Alma Mater Studiorum - Università di Bologna

Dipartimento di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria Elettronica

Tesi di laurea
in
Hardware-Software Design Of Embedded Systems

Progetto di un acceleratore hardware per layer di convoluzioni depthwise in applicazioni di Deep Neural Network

Presentata da:
Pietro Maltoni

Relatore:
Chiar.mo Prof.
Francesco Conti

Appello II
Anno Accademico 2020-2021

Abstract

Nel mondo di oggi con il progressivo sviluppo tecnologico, l'industria 4.0, la guida autonoma, le smart cities, i nuovi device indossabili e le applicazioni in ambito biomedico è necessario un costante monitoraggio, controllo e analisi della realtà intorno a noi. Con lo sviluppo di dispositivi IoT sempre più performanti si inizia a parlare di Edge Computing, cioè di calcolo al "confine". In questi dispositivi sono presenti le risorse per elaborare i dati provenienti dai sensori direttamente in locale, senza il bisogno di un trasferimento di dati tra il dispositivo e il server. Questa tecnologia si adatta bene ai nuovi algoritmi di Deep Learning, un insieme di tecniche basate sulle reti neurali artificiali, e, in modo particolare, alle Convolutional Neural Network (CNN), molto efficaci per l'analisi e il riconoscimento di immagini. La nuova frontiera nell'ambito delle CNN è rappresentata dalle Separable Convolution perchè permettono di diminuire in modo massiccio la quantità di operazioni da eseguire su tensori di dati dividendo la convoluzione in due parti: una Depthwise e una Pointwise, rendendo il calcolo sempre più ottimizzato. Tutto questo porta a risultati molto affidabili in termini di accuratezza e velocità nel riconoscimento di immagini, comparabili con quelli dell'essere umano. Il grande problema di questo tipo di applicazioni, però, è rappresentato dal consumo di potenza in quanto i dispositivi si affidano solamente ad una batteria intera. Per questo è necessario avere un buon trade-off tra consumi e capacità computazionale. Per rispondere a questa sfida tecnologica lo stato dell'arte della ricerca in questo ambito propone architetture diverse, composte da cluster con core ottimizzati e istruzioni dedicate o FPGA. In questa tesi proponiamo un acceleratore hardware sviluppato in PULP, una piattaforma open hardware e open software per il calcolo ultra-low power, nata in collaborazione tra l'università di Bologna e l'ETH di Zurigo, orientato al calcolo di layer di convoluzioni Depthwise. Grazie ad una logica HWC dei dati in memoria e al Window Buffer, una finestra che trasla sull'immagine per effettuare le convoluzioni canale per canale, in grado di elaborare tensori di grandezza 3x3 pixels a 8 bit profondi 16 canali, è stato possibile sviluppare una architettura del datapath orientata al riuso dei dati; questo porta l'acceleratore ad avere come risultato in uscita uno throughput massimo di 4 pixel per ciclo di clock. Con le performance di 6 GOP/s, un' efficienza energetica di 101 GOP/j e un consumo di potenza nell'ordine dei mW, dati ottenuti attraverso l'integrazione dell'IP all'interno del cluster di Darkside, nuovo chip di ricerca con tecnologia TSCM a 65 nm sviluppato su PULP, l'acceleratore Depthwise si candida ad essere una soluzione ideale per questo tipo di applicazioni.

Ringraziamenti

Ringrazio in primo luogo il prof. Francesco Conti, grazie per avermi dato l'opportunità di lavorare a questa tesi e per il tuo grande supporto tecnico senza il quale non sarei riuscito a districarmi in un ambiente totalmente nuovo per me. Grazie anche a tutte le persone in Darkside, ho imparato davvero moltissimo in questi mesi di sviluppo del chip. Un ringraziamento speciale a Yvan e Riccardo, tesisti come me e sviluppatori della TPU e del Datamover. Il gruppo di "Attaccamo gli IP" è stato un importante punto di incontro e valvola di sfogo nelle ore di lavoro chiusi in casa per la pandemia.

Un ringraziamento speciale va alla mia famiglia.

A mamma e babbo che mi avete sempre sostenuto durante questi anni universitari e mi avete dato la possibilità di essere la persona che sono oggi.

Allo zio Ale, sei stato un punto saldo a Bologna, una persona su cui ho sempre potuto fare affidamento.

Alla nonna, che mi vizi e mi tratti sempre come il tuo nipotino.

A mia sorella, che, nonostante le tue braccine esili e i pugni che non sai tirare, sei forte come nessuno.

A nonna Nedda e nonno Lino che mi guardano da Lassù.

Non posso non ringraziare anche gli amici che mi hanno accompagnato in questo viaggio: ai Viziatini, Silvia, Serena, Andriy, Golfa, Idol, Riccardo, con cui ricordo con piacere le interminabili giornate al lab 7 accompagnate da Sean Paul e balotte serali. I compagni della triennale, Luca, Filippo, Guerino, Momo e Lorenzo, un gruppo di studio che ci ha portato a superare anche gli esami più difficili. Ai magnifici coinquilini di Galleria 2 Agosto, Luce, Bando, Fraz, alle cene in compagnia, alle serate pizza più Talisman e a Giova sempre in mezzo.

Grazie a Scama, con te ho condiviso tutto e sei sempre stato un punto di riferimento su cui posso contare.

Grazie agli amici di AmaroMonteAgo, per i bei momenti passati insieme, per le serate che nascono da un semplice messaggio: "Stasera?".

Grazie agli amici degli scout, Chiara, Fra, Macel, Scama e Simo, abbiamo condiviso esperienze indimenticabili in questi anni. Grazie tutte le persone in Co.Ca., siamo come una famiglia e so che su di voi posso sempre contare nei momenti di difficoltà.

Indice

Capitolo 1

Introduzione

1.1 Deep neural network

Di questi tempi si sente spesso parlare di intelligenza artificiale (AI) quando si parla per esempio di guida autonoma, riconoscimento di oggetti e altre attività che prima potevano essere ricondotte solo ad azioni e volontà umane. Con l'AI i computer sono in grado di prendere decisioni in autonomia attraverso quello che viene chiamato Machine Learning e, in modo particolare il Deep Learning. Il Deep Learning fa parte della famiglia dei metodi del Machine Learning: algoritmi di raccolta e rappresentazione dati che hanno come caratteristica quella di non essere supervisionati dall'uomo. Questo perchè l'apprendimento avviene attraverso uno o più strati intermedi chiamati "layer". In questi layer sono presenti reti neurali artificiali, come le Deep Neural Network, cioè modelli matematici costituiti da neuroni artificiali che simulano il ragionamento del cervello umano. Il passaggio di informazioni tra i layer consente di simulare la capacità di astrazione delle reti neurali biologiche, un processo con il quale le macchine possono elaborare formule e individuare simboli che permettano loro di risolvere problematiche complesse. L'importanza di questo approccio è ben visibile in applicazioni di Computer Vision, dove il Deep Learning opera attraverso l'acquisizione, l'analisi e la sintesi delle immagini fino a permettere l'identificazione di un determinato oggetto, anche in presenza di tutte le possibili varianti che esso potrebbe assumere nella realtà. Nello specifico ogni strato ha i suoi input che sono elaborati attraverso moltiplicazioni con i "pesi" che permettono al sistema di determinare certe caratteristiche relative all'immagine analizzata. Le CNN (Convolutional Neural Network), introdotte da Yann Lecun nel 1998, sono tra le più importanti reti di DNN pensate specificatamente per il riconoscimento di immagini. Dal 2012, quando AlexNet vinse la ImageNet Large Scale Visual Recognition Competition, hanno sempre di più attirato la curiosità di ricercatori grazie anche alla quantità di dati e immagini che sono state sottoposte a training. Un esempio è quello di ImageNet [1] che contiene milioni di immagini e oltre diecimila classi.

1.2 Convolutional Neural Network

Come suggerisce il nome di Convolutional Neural Network, l'algoritmo di base che queste reti impiegano è la convoluzione, una operazione lineare che opera su due funzioni e ne restituisce una terza. Per cui le reti neurali convoluzionali sono semplicemente reti neurali che utilizzano la convoluzione al posto della moltiplicazione matriciale in almeno uno dei loro strati. In matematica la convoluzione è definita come l'integrale del prodotto tra la prima e la seconda funzione traslata di un certo valore. L'integrale viene poi valutato per tutti i valori di traslazione producendo il risultato della convoluzione.

Nel dominio del tempo:

$$y = (f * g)(t) \int f(\tau)g(t-\tau)d\tau, \tag{1.1}$$

Di certo nel mondo digitale come quello del machine learning non può essere usata una funzione con variabili continue ma piuttosto viene usata la sua corrispettiva discreta. Questa ha come input un array multidimensionale di parametri e come seconda funzione un Kernel delle stesse dimensioni dell'input. Per esempio con una immagine I di 2 dimensioni probabilmente sarà utilizzato un fitro anch'esso di 2 dimensioni.

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{1.2}$$

Questa operazione viene usata quando si parla di filtri digitali, questi sono particolarmente utilizzati in due casi principali:

- Elaborazione di segnali digitali.
- Elaborazione di immagini.

Nell'elaborazione di segnali digitali la convoluzione avviene per applicare filtri che eliminano il rumore nel dominio della frequenza. Può essere fatto un discorso simile per l'elaborazione delle immagini dove vengono applicati kernel per individuare delle particolari caratteristiche dell'immagine come i contorni degli oggetti o determinate figure geometriche ed è questo tipo di elaborazione che è il cardine delle CNN in quanto ogni layer rappresenta una proprietà dell'immagine.

1.2.1 Caratteristiche principali

Per capire meglio le CNN e perchè sono diventate così ampliamente utilizzate nelle reti neurali analizziamo le caratteristiche principali:

- Sparse connectivity
- Parameter sharing
- Equivariance

Si parla di **sparse connectivity** perchè i kernel sono più piccoli delle immagini, questo implica che non ci sia una relazione diretta tra gli output e gli input ma piuttosto una "sparsa".

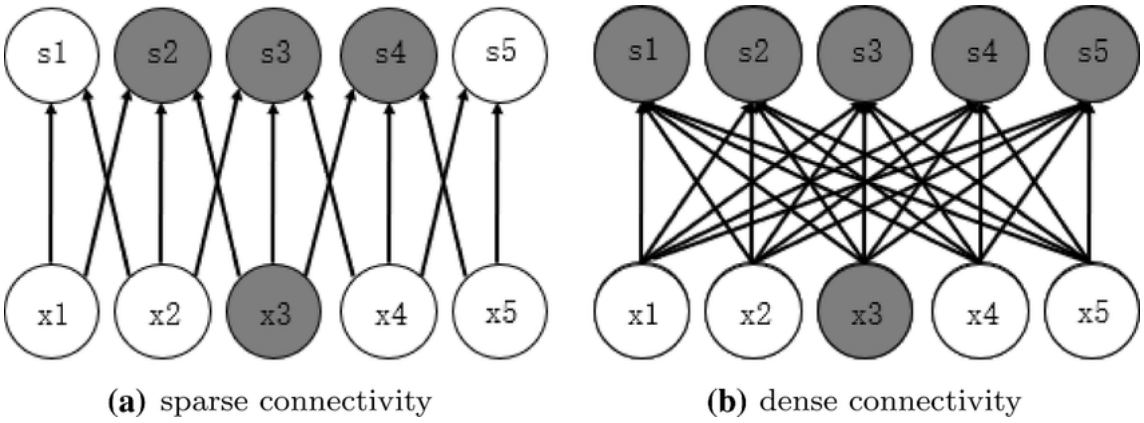


Figura 1.1: Sparse connectivity

Questo permette di avere meno carico sulla memoria perchè i parametri che vengono usati nel kernel sono di meno rispetto ai migliaia di pixel che potremmo avere in ingresso. Di conseguenza ci saranno meno operazioni da compiere quindi un minor dispendio di tempo e consumo di energia. Rispetto alle reti neurali tradizionali dove ogni input viene collegato all'output del layer successivo quella delle CNN rappresenta una migliore ottimizzazione.

Si parla invece di **parameter sharing** perchè, quando si effettua una convoluzione per determinare una determinata caratteristica in ingresso, i pesi del kernel vengono traslati su tutta l'immagine, quindi ogni peso è legato a molteplici pixel di input. Infatti, mentre in genere nelle reti neurali tradizionali lo stesso filtro non viene utilizzato più di una volta, in questo caso i valori del kernel sono legati a tutti gli input.

L'ultimo motivo per cui le CNN sono così efficienti è l'**equivariance**. Questa è introdotta dall'operatore di convoluzione perchè, essendo un sistema lineare nel dominio dello spazio, se ci sono variazione nella immagine di input si ripercuotono nell'output e viceversa.

$$f(g(x)) = g(f(x)) \tag{1.3}$$

1.3 Architettura delle CNN

Come è stato già descritto in precedenza le CNN sono nate e vengono usate per elaborazione di immagini. Questi algoritmi lavorano con una architettura molto specifica, composta da due blocchi principali. Il primo blocco è quello dedicato all'individuazione di determinate caratteristiche dall'immagine selezionata attraverso la convoluzione del kernel con i pixel dell'immagine stessa. Questo primo strato filtra l'immagine e restituisce tante "features maps" (mappe delle caratteristiche), una per ogni kernel applicato nell'immagine.

Nella figura 1.2 si può vedere come l' immagine a colori in input, composta da 224 pixel in altezza e larghezza e 3 canali RGB per il colore, venga elaborata creando dei tensori da 224x224x64. I canali sono aumentati perchè alla stessa immagine solitamente vengono applicati più kernel, ognuno dei quali ha il compito di individuare una determinata caratteristica dell'immagine sotto osservazione. I valori di output vengono poi normalizzati con una funzione di attivazione (ReLU) e/o ridimensionate attraverso il pooling. Questo processo può essere ripetuto più volte: filtrando le mappe delle caratteristiche ottenute con nuovi kernel, che ci danno nuove mappe delle caratteristiche da normalizzare e ridimensionare per poi essere filtrate di nuovo, e così via. Infine, i valori delle ultime "feature map" sono concatenati in un vettore. Questo vettore definisce l'uscita del primo blocco e l'ingresso del secondo.

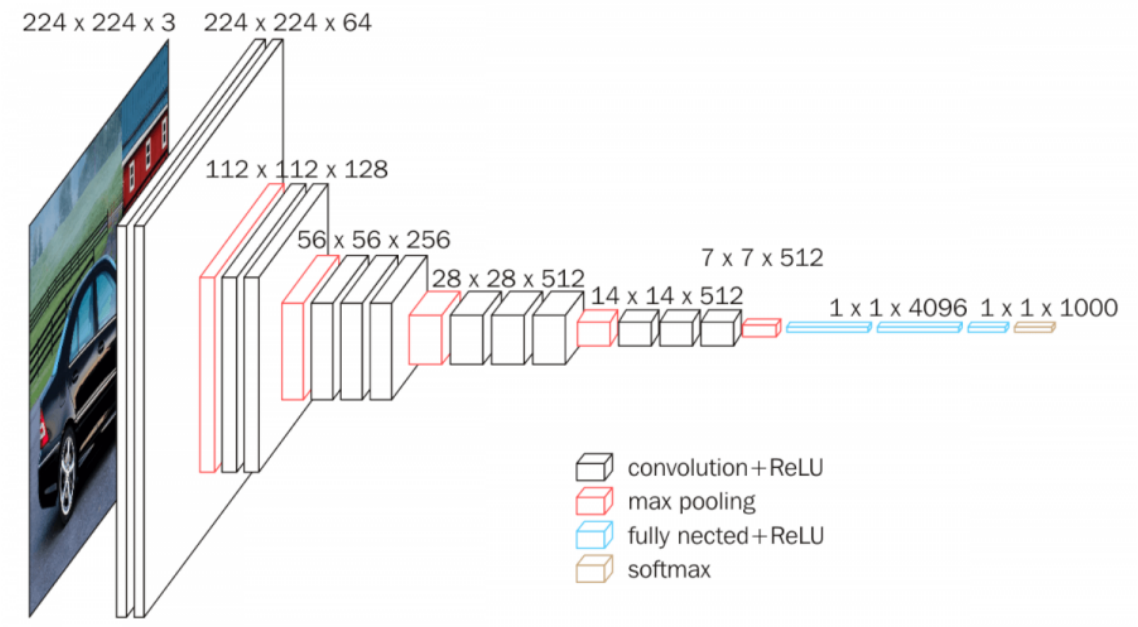


Figura 1.2: Esempio di una CNN

Il secondo blocco non è caratteristico solo alle CNN, ma a tutte le reti neurali. I valori del vettore di input vengono trasformati (con diverse combinazioni lineari e funzioni di attivazione) per restituire un nuovo vettore all'output. Quest'ultimo vettore contiene tanti elementi quante sono le classi: l'elemento i rappresenta la probabilità che l'immagine appartenga alla classe i. Ogni elemento è quindi compreso tra 0 e 1, e la somma di tutti vale 1. Queste probabilità sono calcolate dall'ultimo layer di questo blocco (e quindi della rete), che utilizza una funzione logistica

(classificazione binaria) o una funzione softmax (classificazione multiclasse) come funzione di attivazione. Come per le normali reti neurali, i parametri degli strati sono determinati dalla retropropagazione del gradiente: l'entropia incrociata viene minimizzata durante la fase di addestramento. Ma nel caso della CNN, questi parametri si riferiscono in particolare alle caratteristiche dell'immagine.

Per capire in modo dettagliato l'operazione di convoluzione nelle reti neurali è necessario analizzare passo per passo le operazioni da compiere:

1. Come primo cosa vengono raccolti i valori dei pixel di input e viene applicato ad essi uno specifico kernel partendo dall'angolo in alto a sinistra.

2. Viene effettuata la moltiplicazione elemento per elemento tra i pesi del kernel e i valori di input per essere poi sommati tra loro, questo tipo di operazione viene definita MAC (Multiply ACcumulation)

3. Il kernel viene spostato di una posizione e l'algoritmo si ripete.

La rappresentazione matematica della convoluzione in 3D può essere espressa nel seguente modo:

$$y[k_{out}][w_{out}][h_{out}] = \sum_{k_{in}=0}^{n_c^{l-1}} \sum_{i=0}^{f^l} \sum_{j=0}^{f^l} x[k_{in}][w_{out}+i][h_{out}+j] \cdot W[k_{in}][k_{out}][i][j] \quad (1.4)$$

Con k, w e h sono rappresentate rispettivamente il numero di canali, la larghezza e l'altezza dell'immagine. Per calcolare il pixel di output y viene effettuata la somma fra la moltiplicazione tra il peso in posizione i e j e il pixel corrispondente. Indicando con n_c^{l-1} il numero di canali del layer di riferimento in input e con n_w^{l-1} e $n_h^{l-1}l$ la grandezza e l'altezza dell'immagine da calcolare posso considerare la grandezza del tensore di output come $n_w^l \cdot n_h^l \cdot n_c^l$ dove

$$n_w^l = n_w^{l-1} - f^l + 1 \quad (1.5)$$

$$n_h^l = n_h^{l-1} - f^l + 1 \quad (1.6)$$

$$n_c^l = n_f^{l-1} \quad (1.7)$$

Da queste formule possiamo notare che ogni volta che applichiamo un filtro riduciamo la grandezza dell'immagine perchè l'operazione di convoluzione riduce di un pixel per lato l'immagine di output, per ogni filtro che applichiamo. Dalla equazione 1.7 è evidente che il numero di canali in uscita è uguale al numero di filtri applicati alla immagine perchè ad ogni layer equivale un filtro ed ogni filtro viene utilizzato per individuare una caratteristica dell'immagine.

