

Posit Number → Permettono solo due eccezioni: 0 e Not a Real (NaN)

È una rappresentazione per numeri reali inventata da Gustafson nel 2017.

Un posit number ha 2 parametri di configurazione che sono **nbits** e **esbits**.

Il loro formato è di lunghezza fissata (**nbits**), composto da 4 campi:

1) **Sign field**: 1-bit (Positive - Negative, 0 = Positive, 1 = Negative)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	Regime(1..nbits)																											Exponent (0..esbits)		Fraction (0...)	

2) **Regime field**: lunghezza variabile (al più nbits).

Figure: Illustration of a posit<32, 11> data type.

nbits
esbits

3) **Exponent field**: Al più composta da es bits.

• nbits: n° bits usati dal posit

4) **Fraction field**: lunghezza variabile (mantissa)

• esbits: n° bits exponent field

Decoding da Posit a Real Number

Dato un posit descritto su <nbits, esbits> rappresentato da un intero X, siano e ed f rispettivamente i valori dell'esponente e della frazione, sia k il valore

encoded dal regime, il real value r rappresentato dall'encoding è:

$$r = \begin{cases} 0, \text{ if } X = 0 \\ \text{NaN}, \text{ if } X = -2^{(nbits-1)} \\ \text{sign}(X) \times \text{usecd}^k \cdot 2^e \cdot (1 + f), \text{ otherwise } \end{cases}$$

$$\rightarrow \text{usecd} = 2^{\frac{\text{esbits}}{2}}$$

Attention: e ≠ esbits

Regime Field

È un campo di lunghezza variabile (da 1 a nbits-1 bits), il quale è encoded

e decoded secondo un run-length approach:

• Il valore **l** rappresenta il numero di bit identici successivi terminali da un bit opposto.

→ esempio: 11110 → l=4

• Se la sequenza è composta da 0 allora **b=0**, altrimenti **b=1**

$$k = \begin{cases} -1, \text{ if } b = 0 \\ l - 1, \text{ otherwise } \end{cases}$$

→ k = valore rapp. dal regime

• Data l, il valore di k è dato da:

Decoding Steps

La decoding operation consiste nel:

- 1) Controllare se il sign bit è 1, in tal caso bisogna fare il complementare binario di tutti gli altri bits (cioè se 0 → 1 e se 1 → 0)
- 2) Fare il decode del regime ottenendo il κ value (dopo aver trovato κ i regime bits possono essere ignorati).
- 3) Se ci sono bit rimanenti, i prossimi esbits rappresentano l'interv (con segno) per l'esponente e
- 4) Se ci sono ancora bit rimanenti, si interpretano come mantissa bits (f).

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1001110111011001
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
S R E F
1 1 1 0 0 0 1 0 0 0 1 0 0 1 1 1

Figure: An example of a 16-bit Posit with 3 bits for the exponent (esbits=3). Given the sequence on top of the figure, after detecting it starts with one 1, we have to compute the 2's complement of all the remaining bits (passing from 001-110-111011001 to 110-001-000100111). Then we can proceed to decode the posit. The associated real value is therefore: $-256^1 \cdot 2^{-3} \cdot (1 + 39/512)$. The final value is therefore $-512 \cdot (1 + 39/512) = -551$ (exact value, i.e., no rounding, for this case).

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
S R E
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1

Figure: Another example of Posit16,3 that is 16-bit Posit with esbits=3. The associated real value is: $+256^{12} \cdot 2^{-4} \cdot (1 + 0)$. This example allows to clarify that: i) the fractional part can be missing, ii) the exponent field can be shorter than its maximum size (in that case the missing bits are assumed zero: the exponent 1 comes from the reconstructed exponent field 100).

Posit ranges e Posit ring

Dato un posit $\langle N, \epsilon \rangle$ definiamo:

- $useed = 2^{2E}$
- $maxposit = useed^{N-2}$ = Max representable positive real number
- $minposit = minpos = \frac{1}{maxpos}$ = Min representable positive real number

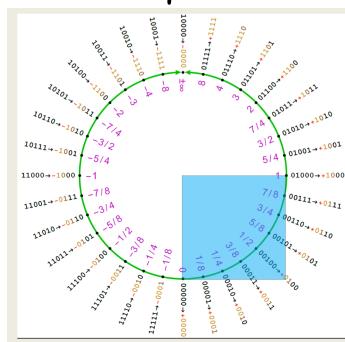
Per definizione, i posit numbers possono essere proiettati su un cerchio chiamato "posit ring".

Nel posit ring possiamo osservare alcuni punti chiave:

posit $\langle 5, 0 \rangle$

- Lo 0 alle 6 in punto
- Lo -1 alle 3 in punto e -1 alle 9 in punto
- NaN (inf) alle +2 in punto

Osservazioni :



- 1) Metà del posit ring è nel range $[-1, 1]$, il quale rappresenta la zona più densa.
Questo vuol dire che i numeri nell'intervallo $[-1, 1]$, la rappresentazione posit fornisce una maggior precisione rispetto ad altri formati.
- Questo vuol dire che per numeri in questo intervallo, il posit fornisce risultati più accurati rispetto alla rapp. floating point, con lo stesso numero di bits (interessante per il machine learning field).

Attenzione: muovendosi sempre più lontani dallo 0, la precisione diminuisce perché la differenza tra i numeri diventa maggiore. (trade-off necessario)

Posit vs IEEE floats

Precision: IEEE fornisce una precisione costante, mentre il posit maggiore vicino lo 0 è minore altrove

Format: IEEE format è fissato (predeterminato num. di bits per ogni field), il posit è configurabile

Wasted bits: IEEE ha 8 milioni di pattern inutili (nan, inf o duplicati), mentre il posit non pattern inutili, ognuno rapp. un differente numero.

0 exponent case

Una configurazione posit particolare è la $\langle X, 0 \rangle$.

In questo caso la posit formula vista precedentemente può essere semplificata

dove:

$$x = 2^k \cdot (1 + \phi \cdot 2^{-F})$$

- ϕ è il fraction field
- F è il fraction length
- k dipende dalla lunghezza del regime (l).

