# POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Elettronica


Tesi di Laurea Magistrale

# Tiny Machine Learning: dalla valutazione delle performance di dispositivi esistenti allo sviluppo di acceleratori ad hoc su FPGA

**Relatori**
Prof. Mario Roberto Casu
Prof. Massimo Ruo Roch

**Candidato**
Gorgerino Giacomo

Dicembre 2024

*Per prima cosa vorrei ringraziare i miei relatori per la loro disponibilità e per la fiducia che hanno riposto in me durante il lavoro svolto per questa tesi.*
*Un ringraziamento speciale alla mia ragazza Matilde, per avermi supportato e sopportato in questi ultimi due anni e mezzo.*
*Grazie ai miei genitori, per avermi sempre sostenuto e per aver creduto in me.*
*Infine non posso non ringraziare il team DIANA, probabilmente non sarei riuscito a terminare la magistrale senza di voi.*

# Abstract

L'edge computing sta rivoluzionando il modo in cui approcciamo il machine learning, portando l'intelligenza artificiale più vicino alla sorgente dei dati. Questo spostamento verso la periferia della rete offre numerosi vantaggi, tra cui una drastica riduzione della latenza, consentendo risposte in tempo reale cruciali per applicazioni come la guida autonoma o i sistemi di controllo industriale. Inoltre, l'edge computing alleggerisce il carico sui data center centralizzati, ottimizzando l'utilizzo delle risorse e migliorando la scalabilità. Un altro aspetto fondamentale è la maggiore privacy e sicurezza, poiché l'elaborazione avviene localmente, riducendo il rischio di violazioni durante la trasmissione. Infine, permette di operare in ambienti con connessioni limitate o instabili, rendendo possibile l'implementazione di soluzioni di intelligenza artificiale anche in aree remote.

Questo nuovo approccio presenta però una serie di sfide e limitazioni che ne condizionano l'adozione su larga scala. Il principale ostacolo è rappresentato dalle limitazioni sulla potenza di calcolo e la capacità di memoria dei dispositivi edge. Questo limite rende difficile l'esecuzione di modelli di machine learning complessi e richiede che i modelli vengano scalati ed ottimizzati, spesso a discapito dell'acuratezza dei risultati. Un'altra sfida è costituita dalla gestione: mantenere aggiornato il firmware di un numero elevato di dispositivi distribuiti potenzialmente su vaste aree geografiche richiede una pianificazione accurata e la realizzazione di strumenti specializzati.

Questa tesi si concentra sull'analisi delle prestazioni di diverse piattaforme hardware per l'edge computing nell'esecuzione di modelli di machine learning. Attraverso l'utilizzo del benchmark MLPerf-Tiny verrà condotta una valutazione comparativa della latenza di inferenza, del consumo energetico e delle prestazioni computazionali di microcontrollori, librerie firmware e acceleratori hardware. L'obiettivo è quantificare l'impatto delle limitazioni hardware sulle metriche di performance.
Per conseguire un ulteriore incremento delle prestazioni è necessario utilizzare acceleratori hardware specificamente progettati e ottimizzati per l'esecuzione dei modelli richesti. La tesi propone una metodologia per la progettazione di tali acceleratori su FPGA, prendendo in considerazione le specifiche di una rete neurale 1D e i vincoli imposti dall'architettura del dispositivo target. Tale metodologia rappresenta un primo passo verso lo sviluppo di acceleratori hardware più avanzati, in grado di ottimizzare l'inferenza di modelli di machine learning su dispositivi edge con risorse limitate.

# Indice

# Elenco delle tabelle

# Elenco delle figure

# Capitolo 1

# Introduzione

## 1.1 Machine Learning

Il machine learning rappresenta un pilastro fondamentale dell'intelligenza artificiale, consentendo ai sistemi informatici di evolvere e adattarsi autonomamente attraverso l'esperienza. A differenza dei sistemi tradizionali, che si basano su regole e istruzioni predefinite, gli algoritmi di machine learning sono in grado di apprendere dai dati, migliorando le proprie prestazioni nel tempo senza la necessità di una programmazione esplicita.
Attraverso algoritmi matematici e statistici, i modelli di machine learning analizzano grandi quantità di dati per estrarre informazioni significative, costruire previsioni e classificazioni, e migliorare le proprie prestazioni nel tempo. Le applicazioni del machine learning sono vaste e spaziano dalla visione artificiale alla elaborazione del linguaggio naturale, passando per la raccomandazione di prodotti, la diagnosi medica e la previsione di eventi futuri.

