Tirocinio su Posit

Luigi Leonardi

Indice

1 Introduzione

1.1 Posit

Il Posit è un formato di numero in virgola mobile ideato da John Gustafson, in alternativa allo standard IEEE 754. L'idea di base è fondamentalmente la stessa, anche nei Posit è presente un bit per il segno, dei bit per l'esponente e dei bit per la mantissa, le principali differenze consistono nella presenza di un "super esponente" o regime e nel non avere un numero di bit fissato per quest'ultimo e per la mantissa.
Il vantaggio nell'avere un super esponente, la cui lunghezza non è definita,

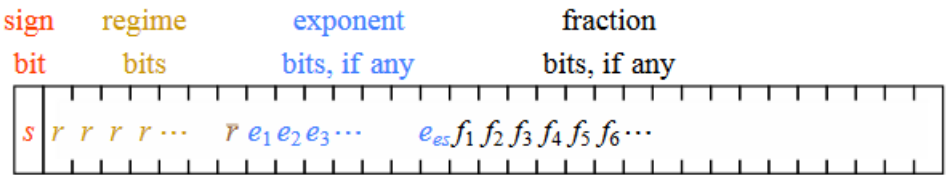

Figura 1: "Formato Posit"

permette di ottenere un range di numeri molto più flessibile, il suo contributo è pari a $2^{2^{es}}$ con es il numero di bit dell'esponente[1], permettendo quindi una riduzione del numero di bit assegnati a quest'ultimo campo oppure un incremento di range.
Questi bit di regime non sono altro che una sequenza di cifre binarie identiche, terminate dal complemento di esse.
Ad esempio: 0001 rappresenta -3, dove 3 è il numero degli 0 ed 1 è il terminatore.[2][3]

1.2 Note sui Test

Tutti i test svolti sui Posit sono stati effettuati sfruttando la libreria in C++ BFP[1] implementando, dove necessario, le funzionalità mancanti.
I Posit scelti sono stati a 32/64 bit, con 0 bit di esponente[3], dove non specificato diversamente.

[1] L'esponente è l'unico campo ad avere una dimensione fissa
[2] Regimi che iniziano per 0 sono negativi, per 1 invece sono positivi. Lo 0 è rappresentato come 10
[3] Ha senso non utilizzare bit di esponente, in quanto si sfrutta il super esponente

2 Test

2.1 Accuratezza

Questo test vuole dimostrare che a parità di bit utilizzati, 32 in questo caso, i Posit risultano essere più precisi nel rappresentare i risultati di prodotti, rispetto ad un Float in precisione singola.
In questo particolare caso sono stati impiegati Posit [32,3], ossia 32 bit totali, di cui 3 di esponente.

2.1.1 Svolgimento

Il test consiste nel creare 9 array, 6 per gli operandi e 3 per i risultati, di una dimensione compresa nell'ordine dei milioni:

- 3 di Posit [32,3]
- 3 di Float a 32 bit
- 3 di Double a 64 bit come riferimento

Il passo successivo consiste nel popolare gli array degli operandi, operazione che viene svolta partendo dai double sfruttando una funzione che genera dei numeri in modo pseudo-casuale, per poi provvedere a riempire gli altri semplicemente operando le necessarie conversioni.
Successivamente sono state eseguiti i vari prodotti i cui risultati sono stati collocati nel terzo array di ciascun tipo. Infine, per poter confrontare il tutto, i risultati sono stati riportarti in double e ne è stata fatta la differenza, in modulo, rispetto al riferimento.

Definisco:

$$\Delta_i(\text{Posit}) = |PositRes_i - DoubleRes_i|$$
$$\Delta_i(\text{Float}) = |FloatRes_i - DoubleRes_i|$$

Dove FloatRes e PositRes sono il risultato del prodotto, fra float e posit rispettivamente, convertiti in double.
Se $\Delta_i(\text{Posit}) < \Delta_i(\text{Float})$ i posit sono più precisi dei float per questo indice.
Se $\Delta_i(\text{Posit}) > \Delta_i(\text{Float})$ i float sono più precisi dei posit per questo indice.