- Example: let's take the number 0.375, its posit representation is 00011000. The correspondent fixed-point target is a 16-bit integer (1-bit for sing, 7 bits for integer part and 8 bits for fractional part). The value 0.375 for a 16-bit fixed point is 0000000.01100000. If we shift the posit representation by two position left and we pad it to reach 16-bit we get the fixed-point representation.

$k = -l$ per $x < 1$ (anche scritto come \bar{x}) e $k = l - 1$ per $x \geq 1$ (x^+).

cioè $x \in [-1, 1]$

Quando $k < 0$ questa formulazione richiama quella del fixed-point format, cioè se aggiustiamo la rappresentazione con un left shift di due bits, otteniamo la stessa rapp. del fixed-point.

Application: Richiede esbits = 0 !! → per fast inverse si intende velocizzare l'operazione $\frac{1}{x}$

1) fast inverse: Possiamo usare il posit per implementare un versione veloce dell'inverse function.

a) Dato x , vogliamo trovare y t.c. $x \cdot y \approx 1$, cioè $y = \frac{1}{x}$

b) x e y rappresentati in posit sono:

$$x = 2^{k_x} \cdot (1 + f_x \cdot 2^{-F_x})$$

$$y = 2^{k_y} \cdot (1 + f_y \cdot 2^{-F_y})$$

calcolare l'inversa senza decodificare ma direttamente sulla posit representation

Quindi vogliamo che

$$x \cdot y = 2^{k_x+k_y} \cdot (1 + f_x \cdot f_y \cdot 2^{-2F_x} + (f_x + f_y) \cdot 2^{-F_x}) = 1$$

c) Possiamo notare che $f_x \cdot f_y \cdot 2^{-2F_x}$ posso ignorarlo perché piccolo e $k_x + k_y = -k_x$,

quindi mi rimane:

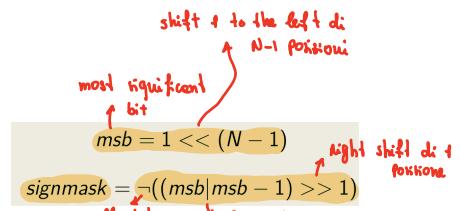
$$1 + (f_x + f_y) \cdot 2^{-F_x} = 2 \quad \text{cioè} \quad f_y = 2^{F_x} - f_x \quad \rightarrow \text{Uguale a fare il complementare di } f_x$$

Questa operazione corrisponde a

$$\text{invert } \Rightarrow X \rightarrow Y = X \oplus (\text{signmask})$$

XOR
bit of the posit

dove la signmask può essere ottenuta come segue:



Tutte queste operazioni possono essere fatte direttamente sulla rapp. posit (senza decoding),

usando solo integer operations sull'ALU.

Altre operazioni "decoding-free" sono

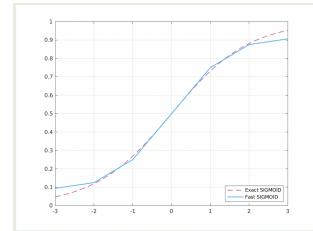
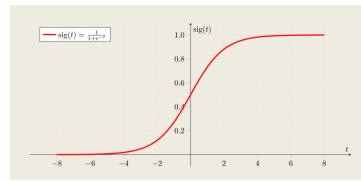
Operation	Approximated	Requirements
$2 \cdot x$	no	esbits=0
$x/2$	no	esbits=0
$1/x$	yes	esbits=0
$1 - x$	no	esbits=0, $x \in [0, 1]$

2) Fast sigmoid: La sigmoid è interessante perché il suo output (0,1) finisce interamente nella metà parte più densa del posit ring (mappa ogni real value in (0,1))

- Original implementation provided by John L. Gustafson and I. Yonemoto in the original posit "standard". Given X the representation of a posit with 0 exponent bits, the approximated result of the sigmoid function applied to that posit is represented by Y .

$$Y = ((1 \ll nbits - 1) + (X \gg 1)) \gg 1$$
- Our refined version adjusts behaviours around 0 and at the domain extremes. Given X the representation of a posit with 0 exponent bits, the result of the sigmoid function applied to that posit is represented by Y .

$$Y = ((1 \ll nbits - 1) + X + 2) \gg 2$$



Operano direttamente sulla rappres. posit binaria, fornendo un'approximazione della sigmoid function.

3) Fast hyperbolic tangent: Un'interessante perché il suo output finisce in [-1,1], massimizzando la posit coverage nell'area più densa del posit ring

- Let's look at the hyperbolic tangent expression

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

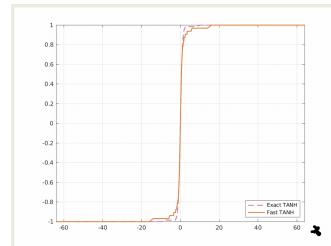
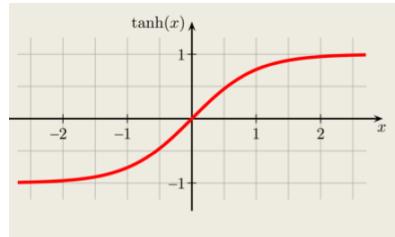
- Let's look now at the sigmoid one

$$\text{sigmoid}(x) = \frac{1}{e^{-x} + 1}$$

- We can easily note that we can scale and shift the sigmoid to match the tanh:

$$\tanh(x) = -(1 - 2 \cdot \text{sigmoid}(2 \cdot x))$$

- If we substitute the sigmoid with its fast version, we obtain the fast version of the tanh

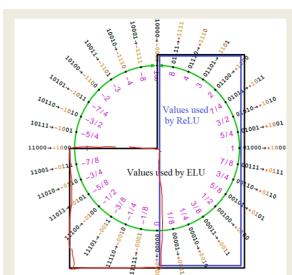


Ottieniamo un'approximazione delle tanh =

4) Fast ELU: R/ELU-like function aiutano a risolvere il problema che hanno le s-shaped functions, cioè il vanishing gradient problem.

Inoltre, il ReLU output sta tra [0, +∞) mentre l'ELU output sta in [-α, +∞).

