

Tirocinio su Posit

Luigi Leonardi

Indice

1	Introduzione	2
1.1	Posit	2
1.2	Note sui Test	2
2	Test su Moltiplicazione	3
2.1	Accuratezza	3
2.1.1	Svolgimento	3
2.1.2	Conclusioni	3
2.1.3	Codice	4
2.2	Tempi	5
2.2.1	Svolgimento	5
2.2.2	Conclusioni	5
2.2.3	Codice	5
3	Test su Sigmoidale	7
3.1	Introduzione	7
3.2	Valori	7
3.2.1	Svolgimento	7
3.2.2	Conclusioni	8
3.2.3	Codice	8
3.3	Tempi	10
3.3.1	Parte 1	10
3.3.2	Parte 1 - Esito	11
3.3.3	Parte 2	11
3.3.4	Parte 2 - Esito	11
3.3.5	Parte 3	11
3.3.6	Parte 3 - Esito	11
3.3.7	Codice	11
4	Divisione e Moltiplicazione mediante Regime	12
4.1	Introduzione	12
5	Codice?	12

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,

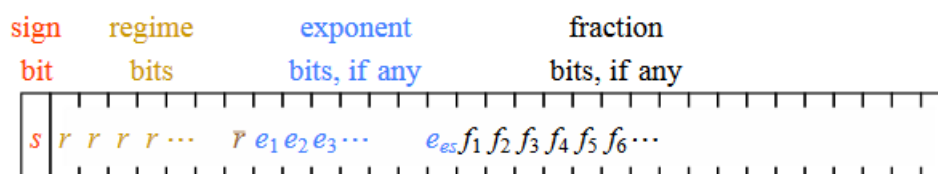


Figura 1: "Formato Posit"

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'esponente¹, 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 esponente³, dove non specificato diversamente.

¹L'esponente è l'unico campo ad avere una dimensione fissa

²Regimi che iniziano per 0 sono negativi, per 1 invece sono positivi. Lo 0 è rappresentato come 10

³Ha senso non utilizzare bit di esponente, in quanto si sfrutta il super esponente

2 Test su Moltiplicazione

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 riportati in double e ne è stata fatta la differenza, in modulo, rispetto al riferimento.

Definisco:

$$\begin{aligned}\Delta_i(\text{Posit}) &= |\text{PositRes}_i - \text{DoubleRes}_i| \\ \Delta_i(\text{Float}) &= |\text{FloatRes}_i - \text{DoubleRes}_i|\end{aligned}$$

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 generateNumber(){
3     double tmp = (double)rand()/(double)RAND_MAX;
4
5     return (tmp*2000);
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 Test su Sigmoide

3.1 Introduzione

Una caratteristica dei Posit con 0 bit di esponente, come osservato da Isaac Yonemoto, è che risulta molto facile e conveniente a livello computazionale, calcolare la funzione sigmoide, infatti bastano dei semplici shift e not. Essa trova largo impiego nell'ambito delle reti neurali e machine learning.

Per poter eseguire i seguenti test, è stato necessario implementare una funzione di conversione da Posit[32-0] a Float, in quanto la libreria ne risulta essere sprovvista.

3.2 Valori

Il primo test che sono andato ad eseguire è stato di tipo numerico, ossia data la funzione sigmoide $f(x) = \frac{1}{1+e^{-x}}$, ne ho confrontato i risultati con quelli restituiti dalla sigmoide ottenuta mediante manipolazione dei bit.

3.2.1 Svolgimento

Per questo tipo di test ho generato un numero di Float, nell'ordine delle migliaia, in modo pseudo-casuale nel range -30 e 30, e ne ho calcolato la sigmoide sfruttando prima la funzione con l'esponenziale, e successivamente, dopo aver convertito il numero in Posit, l'altra. Infine ho plottato tutti i risultati su MatLab ed ho ottenuto il seguente grafico:

