =
_14 + r_20;
```

La riga 48 (nella mia versione del sorgente) è quella che contiene il ciclo “for” della funzione `scalar_dot()`; si tratta dello stesso messaggio di cui abbiamo parlato a lezione, e corrisponde al fatto che le istruzioni:

```
r += x[i] * y[i];
```

sono parte di una operazione di riduzione che coinvolge operandi di tipo *float*. Poiché l'aritmetica floating-point non è in generale commutativa, il compilatore non altera l'ordine delle istruzioni per non rischiare di modificare il risultato finale. Per ignorare il problema si ricompili il programma aggiungendo il flag `-funsafe-math-optimizations`; ripetendo la compilazione con:

```
gcc -O2 -march=native -ftree-vectorize -fopt-info-vec-optimized -fopt-info-
vec-missed -funsafe-math-optimizations simd-dot.c -lm -o simd-dot 2>&1 |
grep "loop vectorized"
```

dovrebbe ora comparire (due volte) il messaggio

```
simd-dot.c:48:5: note: loop vectorized
```

che indica che il ciclo è stato vettorizzato.

3. Vettorizzare manualmente il codice. Si realizzi la funzione `simd_dot()` usando i *vector datatype* del compilatore GCC. La funzione sarà quasi identica alla funzione per il calcolo della somma del contenuto di un array vista a lezione (si faccia riferimento al programma `simd-vsum-vector.c` nell'archivio degli esempi). La funzione `simd_dot()` realizzata deve funzionare correttamente per qualsiasi lunghezza n degli array di input, che non deve quindi essere multipla dell'ampiezza dei registri SIMD; si garantisce però che gli array di input siano sempre correttamente allineati.

2. Prodotto di matrici

Il file `simd-matmul.c` contiene la versione seriale del prodotto matrice-matrice $r = p \times q$, sia nella versione “normale” che nella versione *cache-efficient* che traspone la matrice q in modo da accedere per riga anche a q sfruttando al meglio la cache della CPU (ne abbiamo parlato all'inizio del corso).

La versione *cache-efficient* consente di sfruttare le estensioni SIMD del processore, in quanto il prodotto riga-colonna diventa un prodotto riga-riga, ed è quindi possibile trasferire elementi contigui dalla memoria in registri SIMD. Osserviamo che il corpo della funzione `scalar_matmul_tr()`

```
for (i=0; i<n; i++) {
```

```
        for (j=0; j<n; j++) {
            double s = 0.0;
            for (k=0; k<n; k++) {
                s += p[i*n + k] * qT[j*n + k];
            }
            r[i*n + j] = s;
        }
    }
```

non fa altro che calcolare il prodotto scalare di due vettori di *n* elementi memorizzati a partire dagli indirizzi di memoria `(p + i*n)` e `(qT + j*n)`. Sfruttando il calcolo SIMD del prodotto scalare realizzato nell'esercizio precedente, realizzare la funzione `simd_matmul_tr()` in cui il prodotto scalare di cui sopra viene calcolato usando i *vector datatype* del compilatore. Si garantisce che le dimensioni delle matrici siano multiple della lunghezza dei vettori SIMD.

Si presti attenzione che in questo esercizio si usa il tipo `double`; è pertanto necessario definire un tipo vettoriale `v2d`, di ampiezza 16 Byte e composto da due valori `double`, utilizzando una dichiarazione del tutto simile a quella vista a lezione:

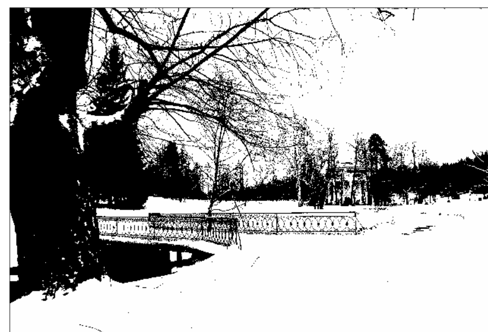
```
typedef double v2d __attribute__((vector_size(16)));
#define VLEN (sizeof(v2d)/sizeof(double))
```

Il processore del server usato per le esercitazioni supporta le estensioni SIMD di Intel fino ad AVX2. e pertanto dispone di registri SIMD di ampiezza 256 bit = 32 Byte. Si provi a modificare la propria implementazione per sfruttare un tipo vettoriale `v4d` contenente 4 valori di tipo `double`.

3. Image thresholding

Consideriamo una immagine bitmap a toni di grigio di *M* righe e *N* colonne, in cui la tonalità di grigio di ogni pixel è codificata con un intero nell'intervallo 0 (bianco) – 255 (nero). Dato un parametro intero $0 \leq thr < 255$, la funzione `threshold(img, thr)` modifica l'immagine `img` rendendo bianchi tutti i pixel il cui tono di grigio è minore o uguale a *thr* e neri tutti gli altri (si veda l'esempio in figura 1).

Originale



thr = 128

Figura 1: Esempio di applicazione dell'operatore threshold

Il file `simd-threshold.c` contiene una implementazione seriale dell'operatore threshold. Scopo di questo esercizio è svilupparne una versione SIMD utilizzando i *vector datatype* del compilatore GCC. Poiché ogni pixel dell'immagine è codificato da un valore di tipo `unsigned char`, è possibile rappresentare i valori di 16 pixel in un vettore SIMD ampio 16 byte. Definiamo un tipo `v16uc` per rappresentare un vettore SIMD composto da 16 elementi di tipo `unsigned char`:

```
typedef unsigned char v16uc __attribute__((vector_size(16)));
#define VLEN (sizeof(v16uc)/sizeof(unsigned char))
```

L'idea è di elaborare l'immagine a blocchi di 16 pixel adiacenti. L'unica difficoltà consiste nella struttura condizionale necessaria per determinare il nuovo colore di un pixel. Il blocco di codice:

```
unsigned char *pixel = bmp + i*width + j;
if (*pixel <= thr) {
     *pixel = 0;
} else {
     *pixel = 255;
}
```

deve essere modificato utilizzando la tecnica di "selection and masking" vista a lezione, in modo da eliminare la condizione e poter usare solo operazioni SIMD. A tale scopo definiamo due costanti di tipo `v16uc` che contengono tutti valori 0 e tutti valori 255, rispettivamente:

```
const v16uc black = {255, 255, 255, …, 255}; /* 16 valori 255 */
const v16uc white = {0, 0, 0, …, 0};         /* 16 valori 0   */
```

Supponiamo che `pixels` sia un puntatore a 16 pixel adiacenti nell'immagine. L'espressione `mask = (*pixels <= thr)` produce come risultato un array SIMD di tipo `v16uc` i cui elementi valgono -1 in corrispondenza dei pixel con valore minore o uguale alla soglia, e 0 per gli altri pixel. Pertanto, l'if precedente può essere riscritto in forma vettoriale come:

```
v16uc *pixels    = (v16uc*)(bmp + i*width + j);
const v16uc mask = (*pixels <= thr)
*pixels = (mask & white) | (~mask & black);
```

Si noti che l'espressione `(mask & white)` produce come risultato un vettore SIMD i cui elementi valgono sempre tutti zero (perché?), quindi è possibile scrivere semplicemente:

```
*pixels = (~mask & black);
```

Si noti che per funzionare correttamente la versione SIMD richiede che:

1. La bitmap sia allocata a partire da un indirizzo di memoria multiplo di 16;
2. La larghezza dell'immagine sia multipla dell'ampiezza di un registro SIMD (16, nel nostro caso)

Entrambe le condizioni precedenti vengono già soddisfatte nel codice fornito.

4. La mappa del gatto versione SIMD

Lo scopo di questo esercizio è sviluppare una versione SIMD di una funzione che calcola l'iterazione della *mappa del gatto di Arnold*, una vecchia conoscenza che abbiamo già incontrato nelle esercitazioni OpenMP e CUDA. Riportiamo nel seguito la descrizione del problema.

La mappa del gatto trasforma una immagine quadrata P di dimensione $N \times N$ in una nuova immagine P' delle stesse dimensioni: il pixel di coordinate (x, y) in P, $0 \leq x < N$, $0 \leq y < N$, viene collocato nella posizione (x', y') di P' dove:

$$x' = (2x + y) \bmod N, \quad y' = (x + y) \bmod N$$

(mod è l'operatore modulo, corrispondente all'operatore `%` del linguaggio C). Si può assumere che le coordinate (0, 0) indichino il pixel in alto a sinistra e le coordinate $(N - 1, N - 1)$ quello in basso a destra, in modo da poter rappresentare l'immagine come una matrice.

La mappa del gatto ha proprietà sorprendenti. Applicata ad una immagine, se ne ottiene una versione molto distorta. Applicando nuovamente la mappa a quest'ultima immagine, se ne ottiene un'altra ancora più distorta, e così via (figura 2). Tuttavia, dopo un certo numero di iterazioni (il cui valore dipende dalla dimensione dell'immagine, e nel caso di immagini quadrate di dimensione

$N \times N$ risulta sempre minore o uguale a $3N$) ricompare l'immagine di partenza.

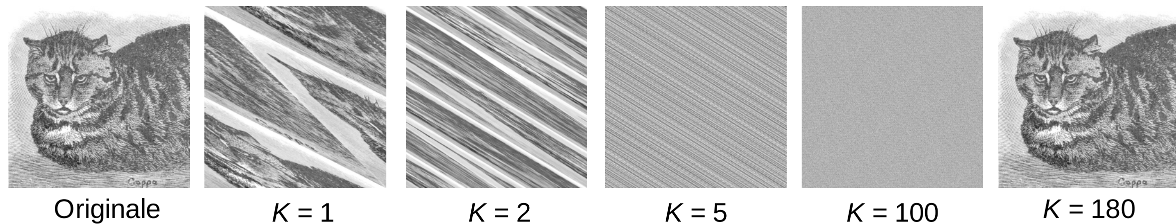


Figura 2: Esempio di applicazione iterata della mappa del gatto

Nel caso dell'immagine `cat.pgm` fornita come esempio, il tempo minimo di ricorrenza (cioè il numero minimo di iterazioni dopo le quali ricompare l'immagine originale) è 180; quindi, iterando k volte della mappa del gatto si otterrà l'immagine originale se e solo se k è multiplo di 180. Al momento non è nota alcuna relazione che lega il tempo minimo di ricorrenza alla dimensione N dell'immagine.
Viene fornito un programma sequenziale che calcola la k-esima iterata della mappa del gatto usando la CPU. Il programma viene invocato specificando sulla riga di comando il numero di iterazioni k. Il programma legge una immagine in formato PGM da standard input, e produce una nuova immagine su standard output ottenuta applicando k volte la mappa del gatto. Occorre ricordarsi di redirezionare lo standard output su un file, come indicato nelle istruzioni nel sorgente.
La struttura della funzione che calcola la k-esima iterata della mappa del gatto è molto semplice:

```
for (y=0; y<N; y++) {
      for (x=0; x<N; x++) {
            "calcola le coordinate (xnew, ynew) del punto (x, y)
                  dopo k applicazioni della mappa del gatto"
            next[xnew + ynew*N] = cur[x+y*N];
      }
}
```

Per sfruttare il parallelismo SIMD possiamo ragionare come segue: anziché calcolare le nuove coordinate di un punto alla volta, calcoliamo le coordinate di quattro punti adiacenti (x, y), $(x + 1, y)$, $(x + 2, y)$, $(x + 3, y)$ usando i vector datatype del compilatore. Per fare questo, definiamo le seguenti variabili di tipo `v4i` (vettori SIMD di 4 interi):

- *vx*, *vy*: coordinate di quattro punti adiacenti, prima dell'applicazione della mappa del gatto;
- *vxnew*, *vynew*: nuova coordinate dei punti di cui sopra dopo l'applicazione della mappa del gatto.

Ricordiamo che il tipo `v4i` si definisce con gcc come

```
typedef int v4i __attribute__((vector_size(16)));
#define VLEN (sizeof(v4i)/sizeof(int))
```

Posto $vx = \{x, x + 1, x + 2, x + 3\}$, $vy = \{y, y, y, y\}$, possiamo applicare ad essi le stesse operazioni aritmetiche applicate agli scalari x e y per ottenere le nuove coordinate *vxnew*, *vynew*. Fatto questo, al posto della singola istruzione:

```
            next[xnew + ynew*N] = cur[x+y*N];
```

per spostare materialmente i pixel nella nuova posizione occorre eseguire quattro istruzioni scalari:

```
next[vxnew[0] + vynew[0]*N] = cur[vx[0] + vy[0]*N];
next[vxnew[1] + vynew[1]*N] = cur[vx[1] + vy[1]*N];
next[vxnew[2] + vynew[2]*N] = cur[vx[2] + vy[2]*N];
next[vxnew[3] + vynew[3]*N] = cur[vx[3] + vy[3]*n];
```

Si assuma che la dimensione N dell'immagine sia sempre multipla di 4.

Estensione

Le prestazioni della versione SIMD dell'algoritmo della mappa del gatto dovrebbero risultare solo marginalmente migliori della versione scalare (potrebbero essere peggiori). Analizzando il codice assembly prodotto dal compilatore, si scopre che il calcolo del modulo nelle due espressioni

```
vxnew = (2*vxold+vyold) % N;
vynew = (vxold + vyold) % N;
```

viene realizzato usando operazioni scalari. Consultando la lista dei *SIMD intrinsics* sul sito di Intel (il link è sulla pagina Web del corso) si scopre infatti che non esiste una istruzione SIMD che realizzi la divisione intera. Per migliorare le prestazioni del programma occorre quindi ingegnarsi per calcolare i moduli senza fare uso della divisione. Ragionando in termini scalari, osserviamo che se $0 \leq xold < N$ e $0 \leq yold < N$, allora si ha necessariamente che $0 \leq 2 \times xold + yold < 3N$ e $0 \leq xold + yold < 2N$.
Pertanto, sempre in termini scalari, possiamo realizzare il calcolo di *xnew* e *ynew* come segue:

```
xnew = (2*xold + yold);
if (xnew >= N) { xnew = xnew - N; }
if (xnew >= N) { xnew = xnew - N; }
ynew = (xold + yold);
if (ynew >= N) { ynew = ynew - N; }
```

Il codice sopra è certamente meno leggibile della versione scalare, ma ha il vantaggio di poter essere vettorizzato ricordando che gli *if* possono essere vettorizzati con il meccanismo di "*selection and masking*" visto a lezione. Quindi ad esempio il codice scalare:

```
if (xnew >= N) { xnew = xnew - N; }
```

può essere vettorizzato come

```
const v4i mask = (xnew >= N);
xnew = ((xnew - N) & mask) | (xnew & ~mask);
```

Si ottiene in questo modo un codice sorgente più complesso rispetto alla versione scalare, ma molto più veloce in quanto si riesce a sfruttare al meglio le istruzioni SIMD.