Aggiungendo i valori negativi, ELU aggiunge una parte con molti encoding



$$\begin{aligned} \text{Sigmoid}(-x) &= \frac{1}{1 + e^x} & (3) \\ 1/\text{Sigmoid}(-x) &= 1 + e^x & (4) \\ 1/(2 \cdot \text{Sigmoid}(-x)) &= \frac{1 + e^x}{2} & (5) \\ 1/(2 \cdot \text{Sigmoid}(-x)) - 1 &= \frac{1 + e^x}{2} - 1 = \frac{e^x - 1}{2} & (6) \\ * [2 \cdot [1/(2 \cdot \text{Sigmoid}(-x)) - 1]] &= e^x - 1 & (7) \end{aligned}$$

$$\begin{aligned} \text{ReLU}(x) &= \begin{cases} 0, & \text{if } x \leq 0 \\ x, & \text{otherwise} \end{cases} \\ \text{ELU}(x) &= \begin{cases} \alpha \cdot (e^x - 1), & \text{if } x \leq 0 \\ x, & \text{otherwise} \end{cases} \end{aligned}$$

Posso sostituire e^{-x} e usare la fast sigmoid per ottenere una fast approx. delle ELU function

Ottimizzare le Neural Network per la massima efficacia di posit

Il nostro obiettivo è massimizzare l'uso del range $[-1,1]$ in tutti i NN layers.

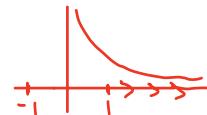
Per fare ciò dobbiamo tenere di conto di 4 aspetti:

- 1) Activation Functions: le funzioni che coprono il range $[-1,1]$ sono ottime per i posit numbers.

Quelle interamente contenute sono ideali, ma c'è da far attenzione al gradient vanishing problem.

- 2) Distribution of values: Attraversando diversi layers, i valori tendono a shiftare verso destra allontanandosi dal range $[-1,1]$

Soluzione: Normalization layers



- 3) Loss strategies: Penalizzare valori grandi utilizzando la regolarizzazione, ad esempio l' L2 regularization

$$R(w) = \lambda \cdot \frac{1}{2} \cdot \|w\|_2^2 \rightarrow \text{riduce overfitting}$$

- 4) Data preprocessing:

Anche il valore numerico dei dati dovrebbe essere portato nel range $[-1,1]$.

- Se vogliamo ad esempio utilizzare delle immagini possiamo fare il re-scale dei pixel da 255 a $[-1,1]$, però può comportare a una potenziale perdita di informazioni.
- Un altro modo per fare il re-scale dei pixel values minimizzando la perdita di informazioni può essere ottenuto usando un altro key-point nel posit ring, cioè lo $useed = \pm 2^{2^{esbits}}$.



$$n(p) = 2 \cdot useed \cdot \frac{p}{255} - useed$$

l'obiettivo è di portare il valore dei pixel nel range $[-useed, useed]$, con questa formula, il quale usa il dynamic range dei points e garantisce che la potenza di rappresentazione dei posit sia ben usata. \rightarrow permette di ridurre la perdita di info ma usare comunque bene la rapp. posit.

Posit implementation in CPP

Vediamo come poter implementare il posit format.

Backend

Essendo un formato nuovo, gli manca il supporto HW, cioè nei general purpose processors non ci sono istruzioni dedicate per le operazioni aritmetiche posit o HW dedicato.

Possiamo emulare queste operazioni sui classici backend:

- 1) ALU: Reinterpreta il posit come un intero e svolge le operazioni aritmetiche sugli interi
- 2) FPU: Converte il posit in un 32-bit float e usa il processore della FPU per svolgere le operazioni
- 3) Fixed-Point: Converte posit in una fixed-point representation e usa la big-integer arithmetic per svolgere le operazioni.

Un'idea alternativa può essere fare la pre-computation delle posit operation e salvarle in delle tabelle in modo da evitare la conversione → problema: spazio!

Dati: nbits posit, una tabella per un'operazione binaria richiederebbe $2^{nbits-1}$ righe e colonne, sia $b = \text{sizeof}(T)$ bits dove $T = \text{storage type}$ e la necessità di, tipicamente, 8-10 tabelle, otteniamo che lo spazio occupato è dato dalla seguente formula:

$$S = N \cdot (R \cdot C) \cdot b$$

$\downarrow_{2^{nbits-1}}$ $\downarrow_{2^{nbits-1}}$

Total bits (X)	Storage type bits (b)	Occupation
8	8	64 KB
10	16	2 MB
12	16	32 MB
14	16	512 MB
16	16	8 GB ↗ <small>Foto 3</small>

Possiamo però sfruttare alcune proprietà delle operazioni per ridurre la dimensione delle tabelle:

- 1) Addizione e sottrazione possono essere unite nella stessa tabella e usare solo metà di essa grazie alla symmetry property.
- 2) Moltiplicazione e divisione possono essere convertite a operazioni di somma e sottrazione usando

i logaritmi.

$$p = x \cdot y \rightarrow \log(p) = \log(x \cdot y) \rightarrow \log(p) = \log(x) + \log(y) \rightarrow p = e^{\log(x) + \log(y)}$$

Le operazioni di log ed esponenziale essendo unary operations richiedono tabelle più piccole non scalando quadraticamente con la point size.

→ **Attention:** Nel caso delle point operation con 0 esbits posso ottimizzazione ulteriormente la moltiplicazione e divisione usando la fast-inverse salvando in questo modo solo $x \cdot y$ e x/y per ogni coppia x, y .

Operation levels

Le point operations possono essere classificate in 4 livelli.

L1) Contiene le operazioni più veloci che non necessitano del decoding del point e sono implementate direttamente usando le ALU operations.

Operation	Approximated	Requirements
$2 \cdot x$	no	esbits=0
$x/2$	no	esbits=0
$1 - x$	no	esbits=0, $x \in [-1, 1]$
$1/x$	yes	esbits=0
FastSigmoid	yes	esbits=0
FastTanh	yes	esbits=0
FastELU	yes	esbits=0

L2) le operazioni L2 richiedono il decoding ma senta richiedere il decoding completo dell'esponente.

le operazioni sono eseguite sui fields (segno, regime+exp, frazione) e non sul real value.

Additional computational cost per il decoding ed encoding del point

L3) L'esponente e regime sono fully decoded, le operazioni sono eseguite sui fields del point e non sul real value

L4) Il point viene convertito in uno dei backends format, viene eseguita l'operazione e il risultato riconvertito in point.

Vectorization

La vectorization consiste nel svolgere un'istruzione su diversi dati in parallelo (single instruction, multiple data stream - SIMD).