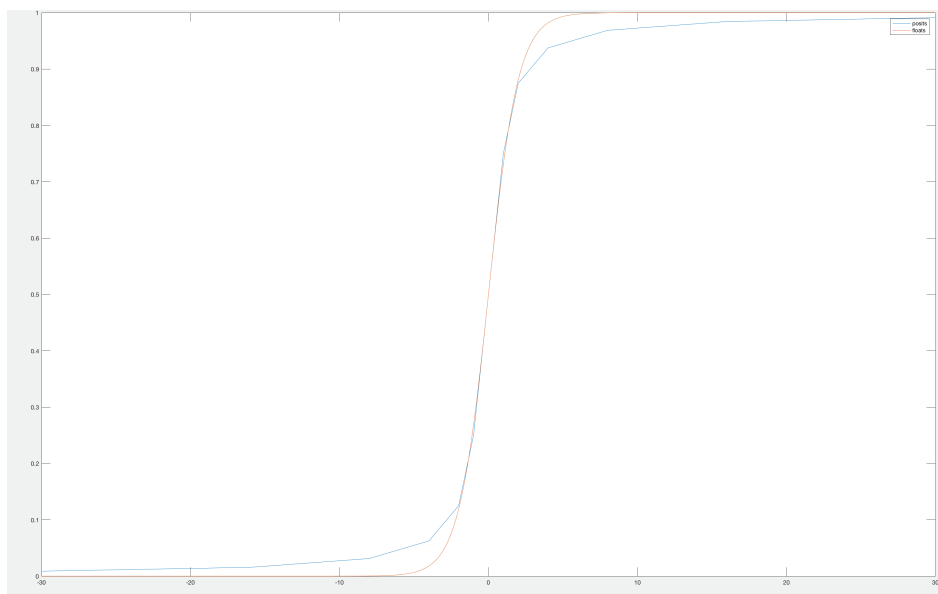


Figura 2: "Sigmoide"

3.2.2 Conclusioni

Come si può evincere dalla figura, il grafico ottenuto con i Posit sembra essere una spezzata che segue l'andamento della sigmoide con i Float, inoltre i due grafici si vanno a sovrapporre nell'origine. In generale comunque possiamo affermare che effettivamente questo tipo di manipolazione sui bit, ha fornito una approssimazione della sigmoide.

3.2.3 Codice

```
1 //Sigmoide con Float
2 float sigmoid(float &x){
3     return 1/(exp(-x)+1);
4 }
5
6 //Sigmoide con Posit
7 unsigned int sigmoid(unsigned int bits){
8     bits = bits ^ 0x80000000;
9     bits = bits >> 2;
10    return bits;
11 }
12
13
14 //Conversione da Posit[32-0] a Float
15 float Posit::subconv(){
16     union {
17         float f;
18         uint32_t bits;
19     };
20     bits=0;
21
22     if((mBits & 0xFFFFFFFF) == 0)
23         return 0.0f;
24     signed char c = (signed char)(regime());
25     c+= 127;
26     unsigned int esponente = (unsigned int)(c);
27
28     esponente = esponente<<23;
29     bits = bits | esponente;
30     unpacked_t aup = unpack_posit(mBits, mNbits, mEs);
31     unsigned int fr_ = aup.frac;
32
33     fr_ = fr_>>9;
34     bits = bits | fr_;
```



```
35     int segno = aup.neg;
36     if(segno == 0)
37         bits &= 0x7FFFFFFF;
38     else
39         bits |= 0x80000000;
40
41     return f;
42 }
```

3.3 Tempi

Una volta dimostrato che possiamo approssimare la sigmoide, il passo successivo è quello di provare che effettivamente vi sia risparmio tangibile nell'eseguire i calcoli.

Questo test, suddiviso in più parti, effettua diversi confronti, sui tempi di elaborazione, fra sigmoide su Posit e sigmoide ottenuta in vari modi sui Float.

3.3.1 Parte 1

Il primo confronto eseguito è stato con la funzione sigmoide esponenziale $f(x) = \frac{1}{1+e^{-x}}$, la stessa utilizzata nel test numerico.

Esso consiste nel generare un milione di Float pseudo-casuali e di scriverli su file sia come Float che come Posit sotto forma di bit, il tutto per evitare di falsare il test a sfavore di questi ultimi, in quanto nel tempo di elaborazione vi sarebbe anche il tempo di conversione.

Una volta acquisiti i dati da file, per ciascun tipo, impiego la funzione `clock_gettime()`, la quale viene richiamata prima e dopo aver finito di calcolare la sigmoide su tutti i dati.

Successivamente ho ripetuto lo stesso esperimento, incrementando gradualmente il numero di elementi, per verificare se l'eventuale margine fosse influenzato dalle dimensioni, nello specifico lo step scelto è stato di 10 mila, con numero massimo di 5 milioni.