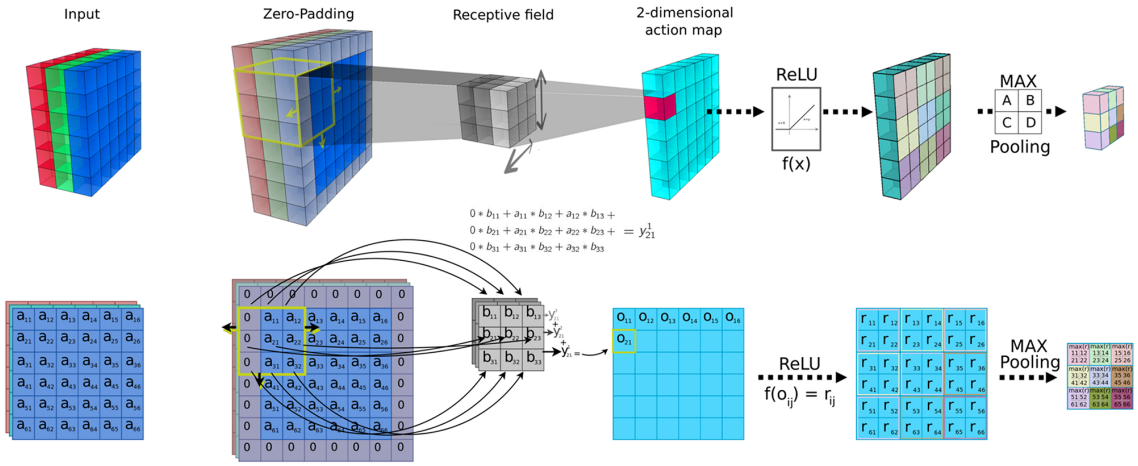


Figura 1.3: Esempio di una convoluzione 3D con Padding e ReLU

1.3.1 Stride e Padding

Le formule enunciate sopra possono variare se consideriamo lo stride e il padding. Il padding viene applicato quando si vogliono preservare le informazioni sul bordo dell'immagine, un modo per fare questa operazione è quello di aggiungere una cornice composta da zeri intorno ai lati come possiamo vedere nella figura 1.3. In questo caso facendo riferimento alle formule precedenti:

$$n_w^l = n_w^{[l-1]} + 2p - f^l + 1 \tag{1.8}$$

$$n_h^l = n_h^{[l-1]} + 2p - f^l + 1 \tag{1.9}$$

$$n_c^l = n_f^{l-1} \tag{1.10}$$

In queste formule p può assumere diversi valori. Quando non c'è la necessità di padding $p = 0$ mentre se vogliamo preservare le dimensioni in output sarà:

$$p = \frac{f^l - 1}{2}$$

La seconda tecnica è quella dello stride. Lo stride controlla come avviene la convoluzione del filtro sull'immagine, se il valore di stride è 1 allora il filtro viene traslato di un pixel, con valori più grandi verrà traslato di s pixels.

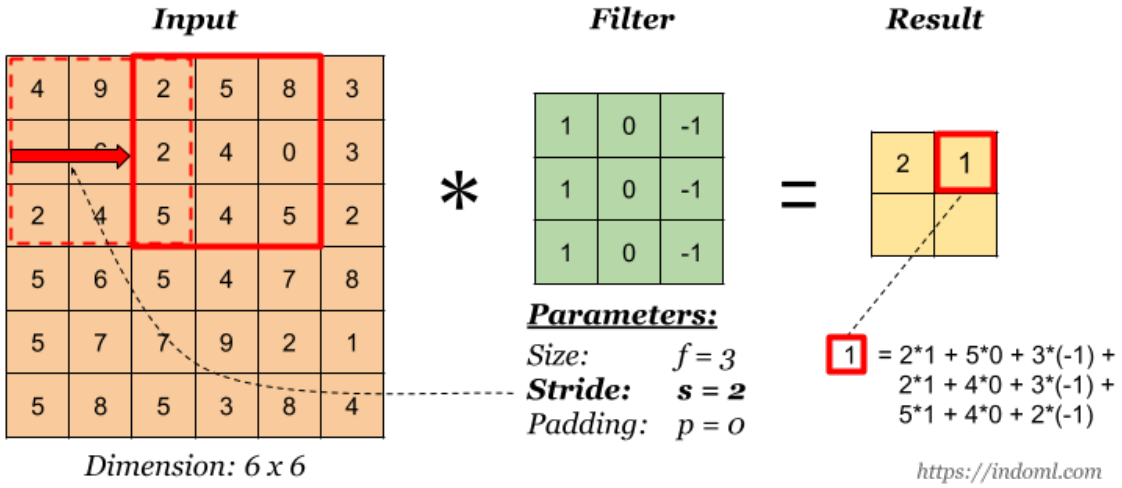


Figura 1.4: Stride di 2 pixels

La formula precedente con l'aggiunta di striding diventa:

$$n_w^l = \lfloor \frac{n_w^{[l-1]} + 2p - f^l}{s} + 1 \rfloor \tag{1.11}$$

$$n_h^l = \lfloor \frac{n_h^{[l-1]} + 2p - f^l}{s} + 1 \rfloor \tag{1.12}$$

1.3.2 Pooling, funzioni di attivazione e ReLU

Come è stato accennato in precedenza tra i vari layer delle reti neurali sono presenti operazioni di pooling e ReLU ma non si è discusso a pieno il loro significato. Torniamoci a bomba.

Semplicemente il pooling è un metodo per ridurre la dimensione dell'immagine suddividendola in blocchi e mantenendo solo quello dal valore più alto, così facendo si riduce il problema di overfitting, cioè di avere più dati di quelli necessari per la classificazione, e si mantengono le aree con maggiore attivazione.

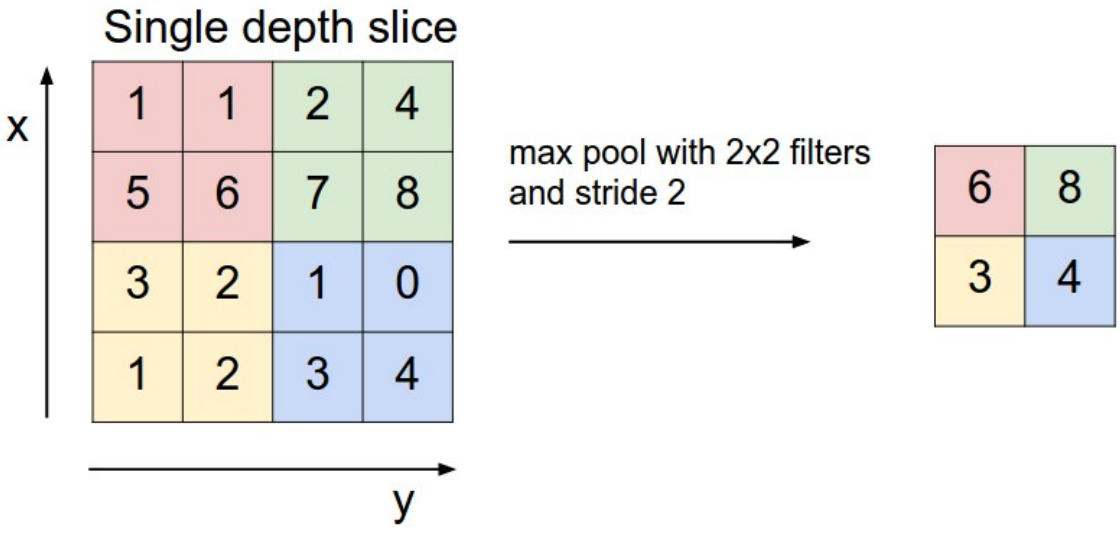


Figura 1.5: Esempio di pooling con stride=2

Quando invece parliamo di ReLU si intende un tipo particolare di funzione di attivazione.

Una funzione di attivazione è una funzione decisionale che determina la presenza di una particolare caratteristica neurale. È mappata tra 0 e 1, dove con zero è indicata l'assenza della funzione, mentre con uno la sua presenza. Una rete neurale deve essere in grado di prendere qualsiasi input, da - infinito a + infinito, e mapparlo su un output che varia in relazione alla necessità. La non linearità è necessaria nelle funzioni di attivazione perché il suo scopo in una rete neurale è produrre un confine di decisione non lineare attraverso combinazioni non lineari di peso e input.

Tra le varie funzioni di attivazione ci sono le seguenti:

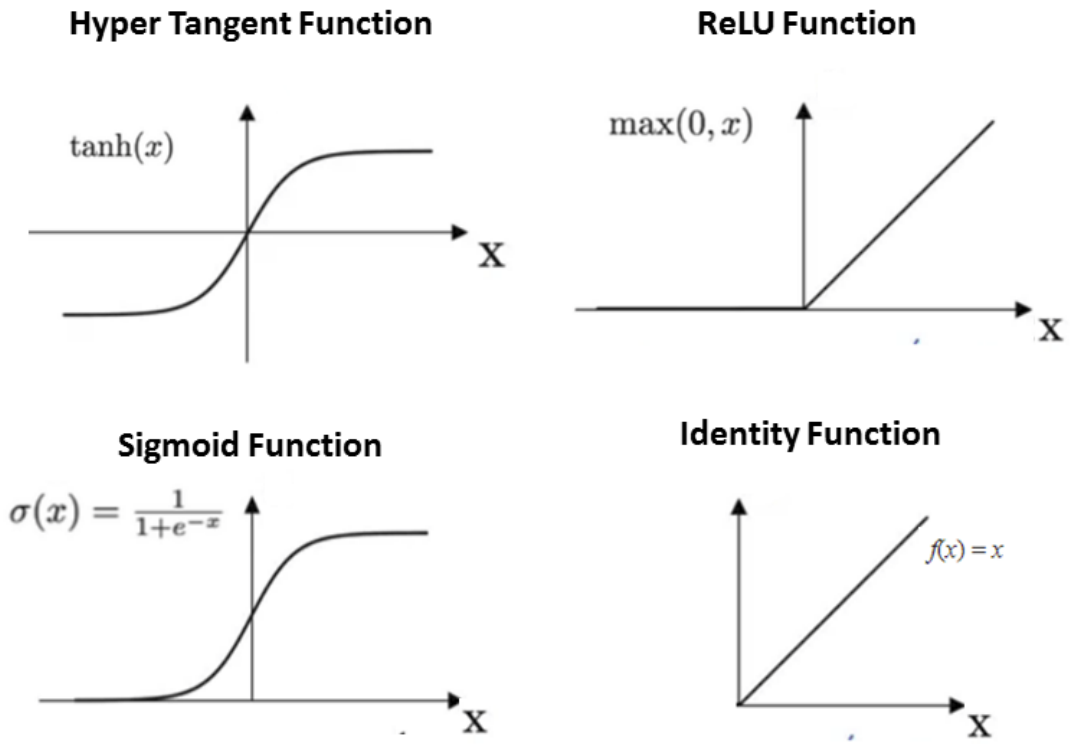


Figura 1.6: Funzioni di attivazione

Quella più usata nelle CNN è la ReLU (Rectified Linear Unit) perchè è veloce, semplice e non soffre dal problema del vanishing gradient. Infatti altre funzioni di attivazione come la Sigmoid Function potrebbero portare i valori di output molto vicini tra loro con derivate molto piccole e questo potrebbe comportare una scomparsa delle caratteristiche che sono fondamentali per la classificazione delle immagini. In più questo rallenterebbe anche la back propagation, l'algoritmo che deriva direttamente con il processo di training i parametri che funzionano meglio per la rete desiderata.

1.4 Soluzioni Hardware

Le moderne attività legate alle Deep Neural Network vengono eseguite su server cloud, personal computer o smartphone. Anche nello scenario più limitato dei dispositivi mobili come i cellulari, la loro esecuzione può contare su vari GB di memoria e notevole potenza di elaborazione disponibile. Negli ultimi tempi con lo sviluppo sempre più massiccio di dispositivi IoT come wearable devices, sensori wireless, occhiali per le realtà aumentata, è diventato necessario eseguire lo stesso tipo di calcoli su dispositivi edge, cioè vicino alla sorgente dei dati. I vincoli a cui questi dispositivi devono sottostare sono nella maggior parte dei casi:

1. Vincoli rigorosi in termini di memoria (pochi MB off-chip e 1 MB on-chip al massimo).
2. Capacità di calcolo limitate.
3. Vincoli in batteria e potenza di picco.

Tra le soluzioni hardware possibili possiamo elencare acceleratori FPGA e System On Chip (SoC). Nella prima soluzione il circuito integrato è programmabile dinamicamente, questo permette di realizzare diverse architetture per implementare le CNN nel modo più ottimizzato a seconda dell'applicazione da sviluppare senza dover sostenere i costi elevati per la fabbricazione di un chip. Mentre i SoC sono circuiti integrati che all'interno contengono un intero sistema. Oltre al processore centrale integra dei controller per la memoria e le periferiche. In questo ambiente possono essere aggiunti acceleratori hardware specifici per un determinato tipo di operazioni, nel nostro caso CNN. Nel prossimo capitolo verrà analizzato lo stato dell'arte di alcuni acceleratori hardware moderni, sia FPGA che SoC, che tentano di risolvere questa sfida tecnologica.

Capitolo 2

Stato dell'arte degli acceleratori hardware per CNN

In questo capitolo analizziamo lo stato attuale della ricerca per quanto riguarda gli acceleratori hardware per CNN facendo riferimenti a esempi su schede FPGA e su system on chip, dei quali sarà particolarmente in evidenza PULP, la piattaforma nella quale è stato implementato l'acceleratore di questa tesi.

Lo studio su architetture hardware specifiche per applicazioni di Deep Learning e Computer Vision fa parte di un'area di ricerca molto attiva. Per migliorare l'efficienza energetica, molte piattaforme fanno affidamento su architetture specifiche, ottimizzate per il flusso di dati per CNN, spesso implementate su FPGA o CGRA. Tra gli esempi di questo approccio possiamo trovare Vortex [2], un System-On-Chip per applicazioni di CV ispirate biologicamente per accelerazione della visione ispirata ai sensori neuromorfici. I sensori neuromorfici tentano di imitare le caratteristiche di rilevamento e di elaborazione visiva degli organismi viventi. L'obiettivo di questi sensori è quello di aiutare a ridurre il carico computazionale richiesto per la percezione visiva estraendo solo le informazioni rilevanti.

NeuFlow [3] e nn-X [4] sono altri due esempi di architetture hardware scalabili a bassa potenza che si concentrano sull'accelerazione a tempo reale di DNN. nn-X è implementato su dispositivi logici programmabili e comprende una serie di elementi di elaborazione configurabili chiamati 'collection'. Questi componenti eseguono le operazioni specifiche delle DNN: convoluzione, quantizzazione e funzioni di attivazione. Il sistema nn-X include 4 interfacce di accesso diretto alla memoria ad alta velocità per la memoria DDR3 e due processori ARM Cortex-A9. Un altro approccio comune è quello di aumentare le istruzioni di un processore RISC con un'estensione simile alle SIMD (single istruction multiple data). Qadeer et al. [5] presenta un convolution engine (CE), una estensione al processore Tensilica, che, non essendo un acceleratore autonomo, è utilizzabile tramide le tradizionali estensioni SIMD. Il CE prevede anche una certa flessibilità: è dotato di ALU complete e localizza la memoria in strutture di archiviazione ottimizzate per il flusso di dati specifico per l'applicazione. Questa regolazione elimina i trasferimenti di dati ridondanti e facilita la creazione di percorsi dati strettamente accoppiati e strutture di archiviazione che consentono di eseguire centinaia di operazioni a basso consumo energetico. Tra le piattaforme commerciali che seguono un simile trend possimao trovare Movidius Myriad [6], che estende i processori RISC utilizzando coprocessori VLIW. Clemons

et al. hanno proposto EVA (Efficient Vision Architecture), una architettura multicore asimmetrica con 1 core esterno dediato e molti core a supposrto a basso consumo di energia, tutti potenziati con istruzioni SIMD specializzate ad eseguire operazioni comuni come il prodotto scalare.

2.1 PULP e HWCE

PULP è una piattaforma open hardware e open software per il calcolo ultra-low power, nata in collaborazione fra il gruppo Energy-efficient Embedded Systems (EEES) dell'università di Bologna e l'Integrated System Laboratory (IIS) dell'ETH di Zurigo. La piattaforma PULP include al suo interno un microcontrollore all'avanguardia e una piattaforma multi-core in grado di raggiungere una alta efficienza energetica anche ad elevate prestazioni. Date le caratteristiche che sono state appena elencate l'ecosistema PULP è il candidato ideale per lo sviluppo di applicazioni di Deep Neural Network.

Un esempio importante sviluppato in PULP è l'hardware convolution engine (HWCE) [7] del gruppo di ricerca del professor Benini. Nel loro lavoro hanno proposto di aumentare il cluster di PULP con l'HWCE, un co-processore a basso consumo energetico per accelerare il calcolo delle convoluzioni in contesti BICV (Biological Inspired Computer Vision) e in altre applicazioni che prevedono un alto carico computazionale di CNN. L'HWCE è stata una grossa fonte di ispirazione per lo sviluppo dell'acceleratore che viene presentato in questa tesi e verrà presentato più nel dettaglio nella sezione 2.4.

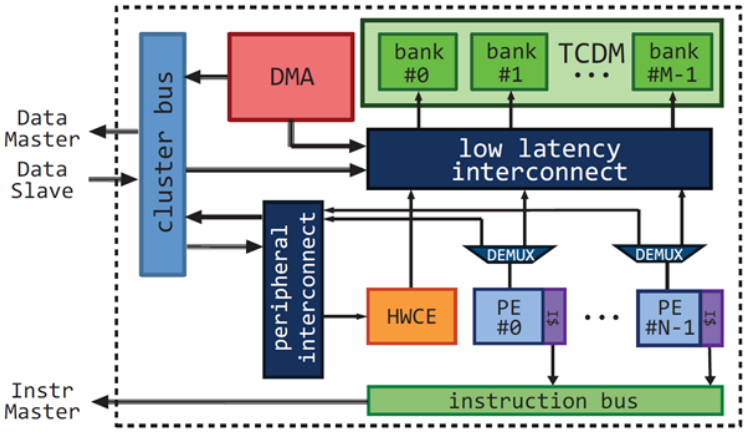


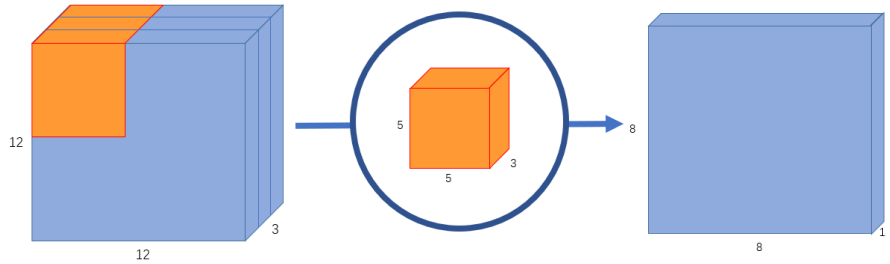
Figura 2.1: HWCE integrato nel PULP CLUSTER

2.2 MobilNet e Convoluzioni Depthwise

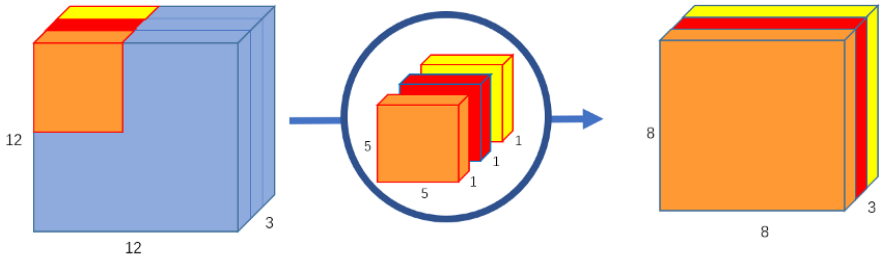
Negli ultimi tempi la ricerca nel campo delle CNN si è evoluta e tra gli ultimi studi effettuati copre un ruolo di particolare importanza per lo sviluppo di questa tesi quello di MobileNet [8]. Nelle reti neurali convoluzionali (CNN), le convoluzioni 2D sono lo strato più utilizzato. MobileNet rappresenta un'architettura CNN molto più veloce e un modello più piccolo che fa uso di un nuovo tipo di strato convoluzionale, noto come Separable Convolution. A causa delle ridotte dimensioni del modello è implementabile su dispositivi mobili e embedded, da qui il nome MobileNet.