Per fare ciò abbiamo bisogno che le functional units (es: ALU) nel processore siano replicate più volte.

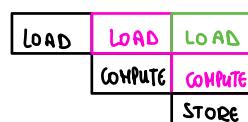
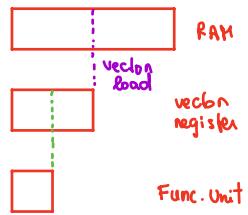
In questo modo più elementi del vettore possono essere usati da questo insieme di functional units che eseguiranno la stessa istruzione indipendentemente.

Dalla RAM vengono caricati nel vector register e poi nella functional unit

Essendo la functional unit più piccola rispetto al vector register posso velocizzare

facendo in parallelo le operazioni di load / store / compute

la maggior parte dei processori moderni hanno le vector units (SIMD units):



1) Intel con "Streaming SIMD" e AMD con "AVX/2" supportano 512-bit vectors

2) ARM con "NEON" supporta 128 bit vectors

3) ARM SVE e RISC-V "V" sono chiamati "length agnostic", cioè la vector size non è pre-determinata ma viene conosciuta a run-time. Questo ha il vantaggio di aumentare la portabilità del codice.

Nel contesto delle DNN la vectorization è particolarmente importante per velocizzare operazioni come la matrix-matrix multiplication, matrix-vector multiplication, activation functions ecc.. fornendo una versione vettorizzata di queste operazioni.

Problema: Non c'è un supporto diretto per i punti nei vectorized environment, dobbiamo fornire una specifica implementazione su come i punti debbano essere gestiti nel caso di operazioni vettorizzate (su SIMD processors).

Vectorized Posit operation

Per implementare la vectorized version delle operazioni posit dobbiamo:

1) Preparare i dati - Prologo

Preparare i dati per essere dati al SIMD engine.

Per le operazioni L1 vuol dire preparare il vettore con i signed integer che rappresenta il posit, quindi

non necessitano di conversione. $[x_1, x_2, \dots, x_n]$ dove $x = \text{signed integer}$ che rapp. un posit number

Per le operazioni L4 necessitiamo di un vectorized decoding e conversion algorithm da posit a fixed point o floating point.

2) Body

Il body contiene tutte le funzioni logiche e aritmetiche necessarie per applicare l'operazione.

Incluse sia integer che floating point vector instructions, dipendentemente dall' operational level della funzione che stiamo implementando.

Naturalmente le istruzioni usate sono disegnate per lavorare efficientemente sui processori SIMD.

3) Epilogo

Necessitiamo di ricomporre il posit dal result vector ottenuto dal function body.

- Se l'operazione era L1 \rightarrow Dobbiamo passare dall' INT al posit senza conversione
- Se l'operazione era L4 \rightarrow Dobbiamo passare dal FP32 al posit con conversione

Nel caso L2 e L3 vengono estratti i field individuali dal posit number e organizzati nei SIMD vectors (le operazioni del body vengono applicate sui singoli field e poi nell' epilogo viene ricomposto il posit number partendo dai field value individuali)

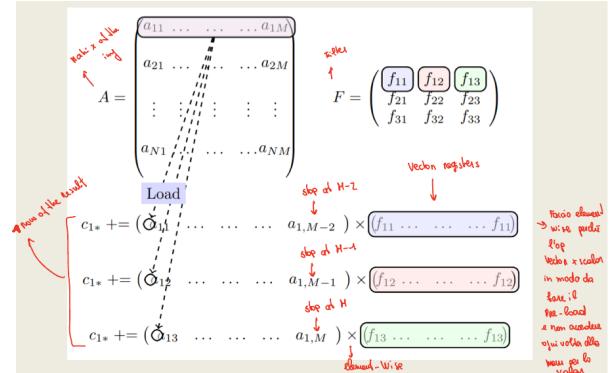
Vediamo adesso la vectorized version di qualche operazione:

1) 3×3 convolution with stride = 1

- We want to implement a 3×3 convolution with stride = 1.
- A standard algorithm would take the 3×3 filter and compute the product each time moving the filter along the image
- In a vectorization environment we need to rethink this algorithm:
 - We need to maximize register utilization (vector registers are like a super-fast cache)
 - We must minimize loads from memory that may stall the vector pipeline.

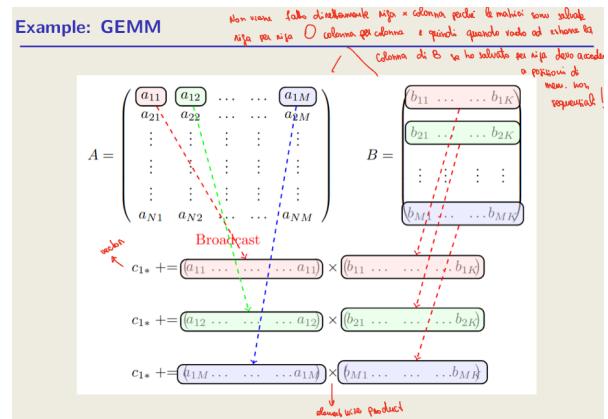
Example: 3×3 Convolution - Solution

- We take the 9 filter elements and replicate each of them on the vector registers
- Vector registers that contains filter elements on the second column are shifted right of 1 position
- Similarly for the vector registers containing filter elements on third column
- We loop on the image rows, loading 3 rows at the same time and multiplying the correspondent filter rows by the whole lines. In this way we compute the whole contribution of a single image line to the convolution result
- Finally we sum the convoluted rows and columns to obtain the convolution results.



2) General matrix-matrix multiplication (GEMM)

- We want to implement a typical matrix-matrix multiplication
- The standard algorithm with 3 nested loops is not highly vectorizable
- We try to maximize register utilization and minimize memory access



Tiny DNN

è un C++ framework per reti neurali, opensource, AVX vectorization.

Alcuni problemi di questo framework sono:

- Difficile implementare le non-feedforward networks
- Non usa la GPU (cuda)
- Supporto discontinuo

Questo framework è stato esteso dai prof. per l'utilizzo dei posit numbers, aggiungendo un supporto parallelo per il posit^{type} attivabile a run-time, permettendo un confronto in termini di inference time e accuracy dei pretrained model rispetto all'utilizzo dei float classici.