## 1.2 Cloud,Fog e Edge Computing

Il cloud computing e l'edge computing rappresentano approcci diversi per l'elaborazione dei dati. Il primo prevede che i dati acquisiti siano inviati, tramite una connessione a internet, a datacenter centralizzati, dove vengono memorizzati ed elaborati. Questo approccio offre una grande scalabilità e flessibilità, oltre che una maggiore facilità di gestione. Il punto debole è rappresentato dalla latenza ovvero il tempo che intercorre tra l'invio dei dati e la ricezione dei risultati. In alcune applicazioni e in alcune situazioni, come ad esempio la guida autonoma o l'elaborazione di dati in zone remote del pianeta, lo svantaggio introdotto dalla latenza può non essere trascurabile. Al contrario, l'edge computing prevede che l'elaborazione dei dati avvenga il più vicino possibile alla sorgente, ovvero ai dispositivi che li acquisiscono. Questo approccio consente di diminuire drasticamente la latenza permettendo risposte in tempo reale. Altri vantaggi introdotti dall'edge computing sono la maggiore privacy e sicurezza, poiché i dati non devono essere inviati in remoto, e la possibilità

di operare in ambienti con connessioni limitate o instabili[1].
Esiste una terza tipologia di soluzioni che si colloca a metà tra il cloud computing e l'edge computing chiamata fog computing. Questo approccio prevede che i dati vengano elaborati in centri, chiamati *fog nodes*, che si trovano tra i dispositivi edge e i datacenter centralizzati. Questo approccio permette di mantenere la flessibilità e la scalabilità del cloud computing, riducendo la latenza grazie alla vicinanza ai dispositivi edge.

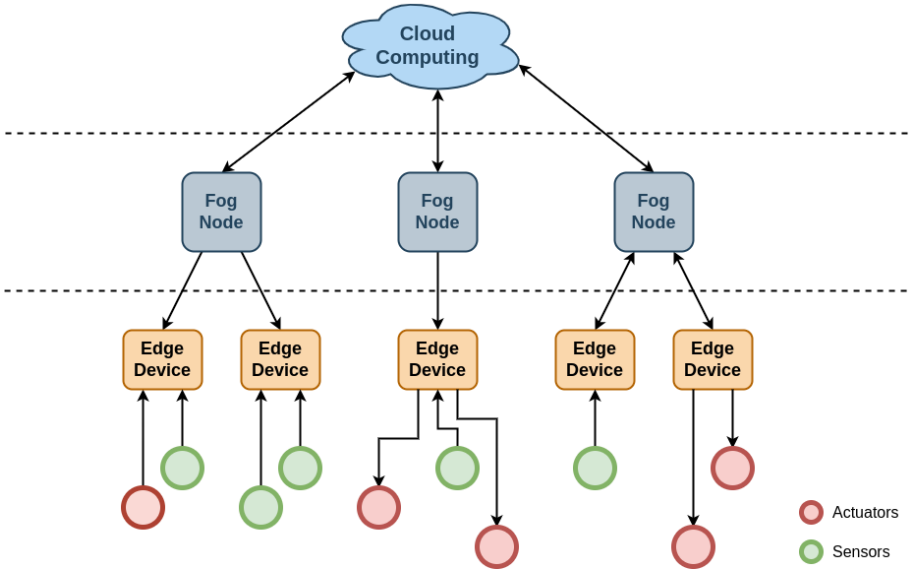



Figura 1.1: Cloud, Edge e Fog computing

Con l'aumentare dei dispositivi connessi alla rete, l'utilizzo del solo cloud computing per l'elaborazione dei dati potrebbe non essere più sostenibile. Anche se l'evoluzione delle tecnologie utilizzate per le telecomunicazioni sono in rapida crescita (si pensi ai vantaggi introdotti in termini di banda e latenza dal passaggio dal 4G al 5G), il numero sempre maggiore di dispositivi e il conseguente aumento dei volumi di dati generati potrebbe saturare le infrastrutture esistenti.