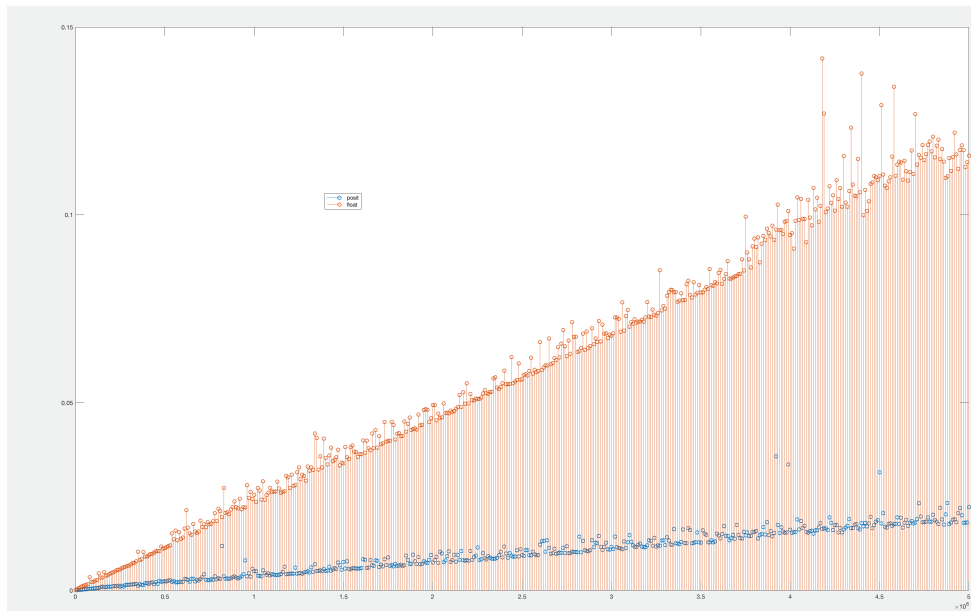


Figura 3: "Tempi all'aumentare degli elementi"

3.3.2 Parte 1 - Esito

I risultati hanno evidenziato che la funzione, calcolata mediante manipolazione di bit, quindi impiegando solamente la ALU del processore, è più rapida di $\approx 1.5 - 1.8$ volte su un campione di 1 milione di elementi.

Per quanto riguarda l'incremento, come si può evincere dalla Figura 3, il tempo di elaborazione aumenta linearmente all'aumentare del numero degli elementi in entrambi i casi, di conseguenza possiamo affermare che il vantaggio rimane pressoché costante.

3.3.3 Parte 2

In questa seconda parte ho voluto usare come metro di paragone, invece della curva logistica su Float, una sua versione approssimata ottenuta mediante quanto descritto nell'articolo "A Fast, Compact Approximation of the Exponential Function"[4]. In particolare si sfrutta il fatto che, i Float, utilizzano una rappresentazione del tipo 2^x per ottenere e^x , il tutto operando semplici operazioni sui bit ed una divisione come fattore correttivo.

3.3.4 Parte 2 - Esito

I risultati mostrano che, nonostante l'aver adoperato questo tipo di approssimazione, risulta essere ancora conveniente l'utilizzo dei Posit.

3.3.5 Parte 3

Test con x/boh

3.3.6 Parte 3 - Esito

3.3.7 Codice

4 Divisione e Moltiplicazione mediante Regime

4.1 Introduzione

Lavorando con Posit con 0 bit di esponente, il "superesponente" $(2^{2^{es}})^{regime}$ viene notevolmente semplificato, in quanto $2^{2^0} = 2 \Rightarrow 2^{regime}$ esattamente come nei Float. L'unica differenza consiste nel fatto che il regime non è un campo a dimensione fissa.

L'idea è stata quella di sfruttare questa caratteristica dei Posit 0, per implementare moltiplicazioni e divisione, per potenze del 2, semplicemente intervenendo sul regime. Il grande vantaggio consiste nel fatto che essendo esso composto da una sequenza di bit uguali, incrementarne o decrementarne il valore è relativamente semplice e poco costoso!

Ad esempio, dato un Posit-0: 0 001 00000⁴, il quale rappresenta +0.25, se volessimo dividerlo per 2, basterebbe aggiungere uno 0 al regime, ottenendo quindi 0 0001 0000⁵ = +0.125

5 Codice?

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

⁴Regime = -2; Mantissa = 0

⁵Regime = -3; Mantissa = 0

Bibliografia

- [1] Boh. "*Beyond Floating Point*". URL: <https://github.com/libcg/bfp>.
- [2] John R. Hauser. "*Berkeley SoftFloat*". URL: <http://www.jhauser.us/arithmetric/SoftFloat.html>.
- [3] École polytechnique fédérale de Lausanne. *GSN*. URL: <https://github.com/LSIR/gsn>.
- [4] Nicol N. Scharaudolph. "A fast, Compact Approximation of the Exponential Function". In: (1998).