Workflow:

- 1) Fare il training della rete usando un formato con tanti bit (float32 o Posit<16,0>) per stabilire una baseline per le performance e network accuracy.
- 2) Convertire il modello trainato in un formato con meno bit (Posit<10,0> o Posit<8,0>) → weight conversion
- 3) Comparare l'accuracy e performance del modello rispetto alla baseline

Posit hardware Acceleration

Per competere con gli standardized numeric format abbiamo bisogno di HW dedicato al posit per velocizzare le posit operations.

La Risc-v architecture, essendo open-source, può essere estesa con istruzioni custom.

Risc-v ISA

La Risc-V ISA è un'architettura modulare composta da sottomoduli con responsabilità limitata.

Ci sono alcuni sottinsiemi di base (base integer instruction set, identificati con I) e alcuni sottinsiemi additionali:

- Integer Multiplication / division operation (M)
- Single precision floating point operations (F)
- Double precision floating point operations (D)
- Atomic Instructions (A)

RISC-V "V"

Versone di Risc-v che usa il **Vector length Agnostic paradigm**, il quale fornisce dei parametri per la vectorization.

Parametri:

- Numero di vector registers (standard 32)
- Lunghezza vector registers (fino a $\approx 16\text{K}$)
- Maximum element size

A un-time è possibile ottenere la "granted vector length" cioè il minimo tra la lunghezza richiesta e la lunghezza disponibile: $vgrant = \min(vlen, vreq)$

Per ottenere la $vgrant$ viene usato il comando "vsetvl" con in input la lunghezza richiesta "vreq"

RISC-V Posit Extension

Per poter sviluppare l'estensione posit dobbiamo fare delle decisioni: (b = risposte prefs)

1a) Quante operazioni includere nell'isa? instruction set architecture

1b) Light version, solo operazioni di compressione e decompressione

2a) Aggiungere registri o no?

2b) Riuso degli ALU registers per mantenere i posit, invece di aggiungere nuovi registri per i posit

3a) Dimensione posit da supporre?

3b) $\text{posit} < 16,1>$, $\text{posit} < 16,0>$, $\text{posit} < 8,0>$

4a) Quali backends usare?

4b) ALU, floating-point units e fixed-point (conversione Point \rightarrow FP32 e Point \rightarrow Fixed Point)

Note on fixed point

Il massimo valore per un posit $\langle n, e \rangle$ è $useed^{n-2}$ dove $useed = 2^{2^e}$, mentre per il fixed point il massimo valore su l bits è $2^{l/2}$.

Per assicurarsi che il fixed-point format possa contenere il massimo valore nel posit format, dobbiamo risolvere la seguente inequazione: $useed^{n-2} \leq 2^{l/2} \rightarrow 2 \cdot (n-2) \cdot 2^e \leq l$

Ottieniamo che per:

- Posit $\langle 8, 0 \rangle \rightarrow l = 16$
- Posit $\langle 16, 0 \rangle \rightarrow l = 32$
- Posit $\langle 16, 1 \rangle \rightarrow l = 64$

Questi valori di l assicurano che il fixed-point format abbia abbastanza bit per rapp. il valore massimo del posit format.

Instructions ($H=16$ bits, $W=32$ bits, $L=64$ bits)

- Floating point and posit conversions
 - FCVT.S.P8/FCVT.P8.S:
Float to/from posit $\langle 8, 0 \rangle$ conversion
 - FCVT.S.P16.0/FCVT.P16.0.S:
Float to/from posit $\langle 16, 0 \rangle$ conversion
 - FCVT.S.P16.1/FCVT.P16.1.S:
Float to/from posit $\langle 16, 1 \rangle$
- Fixed point and posit conversions
 - FXCVT.H.P8/FXCVT.P8.H:
 $fx(16)$ to/from posit $\langle 8, 0 \rangle$ conversion
 - FXCVT.W.P16.0/FXCVT.P16.0.W:
 $fx(32)$ to/from posit $\langle 16, 0 \rangle$ conversion
 - FXCVT.L.P16.1/FXCVT.P16.1.L:
 $fx(64)$ to/from posit $\langle 16, 1 \rangle$

- Posit to posit conversions
 - FCVT.P8.P16.0/FCVT.P16.0.P8:
posit $\langle 8, 0 \rangle$ to/from posit $\langle 16, 0 \rangle$ conversion
 - FCVT.P16.1.P16.0/FCVT.P16.0.P16.1:
posit $\langle 16, 1 \rangle$ to/from posit $\langle 16, 0 \rangle$ conversion
 - FCVT.P8.P16.1/FCVT.P16.1.P8:
posit $\langle 16, 1 \rangle$ to/from posit $\langle 8, 0 \rangle$ conversion

encoding example

RV64Xposit Posit Ext. Instruction Set					
1100000	00010	rs1	000	rd	0001011
1100000	00011	rs1	000	rd	0001011
1100000	00011	rs1	010	rd	0001011
1101000	00010	rs1	000	rd	0001011
1101000	00011	rs1	000	rd	0001011
1101000	00011	rs1	010	rd	0001011

Compiler support

Le nuove istruzioni dobbiamo farle supportare da C++.

Potremmo aggiungere il supporto nel compilatore, ma dovremmo fare il rebuild dell'intero GCC toolchain → Processo laborioso

Possiamo usare quindi un file che tramite la keyword `--asm--`, la quale permette di includere del codice assembly.