2.1.2 Conclusioni

Dai risultati dei test svolti, i Posit si sono rivelati essere più precisi dei Float nel $\approx 56\%$ dei casi. Sono risultati pari merito nel $\approx 16\%$ dei casi.
I risultati risultano essere in linea con le aspettative, in quanto una maggiore precisione, a parità di dimensione, è imputabile ad un numero maggiore di bit disponibili per la mantissa, rispetto ai Float.

2.1.3 Codice

```
1 //Funzione per Numeri Random
2 double genererateNumber(){        // numeri fra -30 e 30
3         double tmp = (double)rand()/(double)RAND_MAX;
4
5         return (tmp*60) -30;
6 }
```

2.2 Tempi

Lo scopo di questo test è incentrato sul capire se, oltre ad esservi dei vantaggi a livello di precisione, vi sono dei vantaggi a livello di prestazioni nell'eseguire moltiplicazioni.
I Posit utilizzati sono del tipo [32-3].

2.2.1 Svolgimento

Per poter effettuare dei test che non avvantaggiassero i Float, sfruttando supporto hardware, è stata sfruttata la libreria SoftFloat[2] la quale implementa i Float interamente via software.
Lo svolgimento del test è simile al precedente, vengono creati 6 array di dimensione nell'ordine dei milioni

- 2 di Float
- 2 di SoftFloat a 32 bit
- 2 di Posit[32,3]

I primi due vengono popolati con numeri generati in modo pseudo-casuale, mentre i rimanenti 4 vengono riempiti convertendo, nei rispettivi formati, i numeri ottenuti precedentemente. Per poter ottenere i tempi è stata sfruttata la funzione clock_gettime(), la quale restituisce il timestamp di sistema. Salvando il timestamp ad inizio e fine elaborazione, durante il quale vengono eseguite le moltiplicazioni fra i due array, si riescono ad ottenere per differenza i tempi di lavoro per ciascun tipo di numero.

2.2.2 Conclusioni

Come potevamo aspettarci i Float con supporto hardware, sono stati ≈ 11 volte più veloci dei SoftFloat o dei Posit. Per quanto riguarda i Posit invece, sono stati più lenti di ≈ 2 volte rispetto ai SoftFloat. Anche questo risultato era prevedibile, in quanto i Posit non sono altro che una generalizzazione dei Float, avendo aggiunto dei campi a lunghezza variabile.

2.2.3 Codice

```
1 //Funzione per calcolo tempi Posit
2 double doTestPosit(unsigned int *X,unsigned int *Y,unsigned long n)
3 {
4
5         auto p1 = Posit(32, 3);
6         auto p2 = Posit(32, 3);
7
```

```
 8           timespec start, stop;
 9
10           clock_gettime( CLOCK_REALTIME, &start);
11
12           for(unsigned long i=0;i<n;i++){
13                   p1.setBits(X[i]);
14                   p2.setBits(Y[i]);
15
16                   p1.mul(p2);
17           }
18
19           clock_gettime( CLOCK_REALTIME, &stop);
20
21           double res = BILLION*( stop.tv_sec - start.tv_sec ) +
22           ( stop.tv_nsec - start.tv_nsec );
23
24           return res;
25
26 }
```

3 Codice?

3.1 Test Sigmoide

```
1          //Prova.cpp
2          printf("Ciao");
```

Bibliografia

[1] Boh. *"Beyond Floating Point"*. URL: https://github.com/libcg/bfp.

[2] John R. Hauser. *"Berkeley SoftFloat"*. URL: http://www.jhauser.us/arithmetic/SoftFloat.html.

[3] École polytechnique fédérale de Lausanne. *GSN*. URL: https://github.com/LSIR/gsn.

[4] Mario. "ciao". In: *Giornale* (2017).