Corso di High Performance Computing

Esercitazione SIMD del 19/5/2017

Moreno Marzolla

Ultimo aggiornamento: 2017/05/19

Per svolgere l'esercitazione è necessario collegarsi al server `disi-hpc.csr.unibo.it` tramite ssh; 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`. Nel campo `flags` occorre cercare la presenza di una o più delle sigle `mmx`, `sse`, `sse2`, `sse3`, `sse4_1`, `sse4_2`, `avx`, `avx2`.

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. La funzione deve operare correttamente per qualsiasi lunghezza *n* dei vettori di input, che non deve quindi essere multiplo dell'ampiezza dei registri SIMD. Fare uso dei *vector datatype* del compilatore `gcc`, cioè del tipo `v4f` visto a lezione; è possibile ispirarsi al programma `simd-vsum-vector.c` visto a lezione.

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. Nella mia implementazione osservo uno speedup di circa 1.62 rispetto alla versione scalare sulla macchina `disi-hpc.csr.unibo.it`; ripetendo il test su una macchina fisica dotata di un processore recente, lo speedup sale a circa 2. I vostri risultati potrebbero essere differenti, in base a come realizzate la versione SIMD.

Può risultare istruttivo verificare l'efficacia del compilatore nell'auto-vettorizzazione della funzione `scalar_dot()`. Una volta realizzata la funzione `simd_dot()` funzionante, provare a ricompilare il programma con il seguente comando:

```
gcc -O2 -march=native -ftree-vectorize -fopt-info-vec-optimized \
          -fopt-info-vec-missed simd-dot.c -o simd-dot
```

I flag `-fopt-info-vec-XXX` stampano una serie di messaggi "informativi" (per modo di dire) su quali cicli sono stati vettorizzati e quali no. Provando ad eseguire il programma, la funzione `scalar_dot()` non dovrebbe risultare più veloce di prima, indicando che il compilatore non è riuscito a ottimizzarla. Esaminando l'output della compilazione, si nota un possibile indizio:

```
simd-dot.c:42:5: note: reduction: unsafe fp math optimization: r_83 =
_81 + r_124;
```

La riga 42 (nella mia versione del sorgente) è quella che contiene il ciclo "for" della funzione `scalar_dot()`; in sostanza il compilatore segnala che il corpo del ciclo, composto dalle istruzioni:

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

costituisce una operazione di *riduzione*, ma poiché coinvolge operandi di tipo *float* non è possibile alterarne l'ordine senza rischiare di modificare il risultato finale. Questo è corretto, dato che l'aritmetica in virgola mobile non gode delle proprietà che ci aspetteremmo; in particolare, la somma in virgola mobile non è associativa né commutativa. Possiamo però forzare il compilatore a ignorare il problema con 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 -o simd-dot
```

si dovrebbe notare come la funzione `scalar_dot()` vettorizzata automaticamente abbia prestazioni simili alla funzione `simd_dot()` vettorizzata "a mano".

2. Prodotto di matrici

Il file `simd-matmul.c` contiene la versione scalare del prodotto matrice-matrice $r = p \times q$, sia nella versione "normale" sia 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 anche 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 infatti che il ciclo "for" più interno di questo frammento 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;
         }
    }
```

calcola il prodotto scalare di due vettori di n elementi memorizzati a partire dagli indirizzi `(p + i*n)` e `(qT + j*n)`. Sfruttando l'esercizio precedente, realizzare la funzione `simd_matmul_tr()` in cui il prodotto scalare di cui sopra vien effettuato usando i *vector datatype* del compilatore.

Si presti attenzione che qui abbiamo a che fare con il tipo di dato `double`; è pertanto necessario definire un tipo vettoriale `v2d`, di ampiezza sempre 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))
```

Suggerimento: è possibile riutilizzare la funzione per il calcolo SIMD del prodotto scalare dall'esercizio precedente, modificata opportunamente per operare su `double` anziché `float`.