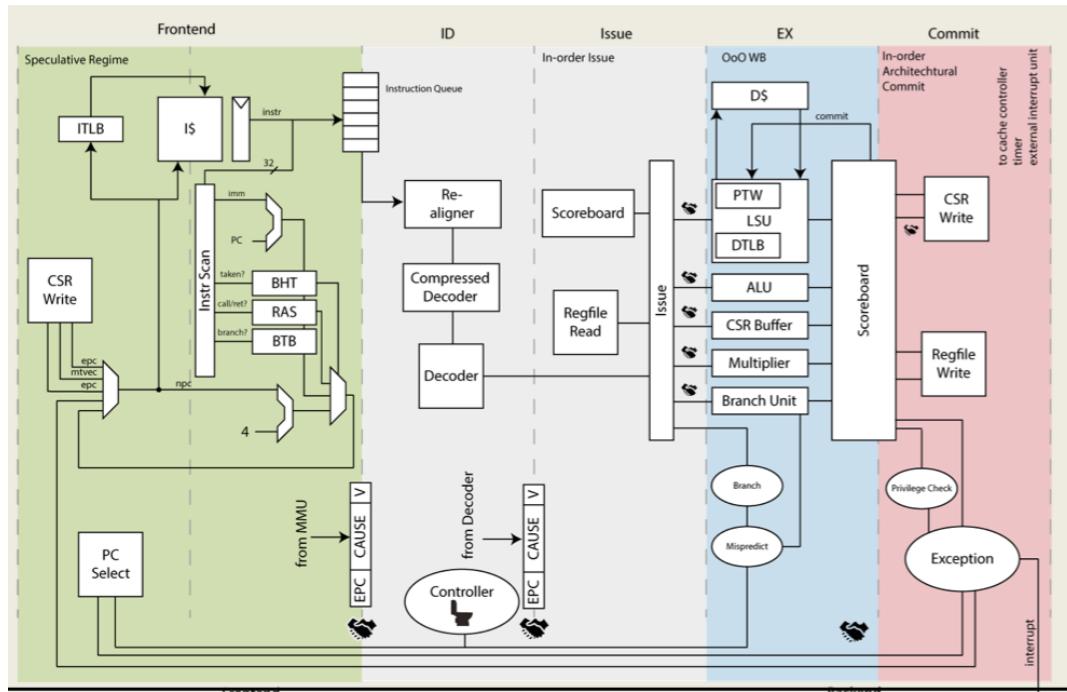
Il codice assembly per le istruzioni viene generato da uno script python e salvato in un C++ header (generato anch'esso dal python script) che contiene tutte le funzioni ad alto livello, utilizzate per eseguire le nuove istruzioni.

Hardware implementation

Per implementare le encoded instructions a livello HW possiamo usare l' ARIANE RISC-V core (opensource).

L' obiettivo è di inserire la point processing unit (PPU) nell' execution stage dell' ARIANE RISC-V core, accanto alla ALU unit.

La PPU svolgerà solo la compressione / decompressione dei float / Fixed point, non introduce nuovi registri ma manda quelli dell' ALU/FPU e supporterà al più 16 - bits points.



ARIANE RISC-V and architecture

Conversion da FP32 a Point <16,0>

1) Essendo che i point operano sul valore assoluto (non considerano il segno), ignoriamo il segno del FP32

e ci concentriamo sul convertire la magnitudo del numero

2) L'operazione più costosa a livello computazionale è convertire l'esponente nel regime.

Il posit k-value può essere ottenuto direttamente dal float exponent dopo avergli sottratto l'exponent bias, cioè il valore aggiunto al float exponent per assicurarsi che esso sia simmetrico attorno allo 0 ($FP32 \rightarrow 127$)

La regime length (l) è determinata da k:

$$k = \begin{cases} -1, & \text{if } b = 0 \\ l + 1, & \text{otherwise} \end{cases}$$

3) Il point <x,0> regime viene costruito shiftando uno specifico valore usando i $\log_2(x)$ bits meno significativi del FP32 normalized exponent.

Nel caso di point <16,0> shiftiamo l'intero (con segno) rappresentato da 2^{15} usando gli ultimi 4 digits del normalized float exponent.

4) La mantissa è ottenuta shiftando (a destra) di regime length volte la float mantissa

5) Il regime e la mantissa vengono combinati usando una bitwise "OR" operation

6) Il segno viene applicato usando la 2's complement notation

Decoding the Regime

Per trovare il primo bit diverso dal precedente per determinare la mantissa length, non viene usato un for loop (inefficiente) ma il Counting Leading Zero (CLZ) algorithm.

L'algoritmo consiste nel confrontare l'integer value con una serie di bit patterns, shiftando verso destra il valore e aggiornando la result variabile sulla base delle comparazioni.

caso 16 bit integers

```
function clz16 (x)
    r = (x > 0xFF) << 3; x >>= r;
    q = (x > 0xF ) << 2; x >>= q; r |= q;
    q = (x > 0x3 ) << 1; x >>= q; r |= q;
                                r |= (x >> 1);
    return r;
```

1. $x = (x > 0xFF) << 3$: Compares the value 'x' with the bit pattern '0xFF' (11111111 in binary). If 'x' is greater than '0xFF', sets 'x' to 8 3 bits shifted to the left. Otherwise, sets 'x' to 0.
2. $x >>= x$: Shifts the value 'x' right by 'x' bits.
3. $q = (x > 0xF) << 2$: Compares the updated 'x' value with the bit pattern '0xF' (1111 in binary). If 'x' is greater than '0xF', sets 'q' to 4 (2 bits shifted to the left). Otherwise, sets 'q' to 0.
4. $x >>= q$: Shifts 'x' right by 'q' bits.
5. $x |= q$: Bitwise OR operation to update 'x' with the value of 'q'.
6. $q = (x > 0x3) << 1$: Compares the updated 'x' value with the bit pattern '0x3' (11 in binary). If 'x' is greater than '0x3', sets 'q' to 2 (1 bit shifted to the left). Otherwise, sets 'q' to 0.
7. $x >>= q$: Shifts 'x' right by 'q' bits.
8. $x |= q$: Bitwise OR operation to update 'x' with the value of 'q'.
9. $x |= (x >> 1)$: Bitwise OR operation to update 'x' with the rightmost bit of 'x'.
10. Returns the final value of 'x', which represents the position of the first bit that differs from the previous bits.