Le convoluzioni Depthwise sono così chiamate perché operano non solo nelle dimesioni 2D, ma anche nella dimensione della profondità (Depth), cioè il numero di canali. Un'immagine in ingresso può avere 3 canali: RGB (Red, Green, Blue). Ma dopo alcune convoluzioni un'immagine può trovarsi ad avere anche centinaia di canali. Ogni canale può essere immaginato come una particolare interpretazione di quell'immagine; ad esempio, il canale rosso interpreta il "rosso" di ciascun pixel, il canale blu interpreta il "blu" e il canale verde interpreta il "verde". Quindi un'immagine con 64 canali ha 64 diverse interpretazioni di quell'immagine.

Per capire meglio come viene velocizzato il calcolo con le Depthwise convolution prendiamo un esempio: se da un immagine 12x12x3 applico un kernel 5x5x3 senza padding e con stride 1 otterei una immagine 8x8x1.



(a) Convoluzione Classica



(b) Depthwise convolution

Nella Depthwise convolution invece ogni canale viene mantenuto separato. Quindi viene effettuata una convoluzione 2D tra l'immagine e il kernel e come risultato ottengo una immagine 8x8x3. Dopo questa operazione per arrivare allo stesso risultato del metodo classico viene applicato un altro layer attraverso un kernel 1x1x3. Questa operazione prende il nome di Pointwise Convolution.

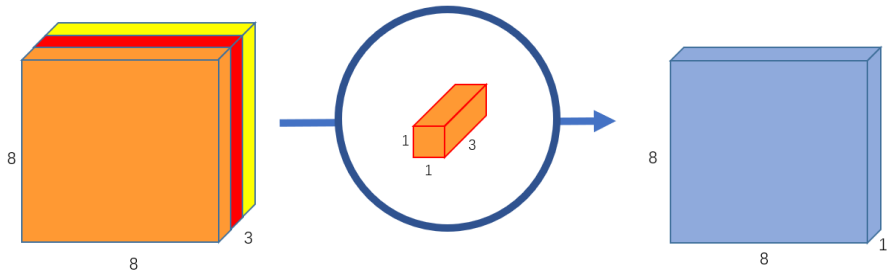


Figura 2.3: Pointwise convolution

Riprendendo l'espressione 1.4 è possibile calcolare il costo computazione della convoluzione classica in termini di moltiplicazioni effettuate: $n_w^{l-1} \cdot n_h^{l-1} \cdot n_c^l \cdot n_c^{l-1} \cdot n_w^l \cdot n_h^l$

La Depthwise separable convolution è composta da due layer, uno per la Depthwise e uno per la Pointwise. La formula per calcolare la Depthwise convolution può essere scritta nel seguente modo:

$$y[k_{out}][w_{out}][h_{out}] = \sum_{i=0}^{f^l} \sum_{j=0}^{f^l} x[k_{in}][w_{out}+i][h_{out}+j] \cdot W[k_{out}][i][j] \tag{2.1}$$

Il costo computazionale diventa: $n_w^{l-1} \cdot n_h^{l-1} \cdot n_c^{l-1} \cdot n_w^l \cdot n_h^l$. A questo è poi da aggiungere il layer per effettuare la pointwise convolution e si arriva ad ottenere un costo computazionale complessivo di $n_w^{l-1} \cdot n_h^{l-1} \cdot n_c^{l-1} \cdot n_w^l \cdot n_h^l + n_c^{l-1} \cdot n_c^l \cdot n_w^l \cdot n_h^l$.

Rapportandole tra loro si ottiene una riduzione di:

$$\frac{n_w^{l-1} \cdot n_h^{l-1} \cdot n_c^{l-1} \cdot n_w^l \cdot n_h^l + n_c^{l-1} \cdot n_c^l \cdot n_w^l \cdot n_h^l}{n_w^{l-1} \cdot n_h^{l-1} \cdot n_c^l \cdot n_c^{l-1} \cdot n_w^l \cdot n_h^l} = \frac{1}{n_c^l} + \frac{1}{n_w^{l-1} \cdot n_h^{l-1}} \tag{2.2}$$

Tornando all'esempio in figura 2.3 è possibile calcolare anche qui la massima riduzione di capacità di calcolo ottenibile. Se per esempio viene considerata in output una immagine con 256 canali si applicano 256 filtri di dimensione 5x5x3. Quindi si avranno 256x5x5x3 moltiplicazioni per ogni posizione del filtro sull'immagine. Il kernel si muove 8x8 volte quindi il totale di moltiplicazioni necessarie sarà di 256x5x5x3x8x8 = 1228800.

Nel caso della depthwise convolution invece sono presenti 3 kernel 5x5x1 che si muovono nell'immagine 8x8 volte, per un totale quindi 3x5x5x8x8 = 4800 moltiplicazioni. Nella pointwise sono presenti 256 1x1x3 kernel che traslano sempre di 8x8 posizioni quindi il totale sarà di 256x3x8x8 = 49152. Sommando questi due numeri si ottiene che nel caso della separable convolution bastano solamente 4800 + 49152 = 53952 moltiplicazioni. Un risultato incredibile che porta a ridurre le operazioni da compiere di un fattore 22x.

2.3 Esempi di acceleratori per le separable convolution

Visto che lo studio di MobileNet sulle separable convolution è abbastanza recente non sono molti gli esempi di acceleratori specifici per questo tipo di convoluzioni. Tra questi possiamo trovare due architetture sviluppate su FPGA sviluppate dai gruppi di ricerca di Lin Bai [9] e di Wei Ding [10]. Questa architetture sono piuttosto simili, la prima è composta da un weight buffer che dialoga con la memoria direttamente per caricare i pesi del kernel, un engine per le moltiplicazioni matriciali e un buffer per i risultati intermedi delle feature maps. Nella seconda invece è integrato un DMA che dialoga con i buffer degli input e dei pesi, con la memoria e con il processore. Una rete di controllo governata da una FSM (Final State Machine) controlla le operazioni.

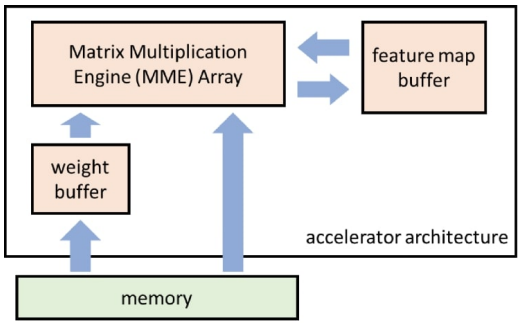


Figura 2.4: Lin Bai architecture

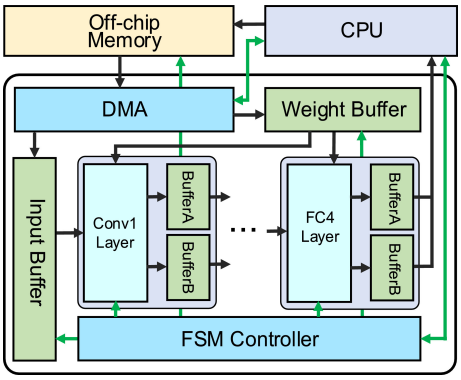


Figura 2.5: Wei Ding architecture

Approfondiamo meglio questi esempi perchè sono utili per capire le scelte architetturali compiute nello sviluppo di questa tesi. Scendendo nel dettaglio il matrix multiplication engine è un array composto da tante unità di calcolo, in ognuna di esse è presente:

1. Un line Buffer: un buffer di linea che prende come input i pixel dell'immagine, li porta al multiplier array e li trasla.
2. Un multiplier array grande 3x3x32 unità di calcolo.
3. Un sommatore ad albero, configurabile per eseguire le somme per le depthwise o le pointwise.
4. Un blocco per la normalizzazione che effettua la batch normalization.
5. Un blocco di ReLU e Pooling.

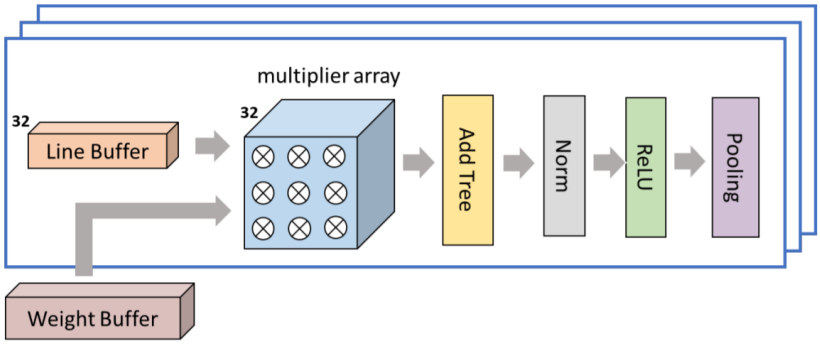


Figura 2.6: MME architecture

Questa metodologia di progettazione è scalabile, il framework può essere implementato più volte per creare un array di MME, in modo da ottimizzare le risorse e le prestazioni del chip. Con questa tecnologia la nuova CNN all'avanguardia di MobileNetV2, è stata implementata su Arria 10 SoC FPGA. I risultati di questo acceleratore mostrano che è in grado di eseguire 266,6 frame al secondo e 170,6 Giga di operazioni al secondo (GOPS) alla frequenza di clock del sistema di 133MHz. Questo rappresenta un'accelerazione di 20 volte superiore rispetto a quella della CPU standard.

Anche nell'acceleratore di Ding ci sono vari strati convoluzionali che sono la parte ad alta intensità di calcolo dell'intero sistema di accelerazione. L'effetto finale dell'accelerazione della CNN dipende principalmente dall'ottimizzazione di questi strati. La Fig. 2.7 è il diagramma schematico del motore di calcolo del layer della separable convolution. Il motore di calcolo include principalmente i banchi per il buffer dei dati di input, i banchi di buffer di peso, le unità di Depthwise convolution (DCU), le unità di pointwise convolution (PCU) e i banchi di buffer di uscita. Con questa architettura sono riusciti ad ottenere i risultati di 98.91 GOP/s con un consumo di 8.5 W.

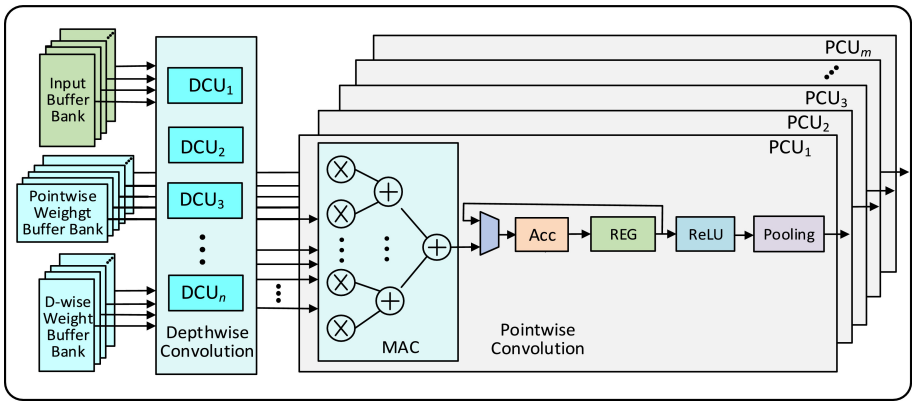


Figura 2.7: Data Path

Prendendo esempio dagli acceleratori appena discussi in questa tesi è stato sviluppato un acceleratore hardware per accelerare le convoluzioni depthwise nella piattaforma PULP, per implementarlo poi come IP nel cluster di Darkside, l'ultimo chip basato su OpenPULP.

2.4 Tecniche per il riutilizzo dei dati

Per ottimizzare il riutilizzo dei dati negli esempi degli acceleratori precedenti sono state usate diverse tecniche di design. Nell'HWCE e nel MME è stato sviluppato un line buffer per il riuso di dati.

Per calcolare una convoluzione con un kernel grande KxK pixels il line buffer usa come strategia quella di caricare in un buffer K-1 righe dell'immagine e K pixels della riga successiva. Nella prima fare il buffer è riempito dai pixel in input fino al riempimento, poi inizia la computazione. Ogni ciclo:

1. un nuovo pixel è inserito nello shift register
2. il pixel più in alto a sinistra viene scartato
3. la finestra di KxK pixel più a sinistra è usata per effettuare la convoluzione.

In questo modo la bandwidth richiesta è solo di un pixel per ciclo in ingresso per raggiungere il picco di throughput di 1 pixel per ciclo in uscita.

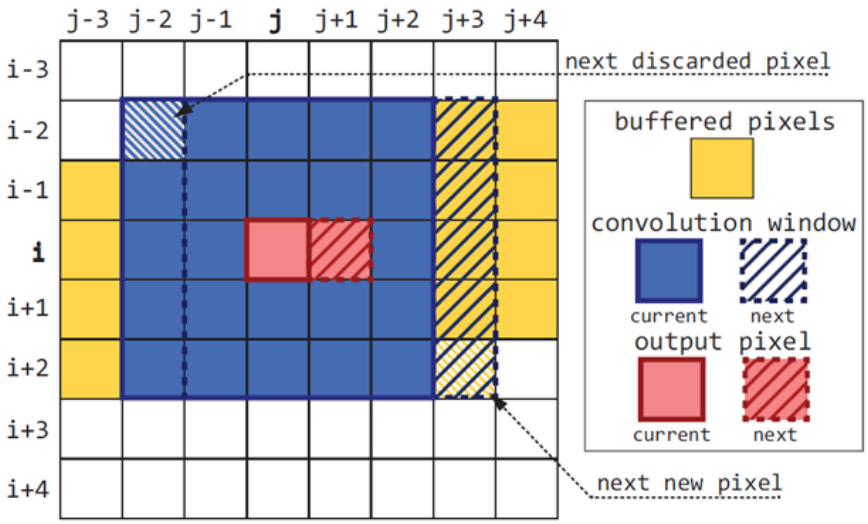


Figura 2.8: Line buffer nell'immagine 2D

Nel design dell'acceleratore di Wei Ding invece, per eseguire letture e scritture dalla memoria all'engine, sono stati messi buffer di input e di output vicino alla rete di computazione. Questa soluzione utilizza 2 banchi di buffer a ping-pong, A e B, sia per i dati in input sia per quelli in output con dimensioni uguali. Un determinato layer durante la computazione in questo modo può scrivere o leggere dal banco buffer A mentre il layer successivo legge dal banco buffer B. Ogni banco ha un quantità di RAM parallele espresso da Tn, valore che è correlato al parallelismo del canale di ingresso di ogni layer. I buffer in output salvano i risultati generati dalla convoluzione nell'engine che possono essere riusati nelle computazioni successive. Il numero delle RAM in ogni Buffer in output è Tm. I trasferimenti di memoria off-chip sono necessari solo per il caricamento dell'immagine di input, il salvataggio in output e il caricamento dei pesi per ogni layer. I buffer da ping-pong in questo

modo possono evitare la sovrapposizione di richieste di trasferimenti in memoria e migliorare le prestazioni.

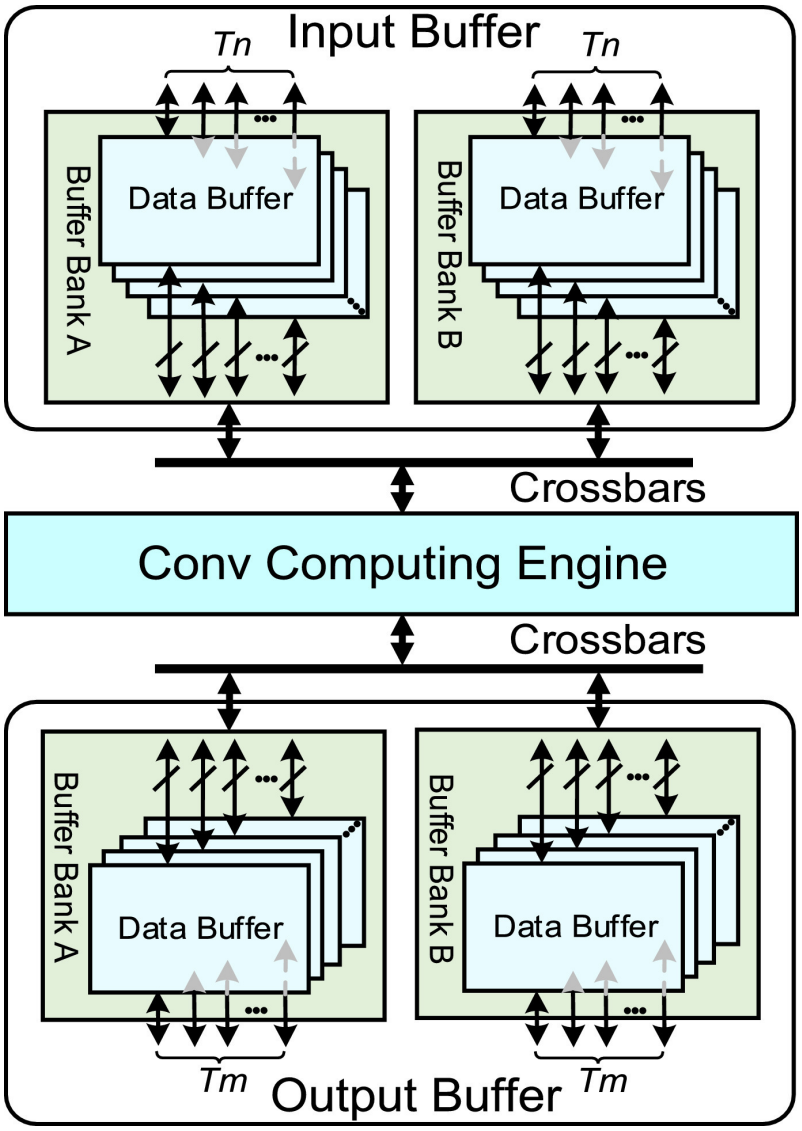


Figura 2.9: Buffer a ping-pong

In questa tesi proponiamo un acceleratore hardware per le convoluzioni depth-wise che utilizza il window buffer, una finestra di scorrimento sull'immagine, per il riuso dei dati. Nel prossimo capitolo verranno illustrate più nel dettaglio le scelte progettuali e i componenti pensati per l'architettura dell'acceleratore.

Capitolo 3

Hardware design dell'acceleratore Depthwise

3.1 Come è sviluppato un HWPE

Come è stato anticipato nel capitolo 2 con l'HWCE, gli HWPE (Hardware Processing Engine) sono acceleratori dedicati, che possono essere inseriti nel SoC o nel cluster di un sistema PULP per amplificare le sue performance e l'efficienza energetica in un particolare task.