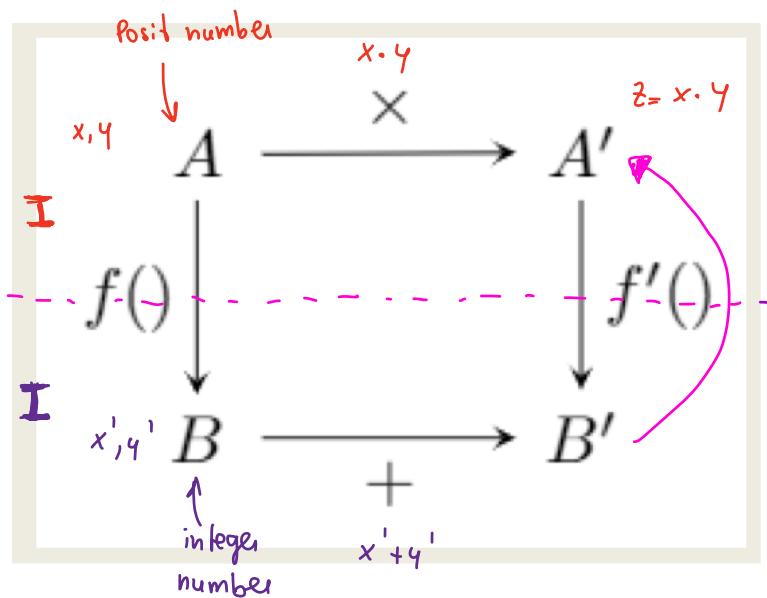
Enabling fast arithmetic operations in HX without decoding

Vorremmo fare real number operations usando la number representation senza nessun decoding.

Abbiamo visto che è possibile farlo, a volte, con posit con esbit a 0 convertendoli in fixed point numbers. Vogliamo però farlo in generale includendo tutti i possibili output dell'operazione.

L'idea è di usare le tabelle, ma invece di avere una singola tabella (es. 16×16) per l'operazione binaria, usiamo due mapping tables (da es. 16) che convertono dal posit domain all'integer domain.

da A a B e da A a B'



L'idea è quindi di:

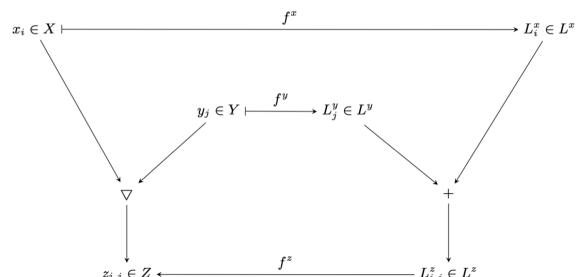
- Mappare ogni intero dell'input operand in un altro spazio di interi
- Scegliere un mapping tale che la somma nel nuovo spazio possa essere inversamente mappata alla corrispondente operazione binaria nello spazio originale
 - cioè: Invece di moltiplicare $a \cdot b$, li mappiamo in a', b' t.c la somma $a' + b'$ possa essere inversamente mappata in $a \cdot b$ senza il decoding di a e b .

Formalmente:

- X e Y sono due insiemi finiti di numeri reali (numeri reali)
- X^* e Y^* sono insiemi di stringhe binarie (0101...) che sono l'encoding di X e Y , quindi il mapping X, X^* e Y, Y^* è bijective
- ∇ è una qualsiasi operazione binaria tra un elemento di X e uno di Y
- Z è l'insieme di numeri reali t.c. $z_{ij} = x_i \nabla y_j \rightarrow$ risultato operazione reale
- \hat{Z} è l'insieme di numeri reali ottenuti amontonando z_{ij} per ottenere valori rappresentabili (reali) nel posit formato
- L^x e L^y sono insiemi ordinati di numeri naturali
- Ogni x è unicamente mappata in L^x (tramite f_x applicata a x^*) e ogni y è unicamente mappata in L^y (tramite f_y applicata a y^*)
- L^z è l'insieme di tutte le somme (distinte) tra L^x e L^y , e ogni valore in Z è mappato in L^z tramite f_z .

Se per ogni coppia x_i, y_j e x_p, y_q abbiamo $L_i^x + L_j^y \neq L_p^x + L_q^y$, otteniamo la seguente relazione:

$$z_{i,j} = x_i \nabla y_j = f^z(f^x(x_i) + f^y(y_j))$$



Nel mapping dobbiamo quindi assicurarsi che differenti risultati siano mappati in differenti somme in L^z (ma non necessariamente il contrario).

Dobbiamo quindi risolvere il seguente integer programming problem:

$$\begin{aligned} \min \quad & \sum_i L_i^x + \sum_j L_j^y \\ \text{s.t.} \quad & L_1^x \geq 0 \\ & L_1^y \geq 0 \\ & L_{i_1}^x \neq L_{i_2}^x \quad \forall i_1 \neq i_2 \\ & L_{j_1}^y \neq L_{j_2}^y \quad \forall j_1 \neq j_2 \\ & L_i^x + L_j^y \neq L_p^x + L_q^y \quad \forall i, j, p, q \text{ s.t. } x_i \nabla y_j \neq x_p \nabla y_q \\ & L_i^x, L_j^y \in \mathbb{Z} \quad \forall i, \forall j \end{aligned}$$

$$\begin{aligned} \min \quad & \sum_i L_i^x + \sum_j L_j^y \\ \text{s.t.} \quad & L_1^x \geq 0 \\ & L_1^y \geq 0 \\ & L_i^x \geq L_j^y + 1 \quad i > j \\ & L_i^y \geq L_j^x + 1 \quad i > j \quad]_{\text{monotonic}} \\ & L_i^x + L_j^y = L_p^x + L_q^y \quad \forall i, \forall j \rightarrow \text{comm.} \\ & L_i^x + L_j^y + 1 \leq L_p^x + L_q^y \quad \forall i, j, p, q \text{ s.t. } x_i \nabla y_j < x_p \nabla y_q \\ & L_i^x, L_j^y \in \mathbb{Z} \quad \forall i, \forall j \end{aligned}$$

trasformato per evitare l'ineq. constraint

che corrisponde a due problemi differenti

Se possiamo fornire una feasible solution iniziale al problema, sotto le giuste

assunzioni possiamo dire che avremo sempre una soluzione ottimale per esso.