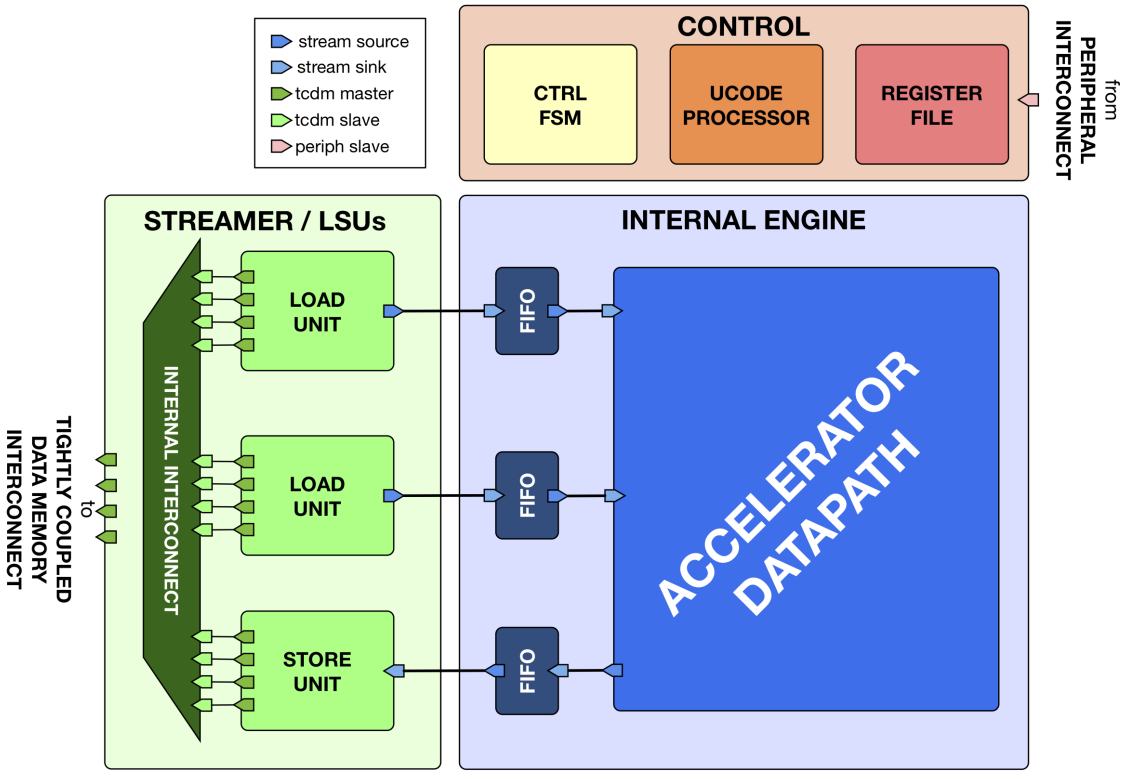


Figura 3.1: HWPE Template

Nella figura 3.1 è presente la struttura di un acceleratore HWPE. Per connettere l'HWPE in un sistema PULP vengono usate una serie di moduli e interfacce divisi in:

1. Streamer: un modulo che integra una interfaccia per collegare l'engine interno con la Tightly Coupled Data Memory (TCDM), una memoria da 128 kB organizzata in 32 banchi SRAM condivisa da tutte le risorse del cluster.
2. Control: un modulo con un'interfaccia predisposta al controllo e alla programmazione dell'acceleratore tramite un register file e una FSM interna.
3. Engine: modulo dove si trova il data-path dell'acceleratore.

Streamer

Lo streamer è il modulo che trasporta i dati dalla memoria al datapath o viceversa. La direzione del trasferimento avviene sempre da una interfaccia di source ad una di sink. L'interfaccia tra i vari moduli interni allo stramer è gestito tramite 4 segnali, data e strobe rappresentano il pacchetto di dati da inviare, valid e ready gestiscono l'handshake. Il trasferimento è soggetto alle seguenti regole:

1. Quando il clock si alza e sia il valid che il ready sono ad 1 avviene il trasferimento
2. Il dato, multiplo di 32 bit, e lo strobe, che indica quali dei bytes in data sono da considerare validi, possono cambiare valore o quando il valid va a zero, oppure nel ciclo che segue un trasferimento avvenuto con successo.
3. Per evitare stalli nel flusso di dati il segnale di valid e quello di ready sono combinati tra loro: il valid non può dipendere in modo combinatorio dal ready, mentre può succedere il contrario. In altre parole il segnale di ready può passare da 0 a 1 se il valid fa altrettanto.
4. Il passaggio del valid da 1 a 0 può capitare soltanto dopo un trasferimento andato a buon fine, questo per evitare consumi indesiderati di dati.

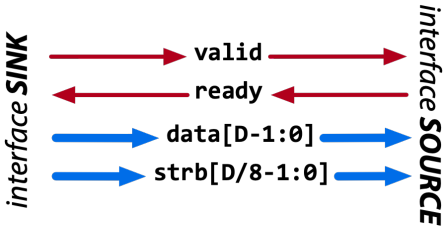


Figura 3.2: Interfaccia HWPE Stream

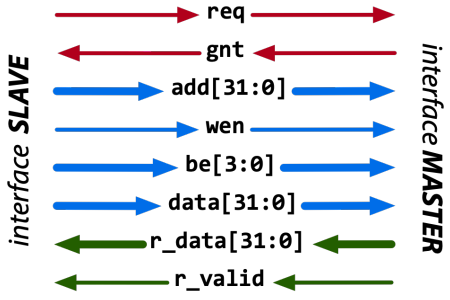


Figura 3.3: Interfaccia HWPE TCDM

Invece nell'interfaccia tra l'HWPE e la TCDM, memoria esterna condivisa, c'è un protocollo simile a quelli usati per le DMA che collega l'interfaccia master alla slave con uno scambio di req e gnt:

1. Nel ciclo di clock dove sia il req sia il gnt sono a 1 avviene l'handshake per le transizioni sia di lettura che di scrittura.

2. Il segnale r_valid deve essere alto dopo che è avvenuta con successo una lettura dalla memoria e la r_data deve essere valida in questo ciclo. Questo avviene perchè le memorie sono fortemente accoppiate, anche se non riescono a rispondere in un ciclo devono garantire la transizione con un delay.

3. Mentre il passaggio da 0 a 1 del req non dipende dallo stato del gnt, il passaggio a 1 del gnt dipende in modo combinatorio dal req, sempre per impedire stalli.

Control

Il modulo di control include al suo interno tre diversi sub-moduli:

1. Una rete FSM che controlla l'acceleratore, specifica per l'acceleratore.

2. Un register file memory-mapped implementato con i latch al posto di flip-flop per salvare area e potenza che include due diversi set di registri:

   - Registri Generici (job-independent): sono registri che salvano parametri che non cambiano durante l'esecuzione
   - Registri job-dipendent: in questi registri sono salvati parametri come il base address per ogni tipo di dati (input, pesi e output), controlli per l'engine come per esempio il numero di righe e di colonne nell'immagine ecc. La peculiarità di questi è che possono cambiare quando viene riprogrammato l'acceleratore per un nuovo lavoro, anche mentre è in funzione.

3. un processore micro-code: usato per aggiornare l'offset per l'accesso dei dati in memoria mentre avviene la computazione.

Engine

L'engine è il cuore dell'acceleratore, il data-path dove avviene l'effettiva computazione e dialoga con l'HWPE-stream per effettuare load e store dalla/alla memoria e viene controllato nella sua esecuzione dalla rete di control.

3.2 Depthwise Engine

L'obiettivo fissato per lo sviluppo del nostro acceleratore è quello di avere un buon riuso di dati per evitare di effettuare troppi accessi in memoria laddove non ce ne fosse il bisogno. Dal capitolo precedente abbiamo visto degli esempi sullo stato dell'arte di diversi acceleratori che hanno risposto in modi diversi a questa richiesta. Un aspetto della convoluzione Depthwise è che ogni canale viene calcolato con un filtro diverso, per cui il calcolo può facilmente essere parallelizzato per poter calcolare nello stesso momento più convoluzioni diverse, ognuna relativa ad un canale dell'immagine con il relativo filtro.

Il modo più conveniente per sfruttare questo tipo di parallelismo è che i dati in memoria siano salvati in formato HWC (Height , Width, Channel).

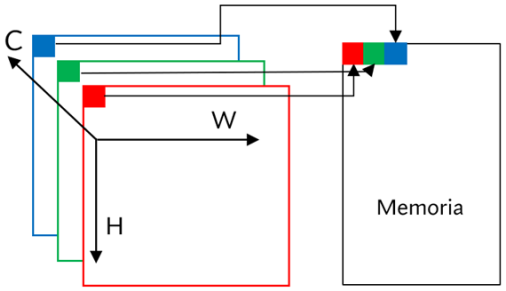


Figura 3.4: CHW

Con questa sigla si intende che ad indirizzi continui in memoria saranno salvati i pixel dei canali del tensore, poi quelli continui nella stessa nella riga e infine nell'altezza. Se per esempio prendiamo in considerazione un immagine con tre canali RGB i bit che rappresentano R, G, B relativi ai tre colori del primo pixel dell'immagine saranno continui in memoria.

L'architettura del Depthwise accelerator ha come idea di base di sfruttare questa disposizione in memoria. Per dialogare con la memoria sono state usate 4 porte tcdm, ognuna da 128 bit. Supponendo che un pixel dell'immagine sia grande 8 bit, in questo modo è possibile fare una load da 16 pixel in un ciclo di clock, corrispondenti ai primi 16 canali del layer depthwise. Per fare questo l'engine è composto da:

- Un window buffer per il salvataggio dei pixel dell'immagine in input.
- Un weight buffer per il salvataggio dei pesi del kernel.
- Una rete di multiply accumulation (MAC).
- Un blocco per la ReLU.
- Un blocco per shiftare il risultato.
- Un blocco per effettuare la clip8.

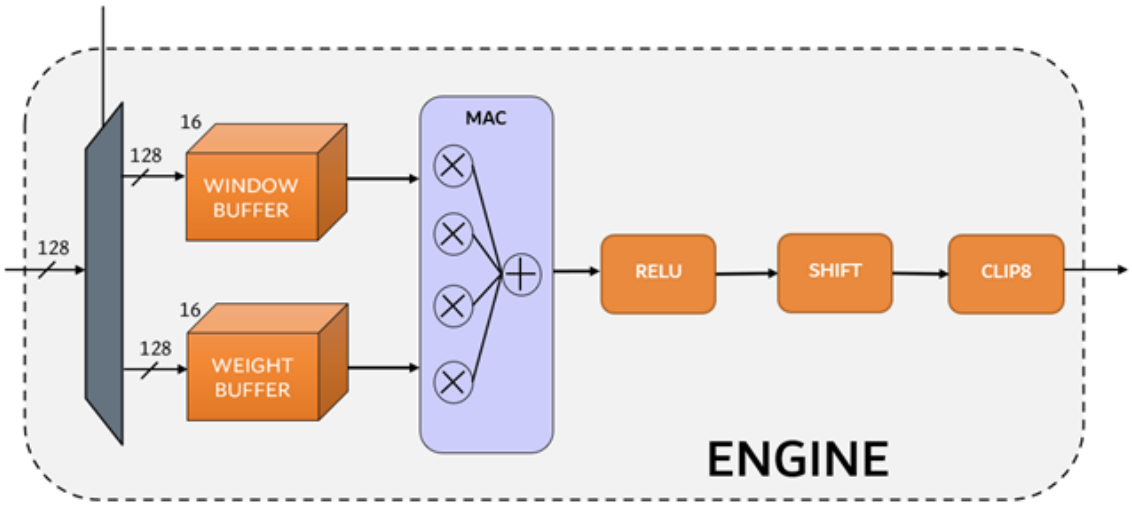


Figura 3.5: Engine

L'acceleratore è stato pensato per effettuare le convoluzioni con kernel 3x3, con stride uguale a 1 e senza padding.

3.2.1 Buffer per pesi e attivatori

Il Windows Buffer svolge il compito di immagazzinare i dati provenienti dall'input in un tensore di dimensione 3x3x16 (width, height, channel) pixel. Ogni pixel è di 8 bit ed è aumentato di un bit di segno (0) per essere poi moltiplicato con i pesi del kernel che possono anche avere valori negativi. Questo porta il tensore per gli attivatori ad avere le dimensioni di 3x3x16 pixel ognuno da 9 bits, per un totale di 1296 bits.

```
//activators are extended with signed bit
        logic [NB_CHANNELS-1:0][BW:0] activators[2:0][2:0];
```

Listing 1: Array multidimensionale per gli attivatori in *depthwise_ engine.sv*

Per effettuare la convoluzione sull'immagine intera il window buffer "trasla" sull'immagine. Per fare questo al suo interno ha una mini buffer per il caricamento dei 3 pixel della prossima riga: ogni ciclo di clock carica i 128 bits relativi ai primi 16 canali di un pixel che vengono salvati nel next_row. Dopo 3 cicli di clock la nuova riga è piena e al quarto ciclo il window buffer "shifta" di una riga verso il basso. Questo ciclo si ripete finchè non si arriva a fine della colonna, dopodichè attraverso la rete di control viene riprogrammato con il puntatore dei nuovi dati impostato sul secondo pixel della prima riga.

```
//internal buffer for loading next row
logic [NB_CHANNELS-1:0][BW:0] next_row[2:0];
```

Listing 2: Buffer interno per la nuova riga in *window_ buffer.sv*

Nella figura 3.6 è visibile il funzionamento, in particolare sono evidenziati i vari cicli di esecuzione con un colore diverso:

- L'azzurro rappresenta il caricamento del window buffer e del weight buffer.
- In giallo sono evidenziati i tre cicli di clock in cui vengono caricati tre pacchetti da 128 bit relativi ai byte dei 16 canali dei pixels della nuova riga.
- Il rosso rappresenta il quarto ciclo dove viene traslata la finestra di un pixel verso il basso e viene fatta la store in memoria dell'output da 128 bit.

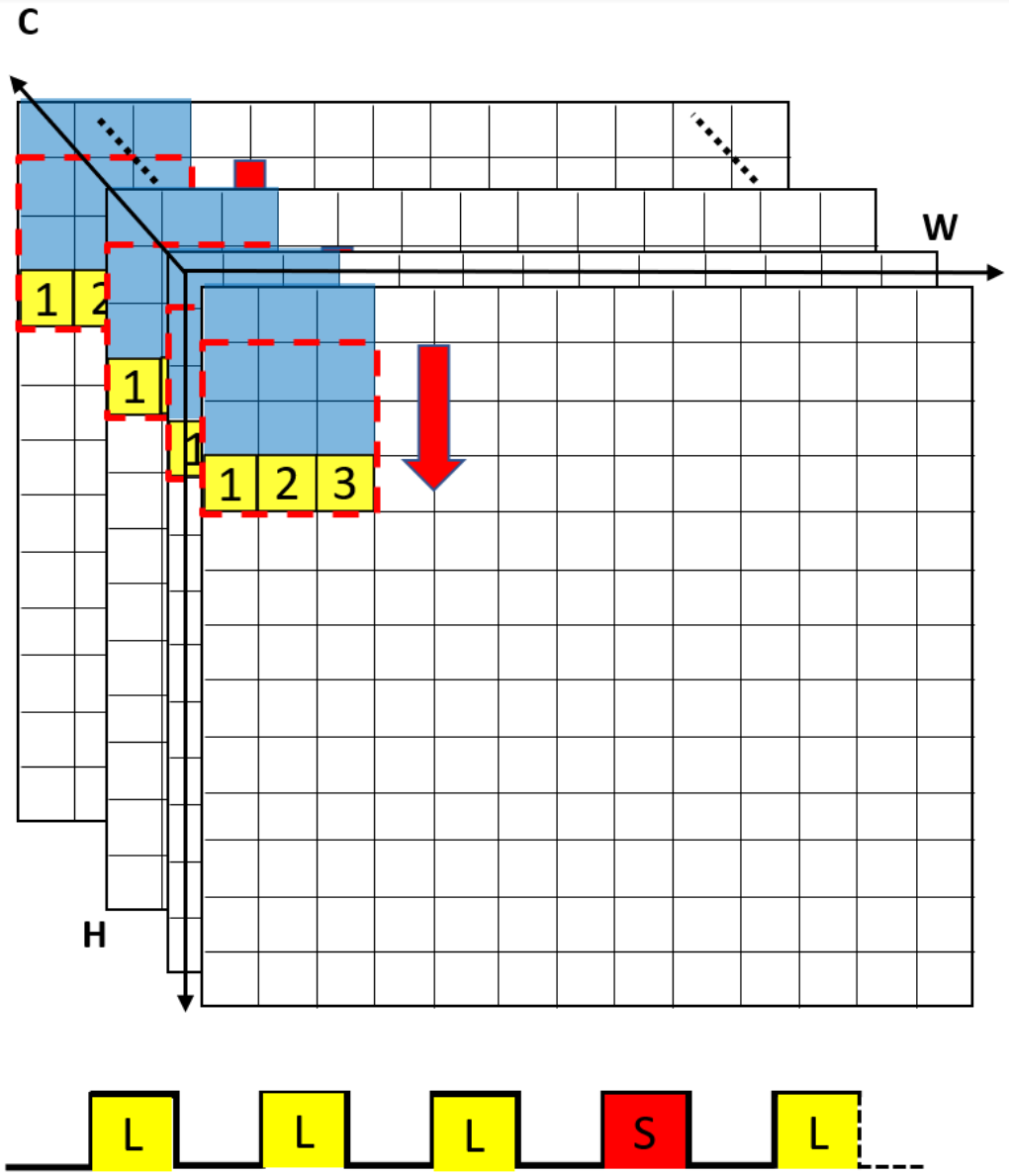


Figura 3.6: Cicli di funzionamento del Window Buffer

Questa è l'istanza del modulo in verilog:

```
window_buffer #(
                .NB_CHANNELS(NB_CHANNELS),
                .BW          (BW         )
        ) i_window_buffer (
                .clk_i           (clk_i               ),
                .rst_ni          (rst_ni              ),
                .clear_i         (ctrl_i.clear_window ),
                .enable_i        (ctrl_i.enable       ),
                .load_window     (ctrl_i.load_window  ),
                .shift           (start_shift         ),
                .load_next_row   (load_next_row       ),
                .block_in        (input_data          ),
                .block_in_valid  (data_in.valid       ),
                .mod_enable      (ctrl_i.mod_enable   ),
                .module_ch       (ctrl_i.mod_channels ),
                .activators      (activators          ),
                .end_load        (end_window          )
        );
```

Listing 3: Modulo del window buffer

Mentre il weight buffer è semplicemente un tensore di dimensioni 3x3x16 che viene aggiornato ad inizio computazione.

```
weight_buffer #(
                .NB_CHANNELS (NB_CHANNELS),
                .BW          (BW         )
        ) i_weight_buffer (
                .clk_i           (clk_i               ),
                .rst_ni          (rst_ni              ),
                .clear_i         (ctrl_i.clear        ),
                .enable_i        (ctrl_i.enable       ),
                .load_en_weights (new_weigths         ),
                .block_in        (input_data          ),
                .block_in_valid  (data_in.valid       ),
                .mod_enable      (ctrl_i.mod_enable   ),
                .module_ch       (ctrl_i.mod_channels ),
                .weights         (weights             ),
                .end_load        (end_weights         )
        );
```

Listing 4: Modulo del weight buffer

3.2.2 Unità di accumulazione MAC, shifting dei risultati e Relu

Come si può osservare dall'architettura dell'engine in figura 3.5 è presente la rete di Multiply accumulation (MAC). Questa rete rappresenta il calcolo computazionale principale dell'intero acceleratore ed è composta da 36 mul per le moltiplicazioni tra attivatori e pesi e un sommatore ad albero per il risultato finale. Avendo 36 mul è possibile calcolare l'output di un blocchetto da 3x3x4 ogni ciclo di clock. In questo modo in quattro cicli calcolo le MAC per tutto il tensore nel window buffer ed eseguo la store in memoria. Nella figura 3.6 sono evidenziati i cicli di clock: nei primi 3 eseguiamo le load dalla memoria dei pixel della nuova riga mentre nella quarta la store del risultato. In figura 3.7 si vede in dettaglio il funzionamento della rete di MAC.

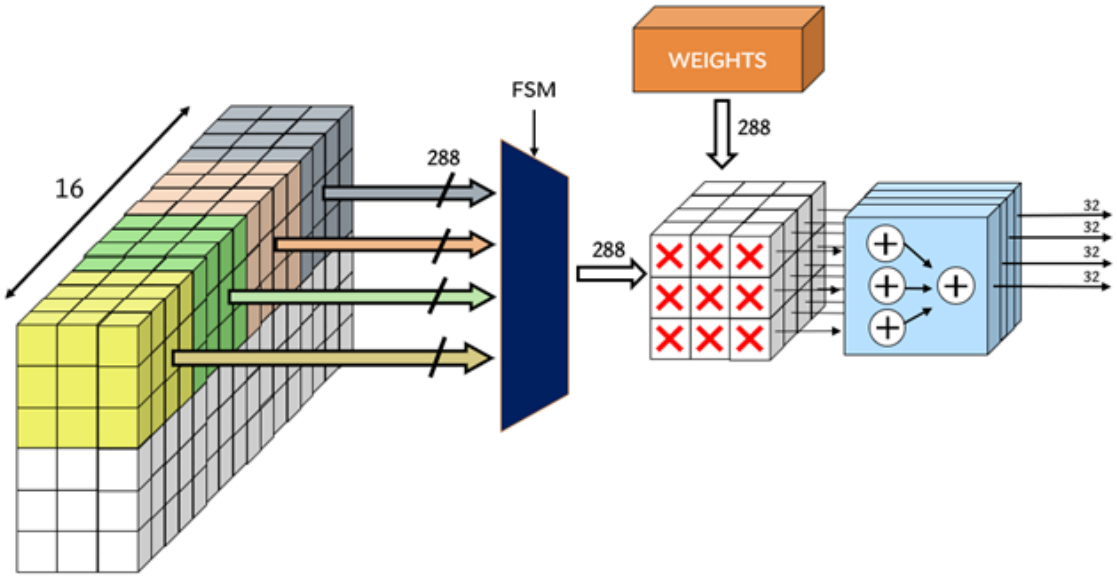


Figura 3.7: MAC

Dal window buffer vengono caricati ogni ciclo i "blocchetti" da 324 bits, considerando che i pixel sono a 9 bits con il bit di segno aggiunto, selezionati da un segnale di controllo proveniente dalla FSM. Questi sono poi moltiplicati pixel per pixel con i pesi del weight buffer: per ogni pixel di un determinato canale viene eseguito il prodotto con il peso del relativo kernel. Infine i risultati parziali sono sommati tra loro riga per riga fino ad ottenere 16 valori di accumulazione relativi ai 16 canali calcolati. Questi risultati sono salvati in un array multidimensionale da 32 bits.

```
//accumulation register
        logic [NB_CHANNELS-1:0][31:0] accumulator;
```

Listing 5: registro di accumulazione in *depthwsie_ engine.sv*

I risultati di accumulazione da 32 bits passeranno poi nella rete di ReLU che farà passare solo i valori positivi mentre i negativi li porterà a 0, poi uno shifter

programmabile e un blocco di clip8 riporteranno i valori a 8 bit saturando i valori più alti.

3.2.3 Macchina a stati

Per effettuare questo tipo di esecuzione è stata pensata una macchina a stati visibile nel diagramma sottostante.

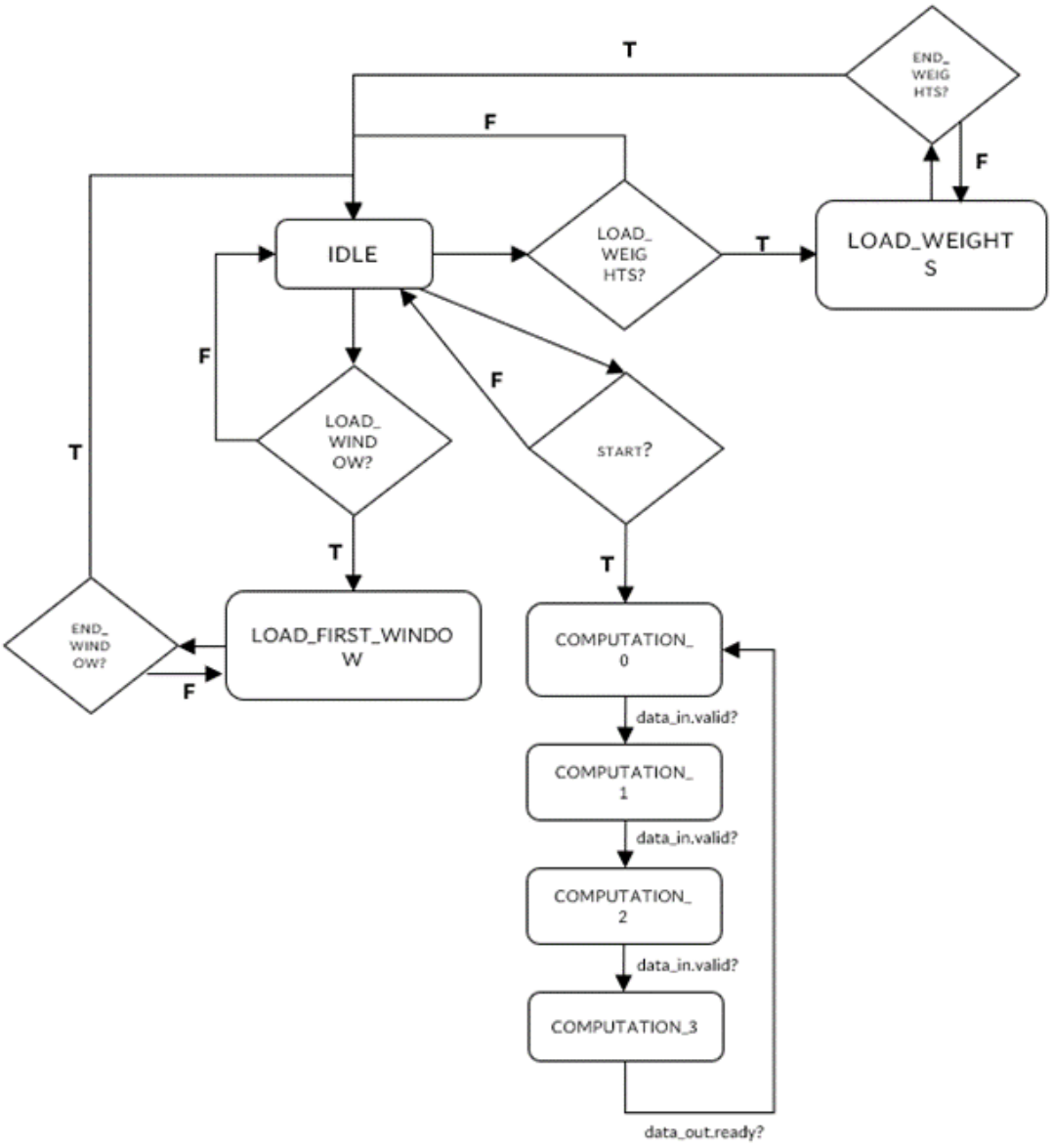


Figura 3.8: FSM engine

Dallo stato iniziale di `IDLE` si passa agli stati di `LOAD_WEIGHTS` o di `LOAD_FIRST_WINDOW` se dalla rete di controllo arrivano i segnali di *ctrl_i.load_weights* o di *ctrl_i.load_window*. La macchina a stati rimane qui finchè non arriva il segnale di *end_weights* e di *end_window* provenienti da due contatori interni rispettivamente nel weight_buffer e nel window_buffer. In questi due stati è anche gestito il protocollo di valid e ready per dialogare con lo streamer.

Dalla rete di control arriva poi il segnale di *start* che fa partire il calcolo vero e proprio e la macchina a stati entra nel computation loop: un loop di 4 cicli di clock

per il calcolo della convoluzione sulla prima colonna dell'immagine. Una volta finito il calcolo della prima colonna dalla rete di control arriva il segnale di *ctrl.clear* per riportare la FSM allo stato di IDLE.

3.3 Depthwise Streamer

Come è stato evidenziato ad inizio capitolo lo streamer è l'interfaccia interposta tra l'engine e la memoria tcdm. Per controllare lo streaming di dati sono stati pensati vari moduli che si possono trovare all'interno della repository *hwpe_stream* su GitHub. Tra questi ci sono:

- basic: moduli base per controllare il flow dei dati con interfacce di hwpe_stream tra cui mux, demux, merge, split, fence e buffer.
- fifo: buffer di first in first out per disaccoppiare gli accessi in memoria con le richeste di dati dall'engine.
- tcdm: moduli per organizzare il flusso di dati dalla memoria con interfacce tcdm.
- streamer: moduli di confine tra l'interfaccia tcdm e di HWPE-stream. Servono per trasformare il flusso di dati e per organizzare le richieste alla memoria. Tra questi:
  - source: da uno stream di load tcdm generato con indirizzi in memoria provenienti dall'addressgen genera un HWPE-stream
  - sink: genera una serie di store in memoria da un HWPE-stream ad indirizzi generati dall'addressgen.
  - addressgen: genera una serie di indirizzi calcolati automaticamente seguendo un pattern 3D, molto utile quando si tratta con tensori in memoria.
  - strbgen: genera uno strobe per indirizzi non allineati in memoria.
  - sink_realign: reallinea l'HWPE-stream per store in memoria non allineate.
  - source_realign: reallinea l'HWPE-stream per load in memoria non allineate.

3.3.1 Architettura dello streamer

Per quanto riguarda lo streamer dell'acceleratore Depthwise il modulo è il seguente:

```
module depthwise_streamer #(
  parameter  int unsigned DW         = 128,
  parameter  int unsigned FIFO_DEPTH = 2
) (
  input  logic                    clk_i     ,
  input  logic                    rst_ni    ,
  input  logic                    test_mode_i,
  input  logic                    enable_i  ,
  input  logic                    clear_i   ,
  // Engine input activators + weights signals (output for the streamer)
  hwpe_stream_intf_stream.source data_in_o ,
  // Engine input weights + HS signals (output for the streamer)
  hwpe_stream_intf_stream.sink   data_out_i,
  // TCDM interface between the streamer and the memory
  hci_core_intf.master           tcdm ,
  // Control signals
  input  ctrl_streamer_t         ctrl_i,
  output flags_streamer_t        flags_o
);
```

Listing 6: Modulo dello streamer in *depthwise_streamer.sv*

Come si può vedere la data width del trasferimento è di 128 bits che passano attraverso la TCDM in una *hci_core_interface* (figura 3.3), per poi arrivare come output in data_in_o in una interfaccia *hwpe_stream* (figura 3.2). Il modulo ha anche come input un *ctrl_i*, un segnale di controllo proveniente dal controllore e come output *flags_o* per condividere il proprio stato agli altri moduli dell'acceleratore.

Per il funzionamento dell'engine secondo la logica pensata nell'accelerator Depthwise servono due stream di dati dalla memoria, uno che richiede le load e uno per le store. Questi accessi saranno entrambi da 128 bits. Visto che secondo il flusso di dati le load e le store avvengono su due cicli di clock diversi basta solamente una interfaccia tcdm da 128 bits. Per gestire gli accessi in memoria è stato usato un mux dinamico. Questo riconosce in automatico, attraverso i req e grnt dell'interfaccia, quando avviene una richiesta e gli dà la priorità.

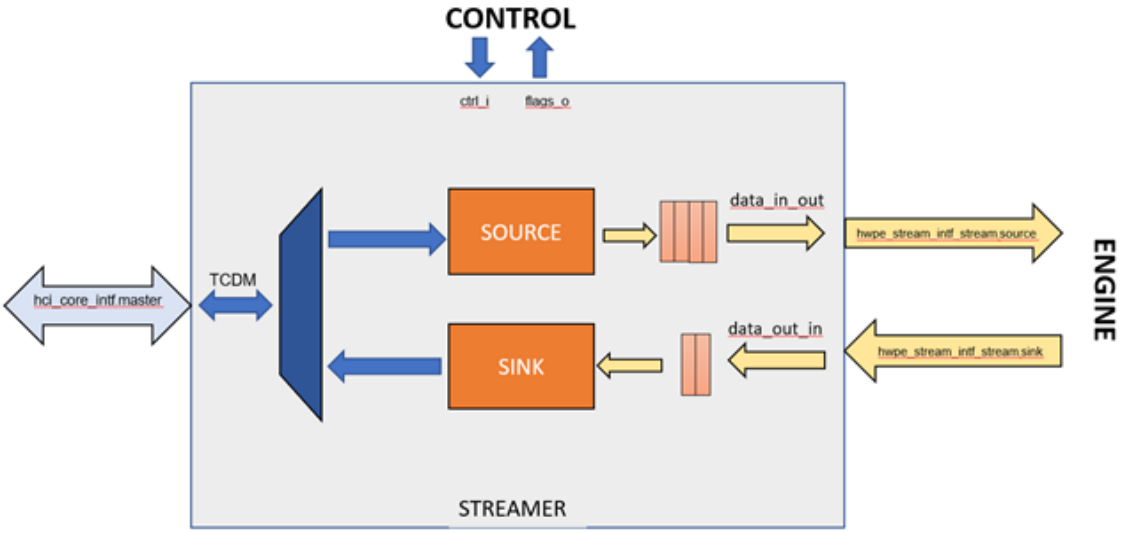


Figura 3.9: Streamer

Al confine tra le interfacce tcdm (frecce blu) e HWPE-stream (frecce gialle) sono presenti i moduli di source e sink. All'interno di questi si trovano i moduli di addressgen, programmati dalla rete di control, fondamentali per organizzare gli accessi in memoria. (Discussi nella prossima sezione) A lato engine sono presenti anche due FIFO, quella per i dati in ingresso all'engine, larga 4 registri, mentre quella prima del sink è larga 2. Infine sono presenti i segnali di controllo e i flags dallo/verso lo streamer.

3.4 Depthwise Control

La rete di controllo per l'acceleratore Depthwise è istanziata così:

```
module depthwise_ctrl #(
  parameter int unsigned NUM_CHANNELS = 16 ,
  parameter int unsigned BW           = 128,
  parameter int unsigned N_CORES      = 8  ,
  parameter int unsigned N_CONTEXT    = 2  ,
  parameter int unsigned N_IO_REGS    = 16 ,
  parameter int unsigned ID_WIDTH     = 3
) (
  // global signals
  input  logic                                clk_i           ,
  input  logic                                rst_ni          ,
  input  logic                                test_mode_i     ,
  output logic                                clear_o         ,
  // events
  output logic [N_CORES-1:0][1:0]             evt_o           ,
  // ctrl & flags
  output ctrl_streamer_t                      ctrl_streamer_o ,
  input  flags_streamer_t                     flags_streamer_i,
  output ctrl_engine_t                        ctrl_engine_o   ,
  input  flags_engine_t                       flags_engine_i  ,
  // periph slave port
  hwpe_ctrl_intf_periph.slave                 periph
);
```

Listing 7: Istanza del controllore in *depthwise_control.sv*

Come si può notare ha in input e in output i flags e i segnali di controllo provenienti dallo streamer e dall'engine. In più ha una porta "periph", con un interfaccia *hwpe_ctrl_intf_periph.slave*, usata per programmare un register file mappato in memoria.

3.4.1 Programmazione registri

In *depthwise_package.sv* ci sono le definizioni di tutte le *typedef_structure* utilizzate per definire i segnali utilizzati per la comunicazione tra i componenti dell'HWPE. Nello stesso file sono anche definiti tutti i registri che vengono utilizzati per la programmazione del funzionamento dell'acceleratore.

```
// registers in register file
parameter int unsigned DEPTHWISE_REG_X_ADDR       = 0;
parameter int unsigned DEPTHWISE_REG_W_ADDR       = 1;
parameter int unsigned DEPTHWISE_REG_Y_ADDR       = 2;
parameter int unsigned DEPTHWISE_REG_IMG_SIZE_INP = 3;
// [15:0] WIDTH, [31:16] HEIGHT
parameter int unsigned DEPTHWISE_REG_CHANNELS     = 4;
parameter int unsigned DEPTHWISE_CONFIG           = 5;
// [15:0] SHIFT, [16] RELU, [17] CLIP8
parameter int unsigned DEPTHWISE_D2_STRIDE_INP    = 6;
parameter int unsigned DEPTHWISE_D2_STRIDE_OUT    = 7;
```

Listing 8: Istanza del controllore in *depthwise_control.sv*

Tutti questi registri sono job-dependent, quindi possono tutti essere impostati via software prima di avviare il lavoro. Ci sono i registri DEPTHWISE_REG_X_ADDR, DEPTHWISE_REG_W_ADDR, DEPTHWISE_REG_Y_ADDR che servono per immagazzinare i puntatori in memoria dei dati di input, pesi e output che saranno poi utili nella programmazione dell'addressgen. DEPTHWISE_REG_IMG_SIZE_INP è un registro per salvare la grandezza dell'immagine mentre DEPTHWISE_REG_CHANNELS è quello per il numero di canali.
DEPTHWISE_CONFIG è un registro di configurazione composto da 16 bits usati per impostare lo shift del dato in uscita, il diciassettesimo per utilizzare la ReLU e il diciottesimo per la funzione di clip8. Infine sono presenti due registri: DEPTHWISE_D2_STRIDE_INP e DEPTHWISE_D2_STRIDE_OUT per impostare lo STRIDE di input e di output per programmare l'addressgen. Per capire bene il motivo di questi registri bisogna scendere nel dettaglio nel funzionamento dell'addressgenerator.

3.4.2 Address Generator

Il modulo *hwpe_stream_addressgen_v3.sv* viene utilizzato per generare indirizzi per le load e le store del flusso di dati nello streamer. L'addressgen può essere utilizzato per generare indirizzi da uno spazio tridimensionale formato da D1, D2 e D3. D1 è la direzione nello spazio 3D dove i dati sono continui in memoria, nel caso dell'acceleratore Depthwise è quella dei canali. D2 e D3 sono invece le direzioni di width e height.

L'addressgen è programmabile dal controllore attraverso dei segnali che arrivano allo streamer che impostano:

- tot_len: il numero totali di trasferimenti da effettuare.
- d0_stride: il numero di byte consecutivi in memoria in cui il puntatore si muove tra un trasferimento e il prossimo in direzione d0.
- d0_len: quanti trasferimenti sono da eseguire in direzione d0.
- d1_stride: il numero di byte consecutivi in memoria in cui il puntatore si muove tra un trasferimento e il prossimo in direzione d1.

- d1_len: quanti trasferimenti sono da eseguire in direzione d1.
- d2_stride: il numero di byte consecutivi in memoria in cui il puntatore si muove tra un trasferimento e il prossimo in direzione d2.
- base_address: l'indirizzo di partenza in cui punta il puntatore.