Application to Posit(4,0)

- We apply the method presented until now to a 4-bit posit, for simplicity.
- We consider the four arithmetic operations $+, -, \times, /$
- We consider the strategies for the solution (i.e. ordering of the resulting L_x, L_y sets) \rightarrow we have to apply on $L_x \times L_y$ sets to ease the computation
- We evaluate the result, comparing it to a traditional 2D look-up table

Strategies for solution

	L^x	L^y
SUM	Increasing	Increasing
MUL	Increasing	Increasing
SUB	Decreasing	Increasing
DIV	Increasing	Decreasing

Optimal problem solution

operation	L^x	L^y
$+$	{0, 1, 2, 3, 5, 6, 11}	{0, 1, 2, 3, 5, 6, 11}
\times	{0, 2, 3, 4, 5, 6, 8}	{0, 2, 3, 4, 5, 6, 8}
$-$	{0, 1, 2, 3, 5, 6, 7}	{15, 14, 13, 12, 10, 8, 0}
$/$	{0, 2, 3, 4, 5, 6, 8}	{8, 6, 5, 4, 3, 2, 0}

Multiplication Example

- Let us take the multiplication results
- We have 3 ordered sets of real numbers
 - $X = Y = \left\{ \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1, \frac{3}{2}, 2, 4 \right\}$
 - $\hat{Z} = \left\{ \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1, \frac{3}{2}, 2, 4 \right\}$
- 3 ordered sets of natural numbers
 - $L_x = \{0, 2, 3, 4, 5, 6, 8\}$
 - $L_y = \{0, 2, 3, 4, 5, 6, 8\}$
 - $L_z = \{0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16\}$ } all the possible sum of pairs from L_x and L_y

Multiplication Example

$z_{i,j}$	$\hat{z}_{i,j}$	$L_{i,j}^x$	$L_{i,j}^y$	w_k
$\frac{1}{16} 1/4$	0	0	$L_i^x + L_j^y$	0 1/4
$\frac{1}{8} 1/4$	2	2	$L_i^x + L_j^y$	2 1/4
$\frac{1}{8} 1/4$	2	2	$L_i^x + L_j^y$	2 1/4
$\frac{3}{16} 1/4$	3	3	$L_i^x + L_j^y$	3 1/4
$\frac{3}{16} 1/4$	3	3	$L_i^x + L_j^y$	3 1/4
$\frac{1}{4} 1/4$	4	4	$L_i^x + L_j^y$	4 1/4
$\frac{1}{4} 1/4$	4	4	$L_i^x + L_j^y$	4 1/4
$\frac{3}{8} 1/4$	5	5	$L_i^x + L_j^y$	5 1/4
$\frac{3}{8} 1/4$	5	5	$L_i^x + L_j^y$	5 1/4
$\frac{3}{8} 1/4$	5	5	$L_i^x + L_j^y$	5 1/4
$\frac{3}{8} 1/4$	5	5	$L_i^x + L_j^y$	5 1/4
$\frac{1}{2} 1/2$	6	6	$L_i^x + L_j^y$	6 1/2
$\frac{1}{2} 1/2$	6	6	$L_i^x + L_j^y$	6 1/2
$\frac{1}{2} 1/2$	6	6	$L_i^x + L_j^y$	6 1/2
$\frac{1}{2} 1/2$	6	6	$L_i^x + L_j^y$	6 1/2
$\frac{9}{16} 1/2$	6	6	$L_i^x + L_j^y$	6 1/2

- We have also the correspondence table from L^z to z built using the previous sets
- A group in the table corresponds to a single mapping entry (in bold)

Multiplication Example – at work!

$x_i L_i^x y_j L_j^y$	$(= L_i^x + L_j^y) (= x_i \times y_j)$	$(= \text{cast}(x_i \times y_j))$
$\frac{1}{2} 2 \frac{1}{4} 0$	2	$\frac{1}{4}$
$\frac{1}{2} 2 \frac{1}{2} 2$	4	$\frac{1}{4}$
$\frac{1}{2} 2 \frac{3}{4} 3$	5	$\frac{3}{4}$
$\frac{1}{2} 2 \frac{3}{2} 5$	7	$\frac{3}{4}$
$\frac{1}{2} 2 2 6$	8	1
$\frac{1}{2} 2 4 8$	10	2

$z_{i,j}$	$\hat{z}_{i,j}$	$L_{i,j}^x$	$L_{i,j}^y$	w_k
$\frac{1}{16} 1/4$	0	0	$L_i^x + L_j^y$	0 1/4
$\frac{1}{8} 1/4$	2	2	$L_i^x + L_j^y$	2 1/4
$\frac{1}{8} 1/4$	2	2	$L_i^x + L_j^y$	2 1/4
$\frac{3}{16} 1/4$	3	3	$L_i^x + L_j^y$	3 1/4
$\frac{3}{16} 1/4$	3	3	$L_i^x + L_j^y$	3 1/4
$\frac{1}{4} 1/4$	4	4	$L_i^x + L_j^y$	4 1/4
$\frac{1}{4} 1/4$	4	4	$L_i^x + L_j^y$	4 1/4
$\frac{3}{8} 1/4$	5	5	$L_i^x + L_j^y$	5 1/4
$\frac{3}{8} 1/4$	5	5	$L_i^x + L_j^y$	5 1/4
$\frac{3}{8} 1/4$	5	5	$L_i^x + L_j^y$	5 1/4
$\frac{3}{8} 1/4$	5	5	$L_i^x + L_j^y$	5 1/4
$\frac{1}{2} 1/2$	6	6	$L_i^x + L_j^y$	6 1/2
$\frac{1}{2} 1/2$	6	6	$L_i^x + L_j^y$	6 1/2
$\frac{1}{2} 1/2$	6	6	$L_i^x + L_j^y$	6 1/2
$\frac{1}{2} 1/2$	6	6	$L_i^x + L_j^y$	6 1/2
$\frac{9}{16} 1/2$	6	6	$L_i^x + L_j^y$	6 1/2

Quality metrics

Total gate count AND-OR for each operation for Posit(4, 0).				
Total gates for L^x	Total gates for L^y	Total gates	Grand total gates	Grand total gates of the naïve solution
10	10	11	31	138
7	7	9	23	138
8	5	5	18	138
7	7	9	23	138

Evaluation of results

- We compare our solution to a typical 2D look-up table
- This table is indexed by the 4 bits of the Posit4,0 encoding integer, therefore it has $2^{2 \times 4} = 256$ entries
- Each entry contains the result, therefore it holds 4 bits.
- In total the 2D LUT occupies 1024 bits at most

Conclusions

- We presented a method to perform two-input arithmetic without decoding the operands
- We proposed a general integer programming model that solves the problem of producing mapping for operands and result
- We applied the method to a Posit4,0 format
- We compared a logic synthesis of the obtained mapping against a 2D Look-Up table, being able to reduce logic gates up to 7 times