Nel nostro caso l'addressgen interno al source viene programmato due volte, una per caricare i pesi nel weight buffer e una per gli attivatori nel window buffer. Per gestire lo scorrimento del window buffer nell'immagine è stato programmato in modo che trasli verso il basso finchè non si raggiunge il limite dell'immagine da elaborare, quindi finchè il contatore interno non raggiunge il valore di tot_len. Nel codice sottostante è presente la programmazione dell'addresgen per il window buffer.

```
ctrl_streamer_o.data_in_source_ctrl.addressgen_ctrl.tot_len = ctrl_i.rows * 3;
ctrl_streamer_o.data_in_source_ctrl.addressgen_ctrl.d0_stride = 0;
ctrl_streamer_o.data_in_source_ctrl.addressgen_ctrl.d1_stride = ctrl_i.channels;
ctrl_streamer_o.data_in_source_ctrl.addressgen_ctrl.d0_len = 1;
ctrl_streamer_o.data_in_source_ctrl.addressgen_ctrl.d1_len = 3;
ctrl_streamer_o.data_in_source_ctrl.addressgen_ctrl.d2_stride = ctrl_i.d2_stride_inp;
//(ctrl_i.columns - 3) * ctrl_i.channels;
ctrl_streamer_o.data_in_source_ctrl.addressgen_ctrl.dim_enable_1h = '1;
ctrl_streamer_o.data_in_source_ctrl.addressgen_ctrl.base_addr = result_x_address;
```

Listing 9: Programmazione dell'addressgen in *depthwise_fsm.sv*

Come si può osservare dal codice in alto d0_stride è settato a 0 mentre d1_stride corrisponde al numero di canali dell'immagine perchè corrisponde al numero di byte in memoria che sono presenti tra due pixel vicini in una riga dello stesso canale. Il valore di d0_len è uguale a 1 perchè effettuiamo solamente un trasferimento da 128 bit che corrisponde a 16 canali nella direzione d0, per poi spostarci di un pixel in direzione d1; per questo d1_len è uguale a 3. Dopodichè ci si sposta alla riga sottostante attraverso il d2_stride. Questo valore viene calcolato moltiplicando il numero di colonne dell'immagine meno 3 (larghezza del window buffer) per il numero di canali ed è impostato in un registro programmabile perchè sarebbe troppo dispendioso, in termini di tempo, una moltiplicazione tra due parametri in questo punto dell'architettura. L'ultimo valore è il base address che è stato impostato al parametro result_x_address, una valore che deriva da una logica di offset interna alla macchina a stati del controllore: ogni volta che il window buffer arriva alla fine di una colonna dell'immagine finisce il trasferimento dell'addressgen, questo valore viene aggiornato e il base_addr viene riscritto sia in input che in output per il prossimo trasferimento. Questo avviene anche quando si arriva alla fine dell'ultima colonna, in questo caso il puntatore sarà da spostare al diciassettesimo canale in quanto il nostro acceleratore è in grado di effettuare la convoluzione di massimo 16 canali alla volta.

In *depthwise_fsm.sv* è stata integrata anche una logica per calcolare un numero di canali non multiplo di 16. Sono stati inseriti due registri, uno registro di divisione e uno di modulo. Il registro di divisione contiene al suo interno semplicemente il quoziente tra il numero di canali totali e 16. Nel registro di modulo viene salvato invece il modulo della divisione, cioè quanti canali avanzano dopo che viene calcolato

l'ultimo blocco da 16. In questo modo tramite un contatore all'interno della FSM, una logica nell'engine che porta a 0 i bits dei canali che non servono e lo strobe in output che fa in modo che vengano scritti solo i dati agli indirizzi corretti, posso gestire anche un numero di canali non multiplo di 16. I canali però devono comunque essere un multiplo di 4 perchè se accedo ad indirizzi non allineati in memoria nello streamer è presente una logica che li reallinea in automatico, con la conseguenza di una perdita di dati.

3.4.3 Macchina a stati

La macchina a stati della rete di control è leggermente più complessa rispetto a quella nell'engine perchè di occupa di gestire i vari moduli all'interno dell'acceleratore.

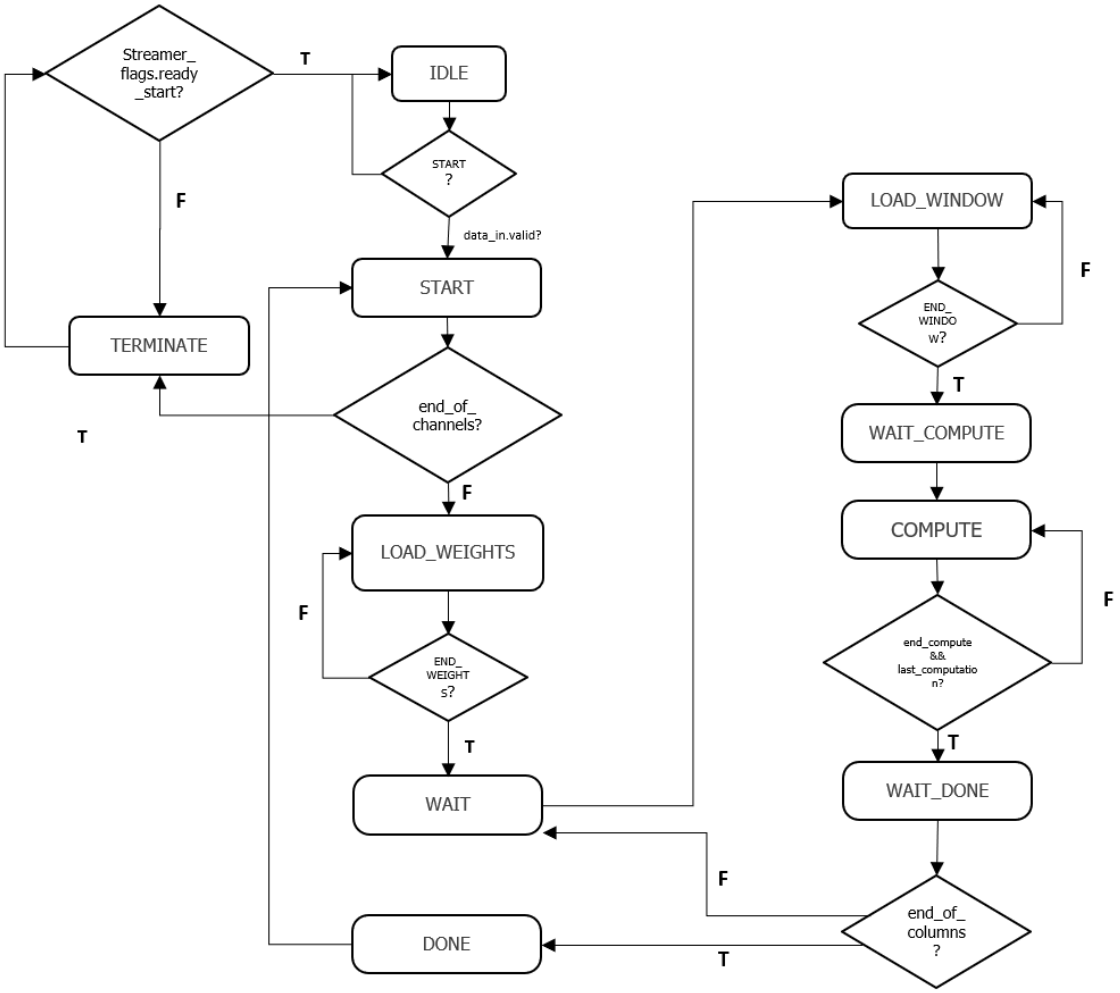


Figura 3.10: FSM control

Come si può osservare in figura dallo stato di IDLE si passa allo stato di START quando arriva il segnale *flags_ slave_ i.start*. Questo input proviene dal register file e viene impostato al valore 1 via software attraverso la scrittura nel registro DEPTH-WISE_TRIGGER, un registro job-indipendent del register file che viene usato come

"trigger" per far partire l'acceleratore. Dallo stato di start la macchina a stati controlla lo stato dei contatori, in particolare viene salvato in *d0_x_offset* il valore di offset lungo D0 per il calcolo del base address dell'addressgen. Se questo valore è più alto del numero dei canali da calcolare la FSM passa allo stato TERMINATE, al contrario continua in LOAD_WEIGHTS e viene fatta la richiesta di inizio trasferimento all'addressgen attraverso il segnale *ctrl_streamer_o.data_in_source_ctrl.req_start*. In LOAD_WEIGHTS viene impostato ad 1 il segnale di *weight_stream* per programmare l'addressgen per il caricamento dei pesi, e il segnale di *ctrl_engine_o.load_weights* per passare allo stato di LOAD_WEIGHTS anche nell'engine. Quando è finito il caricamento dei pesi *weight_stream* torna a 0, in questo modo l'addressgen è programmato per il trasferimento degli attivatori, in più arriva *flags_engine_i.end_weights* dall'engine che porta prima la rete ad uno stato di WAIT, dove viene reimpostato ad 1 il segnale *ctrl_streamer_o.data_in_source_ctrl.req_start*, poi allo stato di LOAD_WINDOW che alza il segnale di *ctrl_engine_o.load_window* per l'engine. Quando è terminato il caricamento del window buffer arriva il segnale *flags_engine_i.end_window* e si passa allo stato di WAIT_COMPUTE dove viene impostato ad uno il segnale di *ctrl_streamer_o.data_out_sink_ctrl.req_start* per far partire l'addressgen interno al sink per le store in memoria. Il ciclo successivo si passa allo stato di COMPUTE. In questo stato l'engine shifta il kernel sulla prima colonna da 3 pixel dell'immagine e, una volta arrivato nell'ultima posizione, viene alzato ad 1 il segnale *ctrl_engine_o.last_computation*, viene aggiornato il contatore di colonne nell'immagine e si passa allo stato di WAIT_DONE. In questo stato viene effettuato il controllo con il contatore delle colonne dell'immagine, se il window buffer è arrivato all'ultima colonna dell'immagine si si manda il segnale di *ctrl_engine_o.clear* all'engine che lo riporta allo stato di IDLE e la FSM passa allo stato di DONE, al contrario si ritorna allo stato di WAIT e si prosegue il calcolo della colonna successiva. Infine dallo stato di DONE si passa semplicemente allo stato di START. Si prosegue poi con il calcolo dei successivi canali.

Capitolo 4

Risultati sperimentali

In questo capitolo verranno esposti i risultati in simulazione e la sintesi dell'acceleratore descritto nel capitolo precedente, per fare questo sono stati usati i programmi Questasim 10.7b e Synopsis Design Compiler.

4.1 Ambiente di simulazione con HWPE-TB

In questa sezione saranno discussi i risultati in simulazioni ottenuti con QuestaSim usando l'HWPE-TB, un testbench pensato appositamente per gli acceleratori HWPE. Questo testbench fornisce un ambiente autonomo per testare l'acceleratore. Infatti istanzia delle dummy memories per i dati e le istruzioni e un semplice core zero-riscy per eseguire istruzioni da test scritti in C. HWPE-TB si può trovare in GitHub in *pulp-platform/hwpe-tb*. Il testbench ha una forte dipendenza dalla PULP-SDK. SDK è l'abbreviazione di Software Development Kit ed è un insieme di strumenti che consente lo sviluppo per una specifica piattaforma hardware. Per usarla basta semplicemente clonare la repository da GitHub nell'ambiente di sviluppo, avere il compilatore RISC-V GNU Toolchain correttamente installato e infine eseguire il comando di source corrispondente alla piattaforma che utilizziamo, nel nostro caso *source configs/pulp-open.sh*.

4.1.1 Test in HWPE-TB

Per testare l'acceleratore sono state modificate varie parti di codice nella cartella hwpe-tb. Il testbench è composto da tre parti principali, una hardware, che instanzia i vari componenti da usare, una software, per dare le istruzioni all'hardware instanziato, e un Makefile per facilitarne l'esecuzione. Il makefile ha al suo interno un comando di *update-ips* che scarica in automatico tramite GitHub tutti gli ips che sono stati definiti nel file *ips_list.yml*. Per includere il nostro acceleratore nel testbench come prima cosa sono stati aggiunti in questa lista i riferimenti alla repository su GitHub e come seconda cosa è stato sostituito in *tb-hwpe.sv* l'istanza dell'acceleratore MAC (esempio di acceleratore precedente) con il nostro acceleratore Depthwise. Nella cartella software sono tre i file che sono stati modificati: *hal_hwpe.h*,*archi_hwpe.h* e *tb-hwpe.c*. In *archi_hwpe.h* sono stati definiti i registri usati per configurare l'acceleratore e il loro indirizzo in memoria.

```
#define DEPTHWISE_ADDR_BASE 0x100000

// commands
#define DEPTHWISE_TRIGGER        0x00
#define DEPTHWISE_ACQUIRE        0x04
#define DEPTHWISE_FINISHED       0x08
#define DEPTHWISE_STATUS         0x0c
#define DEPTHWISE_RUNNING_JOB    0x10
#define DEPTHWISE_SOFT_CLEAR     0x14
#define DEPTHWISE_SWSYNC         0x18
#define DEPTHWISE_URISCY_IMEM    0x1c

// job configuration
#define DEPTHWISE_REGISTER_OFFS       0x40
#define DEPTHWISE_REG_X_ADDR          0x00
#define DEPTHWISE_REG_W_ADDR          0x04
#define DEPTHWISE_REG_Y_ADDR          0x08
#define DEPTHWISE_REG_IMG_SIZE_INP    0x0c
#define DEPTHWISE_REG_CHANNELS        0x10
#define DEPTHWISE_CONFIG              0x14
#define DEPTHWISE_D2_STRIDE_INP       0x18
#define DEPTHWISE_D2_STRIDE_OUT       0x1c
```

Listing 10: Indirizzo dei registri in *archi_hwpe.h*

In *hal_hwpe.h* invece ci sono definite le variabili globali come la grandezza dell'immagine, e le funzioni che scrivono o leggono i valori dai registri. In *tb-hwpe.c* è presente il codice di testing vero e proprio. In questo definiamo dei puntatori per l'immagine da calcolare e i pesi del kernel che puntano agli array definiti in *inc/depthwise_input.h* e in *depthwise_output.h* per i risultati. Dopodichè per programmare l'acceleratore vengono scritti i valori di programmaziione anll'interno dei registri con i valori corrispondenti: i puntatori nei registri degli indirizzi e i valori di configurazione negli altri. In questo test è stata considerata una immagine con 25 righe, 20 colonne e 24 canali. Per confrontare i risultati della simulazione osserviamo le forme d'onda da ModelSim.

4.1.2 Fase di caricamento

In questa fase vengono caricati i pesi e gli attivatori nei weight buffer e nel window buffer rispettivamente. Dalla figura 4.1 si può vedere che, quando la FSM del controllore passa nello stato di LOAD_WEIGHTS, avviene il caricamento del weight buffer. Il delay di 4 cicli di clock è dovuto ai 4 registri di FIFO nello streamer. Dopo 9 trasferimenti da 128 bits torna il controllo alla macchina a stati che cambia stato in LOAD_WINDOW e si ripete in modo analogo il caricamento.

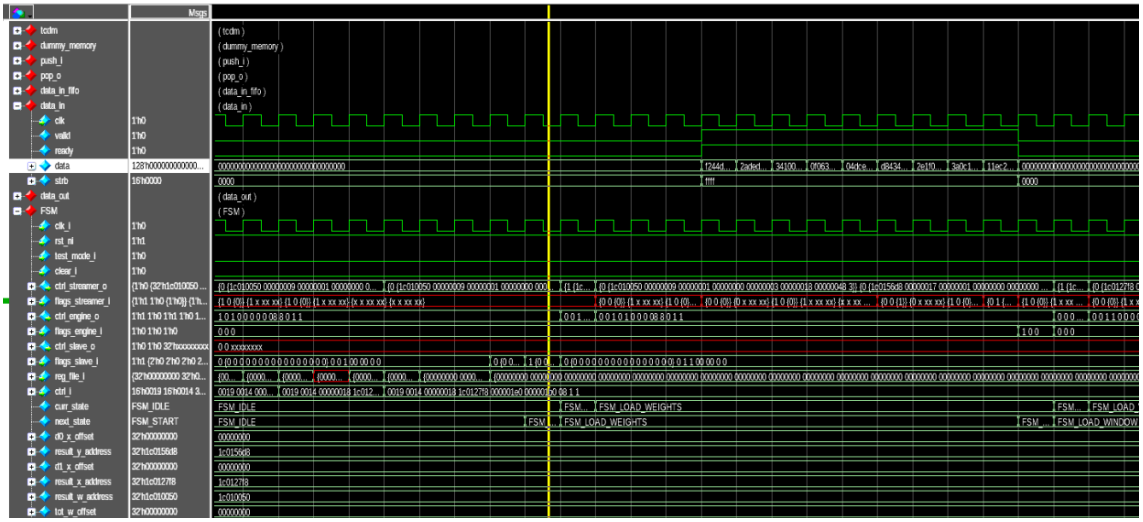


Figura 4.1: Caricamento dei pesi

4.1.3 Fase di calcolo

Nella fase di calcolo la FSM del controllore è nello stato COMPUTE quindi il controllo passa all'engine. Nell'engine inizia così il loop dei 4 stati di COMPUTATION_0/1/2/3 dove vengono calcolati i risultati parziali dalla rete di MAC dei 16 canali della finestra 3x3 dell'immagine, in blocchi di 4 canali alla volta.

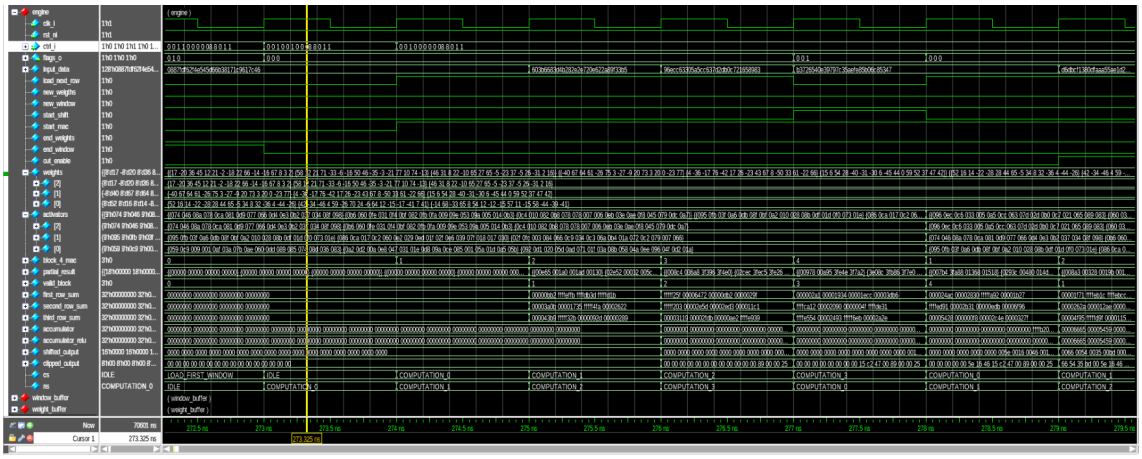


Figura 4.2: Loop di computazione

In figura 4.2 si può vedere chiaramente il loop dei 4 stadi e come il window buffer venga aggiornato alla fine del loop, questo perchè la finestra viene traslata nell'immagine in modo figurato verso il basso, nella simulazione le righe del window buffer sono traslate verso l'alto e viene caricata la nuova nella riga rimasta vuota.

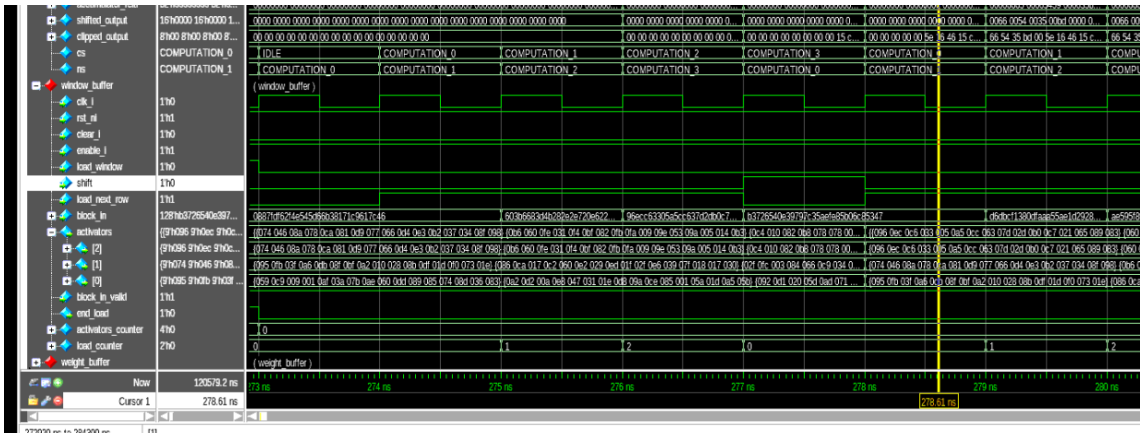


Figura 4.3: Loop nel window buffer

Alla fine della quarta computazione viene portato i 128 bit del segnale di *clipped_output* in *data_out* e, alzando il segnale di valid, avviene il trasferimento verso lo streamer. Finita la prima colonna dell'immagine, quindi dopo 23 cicli di computazione perchè, non essendoci padding, la dimensione dell'immagine diminuisce di 2, viene ricaricata per intero la nuova finestra del window buffer corrispondente a quella iniziale spostata di un pixel verso destra.

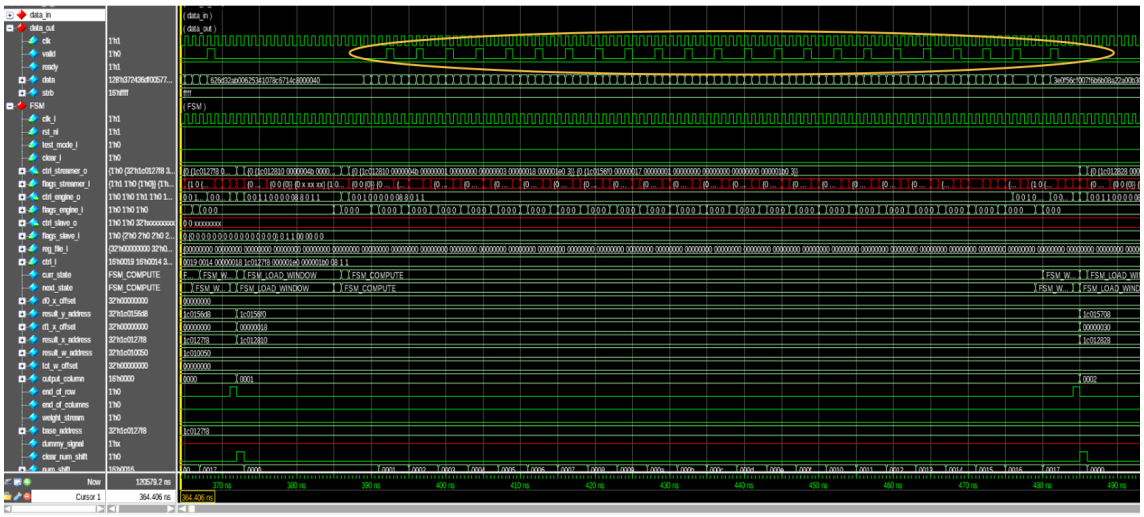


Figura 4.4: In evidenza i dati in output

Questa operazione avviene per ogni colonna dell'immagine, quando viene raggiunta l'ultima colonna viene effettuato l'ultimo ciclo di computazione e dopodichè, se ci sono altri canali da calcolare dell'immagine, vengono ricaricati sia i pesi nel weight buffer relativi ai nuovi kernel sia i nuovi pixel che comporanno il window buffer. L'operazione si ripete finchè non si esauriscono i canali.

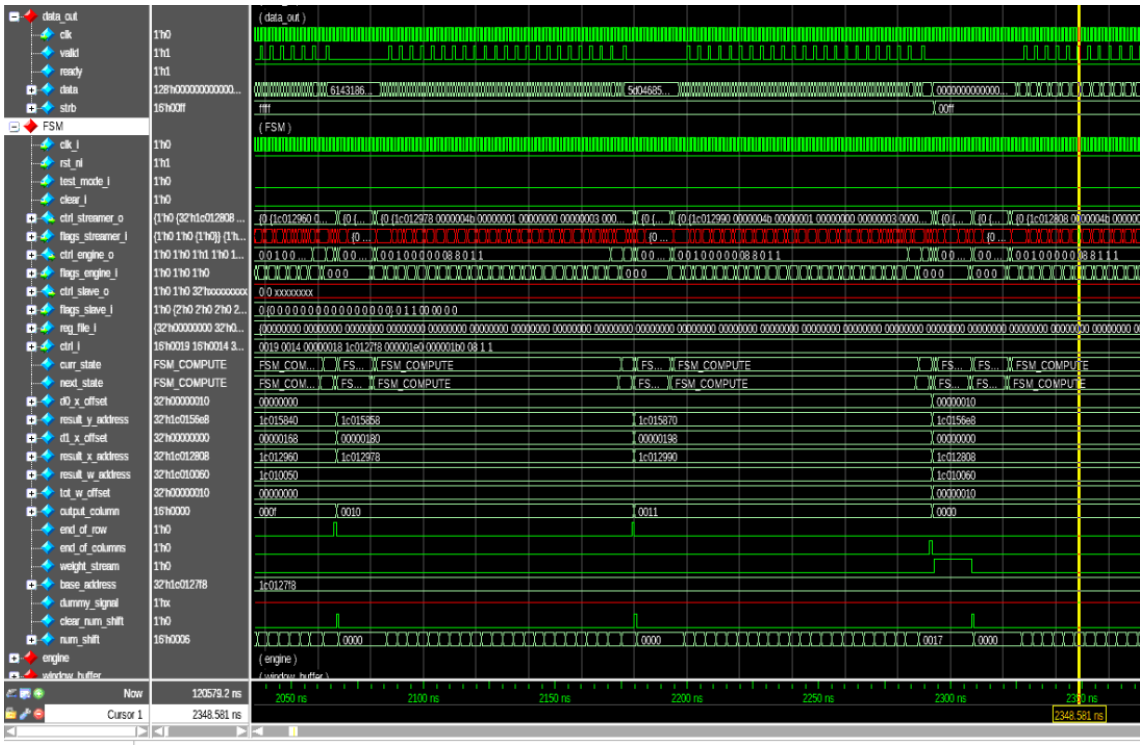


Figura 4.5: Fine calcolo dei primi 16 canali

4.1.4 Stalli in memoria

Quando si eseguono simulazioni software bisogna anche tenere conto che la memoria potrebbe subire degli stalli dovuti a più fattori: accessi contemporanei in memoria, accessi falliti, criticità su dati, ecc. Per questo l'HWPE-TB incorpora al suo interno la possibilità di inserire degli stalli in memoria attraverso il parametro P_STALL nel MakeFile. Inizialmente infatti erano stati inseriti solamente due registri di pipeline nello streamer, ma dopo aver effettuato questa simulazione con P_STALL = 0.2 si è notato che la rete di engine caricava risultati non corretti e questo portava alla corruzione di tutti gli output. E' bastato solamente aggiungere due registri di pipeline tra il source e l'engine in quanto in un ciclo di computazione vengono effettuate 3 load e una sola store.

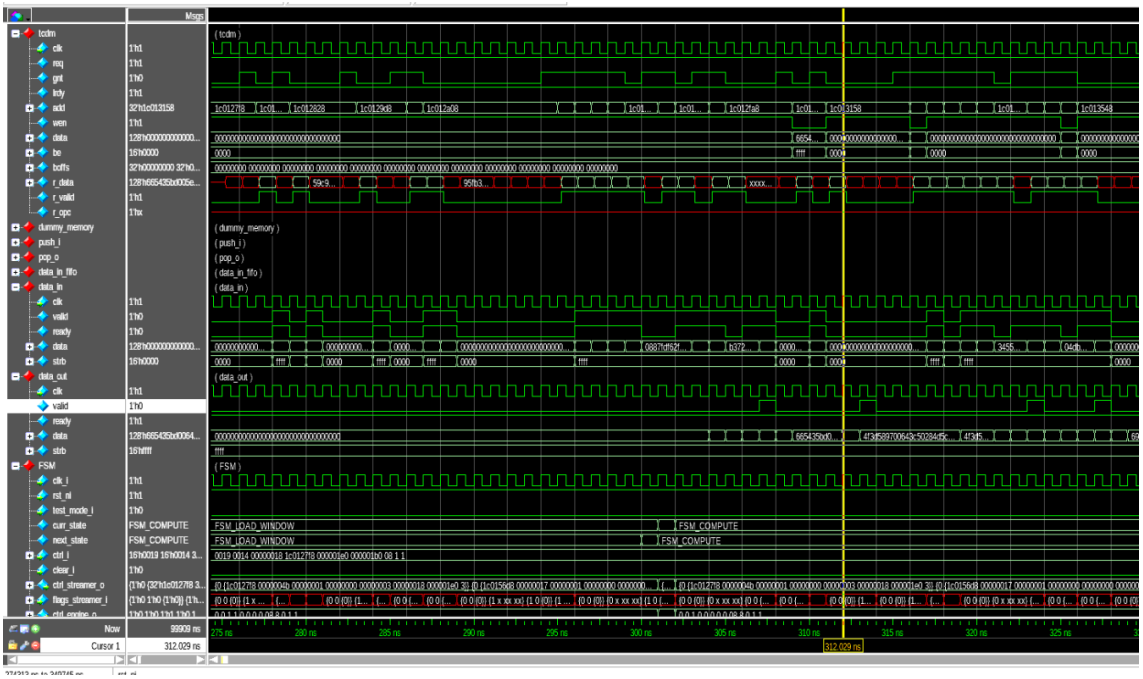


Figura 4.6: Simulazione con $P\_STALL = 0.2$

4.2 Risultati in area e timing

4.2.1 Setup sintesi

La sintesi dell'acceleratore è stata effettuata con il software Synopsys Design Compiler, un tool per effettuare la sintesi partendo da una rete rtl. Per la sintesi del nostro acceleratore è stato scelta come tecnologia quella TSCM a 65 nm in quanto l'acceleratore poi è stato inserito all'interno del nuovo chip di ricerca "Darkside", l'ultimo chip basato su OpenPulp del laboratorio di Bologna in collaborazione con l'ETH di Zurigo che verrà presentato nel capitolo successivo.

Per effettuare la sintesi è stato modificato il file *.synopsys_dc.setup* in cui sono inserite le librerie che si possono usare differenziate per:

- Tecnologia: nel nostro caso usiamo le cln65lp.
- Threshold Voltage: è il minimo voltaggio gate to source per creare il canale conduttivo.
- Process corner: tre parametri divisi in ss (slow slow), tt (typical typical), ff (fast fast) per rappresentare la variazione dei parametri nel processo industriale.
- Voltaggio: il valore in volt dell'alimentazione.
- Temperatura: la temperatura in cui viene simulata la rete sintetizzata.

Nel nostro caso abbiamo effettuato la sintesi con la libreria *sc8_cln65lp_base_lvt_ss_typical_max_1p08v_125c* che usa i parametri in low voltage threshold (lvt), corner di processo a ss_typical_max, voltaggio di alimentazione

a 1,08V e temperatura a 125°C. Il passo successivo è stato quello di creare lo script *synth_ depthwise.tcl* in cui sono raccolti tutti i comandi da impartire al tool di sintesi. Attraverso il comando *analyze* sono stati inseriti tutti i file in systemverilog necessari per la compilazione, è stato inserito il link alle librerie ed è stato impostato un clock delay di 0.1 per tutti gli input e gli output. La sintesi è stata effettuata con più valori di periodo di clock per osservare le variazioni nell'area.

4.2.2 Risultati

Con questi parametri abbiamo trovato i risultati in figura:

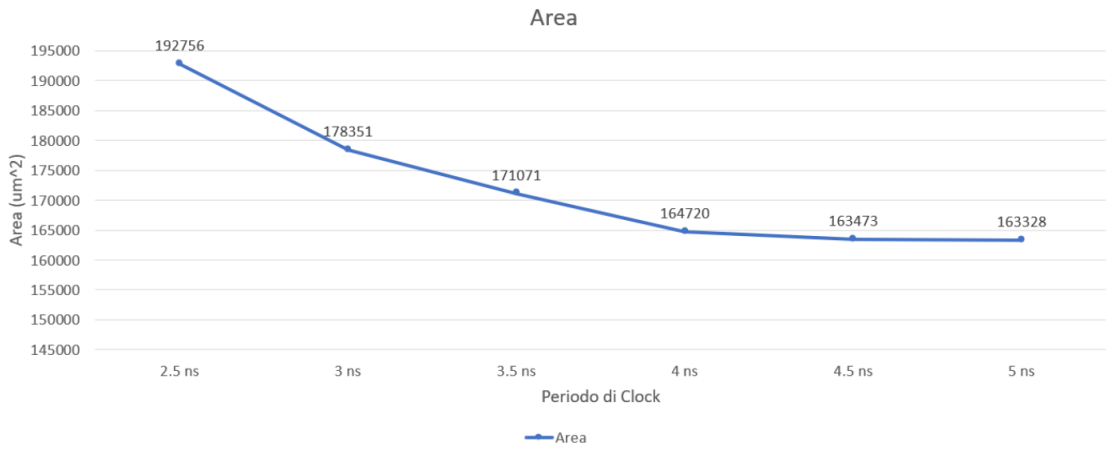


Figura 4.7: Sintesi Depthwise

Si può notare che fino ad un valore di 4 ns come periodo di clock l'area dell'acceleratore rimane pressochè costante mentre a valore di clock più bassi cresce in maniera esponenziale finchè, al periodo di clock di 2 ns, lo slack, cioè il margine di differenza tra il periodo richiesto e quello simulato, diventa violato.

Capitolo 5

Darkside

Darkside è un chip di ricerca, sviluppato nell'Energy-Efficient Embedded Systems Laboratory di Bologna in collaborazione con l'ETH di Zurigo, basato su OpenPulp, la piattaforma open source, multicore, per il calcolo ultra low power in dispositivi come nodi IoT che richiedono un processamento di streaming di dati generati da vari sensori come accelerometro, camera a bassa risoluzione, microfoni, segnali biometrici.

5.1 PULP nel dettaglio

Per capire meglio le novità del chip Darkside è necessario spiegare nel dettaglio la piattaforma PULP. PULP è costituito da un'architettura a microcontrollore avanzata, successore di PULPino, architettura che si compone di un solo core, che include un cluster di processori strettamente accoppiati fra loro a cui è possibilile fare calcolare in modo parallelo kernel di complessità elevata scaricando il carico computazionale dal processore principale. Le novità introdotte nell'architettura di PULP sono:

- Sia RI5CY sia ZERO-RI5CY come possibili core principali.
- Sistema di Input/Output autonomo.
- Nuovo sub-sistema di memoria.
- Supporto per gli HWPE.
- Nuovo controllo per interrupt.
- Nuove periferiche.
- Nuovo cluster per il calcolo in parallelo.
- Nuovo DMA di sistema.
- Nuova event unit.
- Nuova SDK.

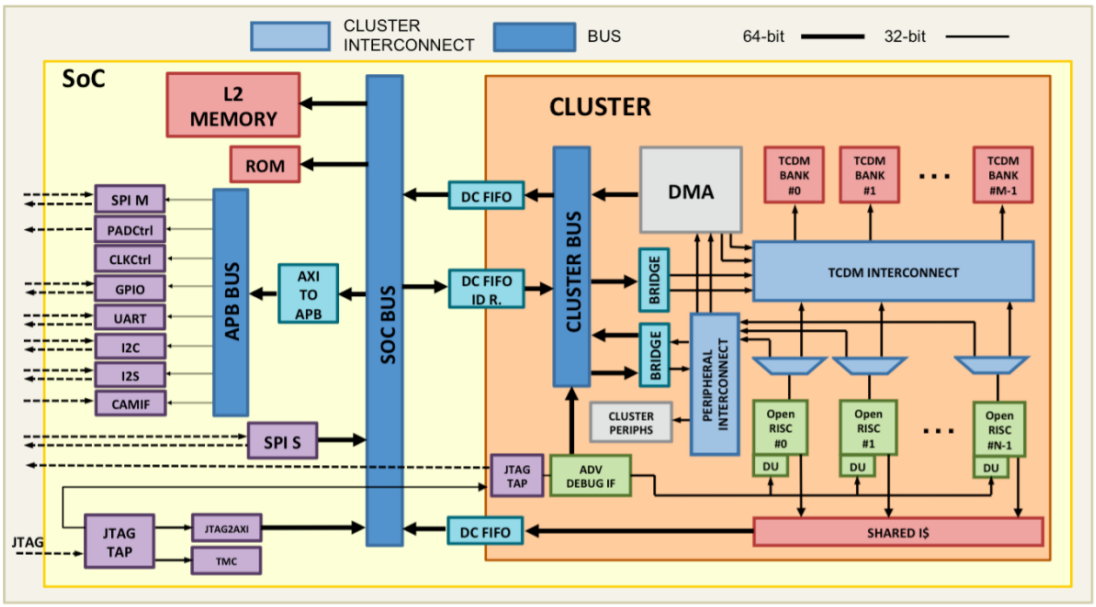


Figura 5.1: Pulp platform

RISCY è un processore in-order, single-issue con 4 stadi di pipeline ed ha un IPC (Istruction Per Cycle) vicino ad 1, supporto per le base integer instruction set (RV32I), le compressed instructions (RV32C) e l'estensione per le multiplication instruction set (RV32M). Può essere configurato per avere le istruzioni in floating point single-precision (RV32F). In più implementa diverse estensioni ISA come: loop hardware, istruzioni di load e store con il post-incremento, istruzioni per la bit-manipulation, operazioni MAC , operazioni fixed point, istruzioni SIMD e il prodotto vettoriale. Il core è stato pensato per incrementare l'efficienza energetica in applicazioni di signal-processing ultra low power. Informazioni aggiuntive si possono trovare al sito: http://ieeexplore.ieee.org/abstract/document/7864441/.

Lo ZERO-RI5CY invece è un in-order, single issue core con 2 stadi di pipeline e supporta, come il RI5CY, le RV32I, RV32C, RV32M, mentre non supporta le RV32F ma le RV32E (estensione per un numero ridotto di registri). Il core è ottimo per ambiti che hanno come target applicazioni ultra-low-power and ultra-low-area. (http://ieeexplore.ieee.org/document/8106976/).

Il nuovo efficiente subsistema di uDMA (micro-DMA) permette il controllo degli I/O che comunicano in modo autonomo con le periferiche. Per fare questo il core ha solo bisogno di programmare il uDMA e aspettare per gestire il trasferimento di dati. Informazioni aggiuntive su http://ieeexplore.ieee.org/document/8106971/

PULP supporta le seguenti interfacce di I/O:

- SPI (come master)
- I2S
- Camera Interface (CPI)
- I2C
- UART
- JTAG

PULP supporta inoltre l'integrazione degli hardware accelerators che sono stati discussi ampiamente nel capitolo 3.

5.1.1 Darkside

In particolare Darkside è il chip basato su PULP in cui è stato integrato il Depthwsie Accelerator. Darkside è il diretto successore di Dustin, chip creato nel 2020 per accelerare le performance delle Mixed-precision Quantized Neural Network nelle applicazioni IoT e consiste in un SoC accelerato da un cluster da 16 cores dotato delle SIMD (single-istruction-multiple-data)/MIMD (Multiple-istruction-multiple-data) set di istruzioni che accoppiano più dati, molto efficienti per applicazioni come le MAC. Al contrario di Dustin, Darkside integra nel cluster solamente 8 cores paralleli che condividono una memoria L1 da 128kB, in più è presente la memoria L2 nel SoC, da 112 kB. Come la maggior parte dei chip PULP, il SoC di Darkside è accompagnato da diverse periferiche come JTAG, QSPI, I2C, I2S, una Camera interface, UART e una unità RISC-V di debug basata su JTAG con accesso completo al bus della memoria del sistema.

5.1.2 Ips per velocizzare il calcolo delle reti neurali

Darkside integra al suo interno quattro nuovi acceleratori per il calcolo delle reti neurali:

- Tensor unit: modulo parametrico sviluppato per eseguire prodotti matriciali in mixed precision su operandi floating point per il training e l'inferenza di reti neurali. Il design è basato su di un array sistolico composto da delle unità Fused Multiply-Accumulate.
- Datamover: modulo per effettuare operazioni di reordering su tensori in memoria. Tra queste operazioni più essere eseguita per esempion una conversione CHW to HWC.
- L'acceleratore Depthwise presentato in questa tesi.
- RI5CV-NN: un core RI5CY con l'estensione per nuove istruzioni mixed-precision per migliorare le performance in termini di CPI (Clock Per Istruction).

5.2 Integrazione degli Ips in Darkside

Per integrare gli IPs in Darkside è stato necessario un lavoro di analisi del cluster di PULP insieme agli sviluppatori della Tensor Unit e del Datamover. In particolare nel cluster sono presenti tre moduli che definiscono gli ambienti in cui vengono gestiti gli hwpe:

- *hwpe_subsystem.sv*: modulo che istanzia le top_level entity degli acceleratori.
- *cluster_peripherals.sv*: modulo che gestisce le periferiche nel cluster, tra cui quella di slave degli acceleratori dove sono gestiti i registri di controllo.
- *cluster_interconnect_wrap.sv*: il modulo che gestisce tutte le connessioni all'interno del cluster.

Nell'*hwpe_subsystem.sv* è presente l'ambiente adatto per inserire gli hwpe, quindi sono stati inseriti le top_level dei moduli degli acceleratori. Per integrare poi questi nel cluster vengono definite due interfacce particolari:

- una di tipo *hci_core_intf.master*, dove hci deriva da Heterogeneous Cluster Interconnect, chiamata *hwpe_xbar_master* per dialogare con l'HWPE, questa interfaccia è la stessa usata per gestire i core all'interno del cluster di PULP.
- l'altra di tipo *XBAR_PERIPH_BUS.slave* chiamata *hwpe_cfg_slave* per dialogare con i registri che programmano gli hwpe.

Come scelta progettuale è stata effettuata quella di collegare il datamover al *cluster_interconnect_wrap.sv* come se fosse un core, in questo modo può lavorare parallelamente agli altri acceleratori. La Depthwise e la tensor unit invece sono gestite attraverso un mux nell'*hwpe_subsystem.sv* così definito:

```
// HCI Mux for HWPE selection
// hwpe_sel = 1'b1 -> depthwise
// hwpe_sel = 1'b0 -> tensorcore
hci_core_mux_static #(
  .NB_CHAN         ( 2                  ),
  .DW              ( N_MASTER_PORT*32   )
) i_hwpe_mux_sel   (
  .clk_i           ( clk                ),
  .rst_ni          ( rst_n              ),
  .clear_i         ( 1'b0               ),
  .sel_i           ( hwpe_sel_i         ),
  .in              ( i_slave            ),
  .out             ( hwpe_xbar_master [0] )
);
```

Listing 11: Mux per selezionare l'hwpe desiderato in *hwpe_subsystem.sv*

Per gestire il Mux è stato creato il segnale *hwpe_sel*. Questo seleziona quale acceleratore usare al momento della configurazione. Per configurare questo segnale è

stato modificato un registro, già presente all'interno della *cluster_control_unit.sv* all'indirizzo 0x18, aggiungendo un bit di configurazione in posizione 13. Se è configurato a 0 viene selezionato il tensorcore mentre ad 1 la depthwise. In questo modo basta una sola interfaccia master e una slave per gestire la depthwise e la TPU attraverso il mux, mentre è stata aggiunta una interfaccia hci *hwpe_xbar_master_datamover* per collegare direttamente il datamover con il *cluster_interconnect_wrap*. *hwpe_cfg_slave* è diventato invece un array di interfacce di dimensione 2 in cui, nella posizione 0 viene effettuato il binding con l'interfaccia o periph della depthwise o della tpu, a seconda del segnale di selezione, e nella posizione 1 con la porta periph del Datamover.

Nel *cluster_peripherals.sv* è stata modificata la porta master con interfaccia *XBAR_PERIPH_BUS* facendola diventare un array di interfacce di dimensione 2 compatibilmente ai cambiamenti fatti nell'*hwpe_subsystem*, sistemando anche gli assegnamenti interni. In più è stato aggiunto anche il segnale di output di *hwpe_sel* proveniente dalla cluster control unit.

Nel *cluster_interconnect_wrap.sv* è stata aggiunta una porta *core_tcdm_slave* in più per il datamover che viene trattato a tutti gli effetti come un core dal cluster. Infine nel cluster sono stati aggiunti tutti i segnali per connettere le nuove porte create nei moduli. Le modifiche fatte sono schematizzabili come nella figura 5.2.

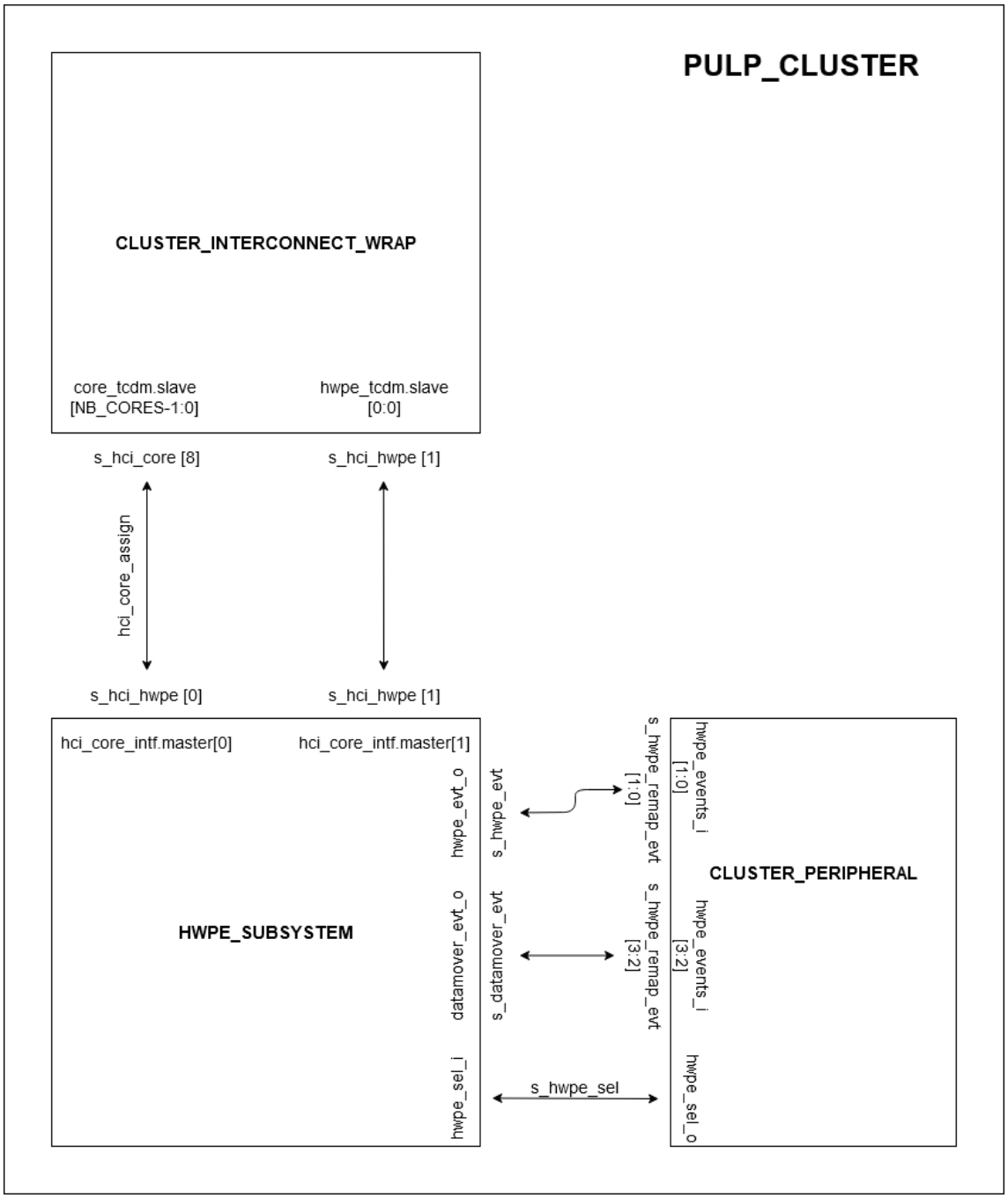


Figura 5.2: Integrazione degli hwpe

5.3 Risultati Finali

Dopo aver integrato l'acceleratore Depthwise all'interno del cluster di Darkside è stato testato per verificarne il corretto funzionamento. Il test è simile a quello fatto nell'hwpe-tb con la differenza che, mentre nel primo venivano istanziati solo i componenti fondamentali per poter eseguire un test sugli hwpe, in questo caso è da simulare l'intero chip. Per questo sono stati presi in considerazione alcuni accorgimenti che lo differenziano dal test stand alone tra cui:

1. Lo spazio degli indirizzi è diverso quindi sono stati inseriti parametri in più in *hal_depthwise.h* per gestire l'indirizzo base dei registri della depthwsise e l'indirizzo della cluster control unit per cambiare il segnale di *hwpe_sel*.

```
// global address map + event IDs
#define DEPTHWISE_ADDR_BASE      0x00201000
#define CLUS_CTRL_ADDR_BASE      0x00200000
#define DEPTHWISE_EVT0           14
#define DEPTHWISE_EVT1           15

// cluster controller register offset and bits
#define CLUS_CTRL_DEPTHWISE_OFFS              0x18
#define CLUS_CTRL_DEPTHWISE_CG_EN_MASK        0x2800 //select hwpe_en=1 & hwpe_sel=1
#define CLUS_CTRL_DEPTHWISE_HCI_PRIO_MASK     0x100
#define CLUS_CTRL_DEPTHWISE_HCI_MAXSTALL_MASK 0xff
```

Listing 12: Indirizzi aggiunti in *hal_depthwise.h*

2. Per scrivere nella cluster control unit il bit di *hwpe_sel* e per avere la priorità hci per la depthwise sono state inseriti due comandi:

```
#define DEPTHWISE_CG_ENABLE()  *(volatile int*) (CLUS_CTRL_ADDR_BASE +
CLUS_CTRL_DEPTHWISE_OFFS) |=  CLUS_CTRL_DEPTHWISE_CG_EN_MASK
#define DEPTHWISE_CG_DISABLE() *(volatile int*) (CLUS_CTRL_ADDR_BASE +
CLUS_CTRL_DEPTHWISE_OFFS) &= ~CLUS_CTRL_DEPTHWISE_CG_EN_MASK

#define DEPTHWISE_SETPRIORITY_CORE() *(volatile int*) (CLUS_CTRL_ADDR_BASE +
CLUS_CTRL_DEPTHWISE_OFFS) &= ~CLUS_CTRL_DEPTHWISE_HCI_PRIO_MASK
#define DEPTHWISE_SETPRIORITY_DEPTHWISE() *(volatile int*) (CLUS_CTRL_ADDR_BASE +
CLUS_CTRL_DEPTHWISE_OFFS) |=  CLUS_CTRL_DEPTHWISE_HCI_PRIO_MASK
```

Listing 13: Comandi aggiunti in *hal_depthwise.h*

   Questi comandi scritti sopra vengono poi chiamati in *test_depthwise.c* per inizializzare il test.

3. Per controllare la validità dei dati in output è stata inserita una funzione che confronta i risultati real-time con valori di output generati da un golden model.

5.3.1 Potenza

Attraverso la sintesi del cluster è stato possibile analizzare il consumo in potenza dell'acceleratore durante il test che è stato creato. Il chip di Darkside è stato sintetizzato con un periodo di clock di 5ns (200 MHz) ad una alimentazione di 1.2V. Con questi valori di sintesi è stata creata una netlist del cluster con cui è possibile effettuare una simulazione post layout del test creato della Depthwise.

La simulazione post layout è stata effettuata alla frequenza di 50 MHz e i valori di potenza dinamica ottenuti sono stati poi scalati alla frequenza di 200 MHz. Con ModelSim è stato creato il file VCD contenente tutte le informazioni sulle switching activity nel periodo in cui avviene il test della Depthwise. In seguito attraverso il comando *start_power_zh.csh* è stato possibile analizzare il consumo effettivo del cluster durante il test della Depthwise.

Il test è stato eseguito con una immagine di 25x20 pixels di grandezza e 24 canali di profondità. I risultati sono:

- Consumo di potenza del cluster: 59.4 mW.
- Consumo di potenza dell'acceleratore Depthwise: 7.1 mW.

5.3.2 Efficienza energetica

Calcolando il numero di MAC totali per terminare l'operazione possiamo effettuare una stima dell'efficienza energetica dell'acceleratore. Considerando:

- Per ogni pixel in output vengono effettuate 9 MAC, il numero totale sarà 3x3x23x18x24 = 89424.
- L'energia necessaria per il test è da considerare come la potenza per la durata del test, quindi $59.4mW \cdot 29715ns = 1.77uJ$.

Otteniamo :

$$\frac{MACs}{energy} = \frac{89424MAC}{1.77mJ} = 50522.03MAC/uJ = 50.5GigaMac/J \tag{5.1}$$

Considerando invece che una MAC è composta da due operazioni (una moltiplicazione e una somma) otteniamo:

$$50.5GigaMAC/J \cdot 2 = 101GOP/J. \tag{5.2}$$

5.3.3 Speed-up

Un altro risultato importante è rappresentato dallo speed-up ottenuto dall'acceleratore rispetto al calcolo ottenuto dalla stessa convoluzione Depthwise usando solamente il cluster. Infatti usando le librerie pulp-nn sulla repository di pulp_platform su GitHub è stato possibile eseguire un test di una convoluzione depthwise all'interno del cluster eseguita su 8 core in parallelo. I cicli di clock sono stati ottenuti inserendo nei due test la funzione *pi_perf_start()*:

- Numero di cicli usando il cluster : 23752.
- Numero di cicli usando l'acceleratore Depthwise: 5943.

Come è visibile da questi risultati l'acceleratore Depthwise termina la sua computazione circa 4 volte più velocemente rispetto alla soluzione software. E' da considerare anche il fatto che utilizzando l'acceleratore Depthwise tutto il calcolo sarebbe concentrato su esso lasciando i core liberi di lavorare su altre istruzioni in modo parallelo. Questi risultati potrebbero anche migliorare considerando immagini più grandi perchè sarebbe maggiore il tempo impiegato durante il calcolo degli output rispetto al tempo in cui il window_buffer carica i nuovi valori al suo interno al completamento della colonna.

Con questo valori è possibile calcolare anche il numero di operazioni eseguibili al secondo infatti, alla frequenza di 200MHz, quindi con un periodo di clock di 5ns, otteniamo:

$$\frac{MAC}{cycles} = \frac{89424}{5943} = 15.01 MAC/cycle. \tag{5.3}$$

$$\frac{MAC}{cycles \cdot T_{clk}} = 3.01 GMAC/s = 6.02 GOP/s. \tag{5.4}$$

Capitolo 6

Conclusioni

Le CNN sono sempre più comuni nelle moderne applicazioni di Deep Neural Network e quando si parla di sistemi come dispositivi IoT o device indossabili queste devono soddisfare limiti molto stringenti in termini di consumo di potenza e area. In questa tesi è stato presentato un acceleratore hardware pensato per le convoluzioni Depthwise, la prima parte delle moderne separable convolution, che introducono una notevole ottimizzazione rispetto alle convoluzioni tradizionali in termini di operazioni da compiere. Per sviluppare l'acceleratore è stata usata la piattaforma PULP e l'ambiente di sviluppo HWPE già presente al suo interno. Grazie ad una architettura HWC dei dati in memoria e un window buffer in grado di elaborare tensori di grandezza 3x3 pixels da 8 bits per un massimo di 16 canali è stato possibile avere un buon riuso dei dati che porta l'acceleratore ad avere uno throughput massimo di 4 pixel per ciclo di clock. Questo acceleratore è stato poi integrato all'interno del cluster di Darkside, il nuovo chip sviluppato dall' Energy-Efficient Embedded Systems Laboratory dell'università di Bologna in collaborazione con l'ETH di Zurigo, ed è stato possibile misurare le performance in termini di consumo di potenza e velocità computazionale. Con un consumo nell'ordine dei mW e una efficienza energetica di 101GOP/J soddisfa i requisiti di potenza necessari per essere utilizzato in ambito IoT. Inoltre con le performance di 6 GOPs/s rappresenta una alternativa 4 volte più veloce rispetto al cluster con 8 core in parallelo. In questo lavoro è stata sviluppata la parte della separable convolution relativa alle Depthwise, in un futuro lavoro potrebbe essere ampliato inserendo anche la logica per eseguire la Pointwise Convolution che potrebbe sfruttare, attraverso una pipeline, i risultati in output già ordinati per canale. Un altro possibile sviluppo per migliorare l'acceleratore potrebbe essere l'aggiunta di uno stride programmabile, che possa assecondare anche richieste di padding.

Bibliografia

[1] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[2] Sungho Park, Ahmed Al Maashri, Kevin M. Irick, Aarti Chandrashekhar, Matthew Cotter, Nandhini Chandramoorthy, Michael Debole, and Vijaykrishnan Narayanan. System-on-chip for biologically inspired vision applications. *IPSJ Transactions on System LSI Design Methodology*, 5:71–95, 2012.

[3] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. pages 109–116, June 2011.

[4] Vinayak Gokhale, Jonghoon Jin, Aysegul Dundar, Berin Martini, and Eugenio Culurciello. A 240 g-ops/s mobile coprocessor for deep neural networks. pages 696–701, 2014.

[5] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A. Horowitz. Convolution engine: Balancing efficiency flexibility in specialized computing. 41(3):24–35, June 2013.

[6] David Moloney. 1tops/w software programmable media processor. pages 1–24, 2011.

[7] Francesco Conti and Luca Benini. A ultra-low-energy convolution engine for fast brain-inspired vision in multicore clusters. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 683–688, 2015.

[8] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[9] Lin Bai, Yiming Zhao, and Xinming Huang. A cnn accelerator on fpga using depthwise separable convolution. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 65(10):1415–1419, 2018.

[10] Wei Ding, Zeyu Huang, Zunkai Huang, Li Tian, Hui Wang, and Songlin Feng. Designing efficient accelerator of depthwise separable convolutional neural network on fpga. 97:278–286, 2019.

[11] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. pages 1–6, 2017.

[12] Stan Kurtz. Hypercompact hii regions. *Proceedings of the International Astronomical Union*, 1(S227):111–119, 2005.