L'elaborazione dei dati all'interno dello stesso dispositivo che si occupa di acquisirli o in alternativa in un dispositivo posizionato nelle strette vicinanze sembra rappresentare una soluzione più sostenibile e scalabile.

Questa transizione verso l'edge computing risulta essere rallentata da una serie di sfide e limitazioni che ne condizionano l'adozione su larga scala[2].
Considerando che i principali dispositivi edge sono costituiti da microcontrollori, il principale ostacolo è rappresentato dalle limitazioni sulla potenza di calcolo e la capacità di memoria dei dispositivi. Per un microcontrollore è possibile stimare una quantità di RAM disponibile tra le centinaia di kilobyte e pochi megabyte e una quantità di memoria flash on-chip tra i pochi megabyte e qualche decina di megabyte. Questo rende impossibile l'esecuzione di modelli di machine learning di grandi dimensioni e richiede in generale che i modelli vengano scalati ed ottimizzati,

spesso a discapito dell'acuratezza dei risultati. Un'ulteriore ostacolo causato dai dispositivi utilizzati è la ridotta potenza di calcolo dei microcontrollori, soprattutto per quanto riguarda le operazioni in virgola mobile.

Anche se questi limiti si stanno riducendo grazie all'evoluzione tecnologica che porta ad avere dispositivi sempre più potenti, con una maggiore quantità di memoria o con periferiche dedicata ai calcoli in virgola mobile, esistono altri ostacoli causati prevalentemente dall'idea stessa di edge computing. Avere un numero elevato di dispositivi distribuiti su aree vaste o remote e, potenzialmente, senza una connessione di rete, rende difficile la gestione e la manutenzione degli stessi o eventuali aggiornamenti del firmware.

Mentre l'esecuzione di algoritmi di inferenza su dispositivi edge, anche se richiede una serie di compromessi, è una pratica sempre più comune, la fase di allenamento rimane ancora una pratica quasi escusiva di elaboratori più potenti, come i server in cloud. Il tempo richiesto per addestrare un modello di machine learning è spesso molto lungo e richiede una grande quantità di risorse computazionali, rendendo difficile o impossibile l'addestramento di modelli complessi su dispositivi edge.

## 1.3 Acceleratori hardware

Per superare questi ostacoli, sono stati sviluppati degli acceleratori hardware specificamente progettati per l'inferenza di modelli di intelligenza artificiale in ambienti embedded.

Questi acceleratori, ispirati alle soluzioni impiegate nei datacenter ma con dimensioni e consumi energetici ridotti, offrono notevoli vantaggi rispetto ai microcontrollori tradizionali:

- **Maggiore potenza di calcolo:** gli acceleratori hardware sono in grado di elaborare i dati e eseguire le operazioni matematiche richieste dai modelli di machine learning in modo significativamente più veloce rispetto ai microcontrollori.

- **Maggiore efficienza energetica:** grazie ad architetture ottimizzate per l'elaborazione di algoritmi di intelligenza artificiale, gli acceleratori consumano meno energia rispetto ai microcontrollori, un aspetto fondamentale per dispositivi alimentati a batteria.

- **Esecuzione di modelli più complessi:** la maggiore potenza di calcolo consente di eseguire modelli di machine learning più grandi e complessi, con un conseguente aumento dell'accuratezza e delle prestazioni.

- **Delega delle operazioni di machine learning:** delegando l'esecuzione del modello all'acceleratore, il microcontrollore può dedicare le proprie risorse ad altre attività, come la gestione delle periferiche, la comunicazione con altri dispositivi o l'esecuzione di altri task.

Esistono due principali approcci per l'implementazione di acceleratori hardware per l'inferenza di modelli di machine learning: il primo prevede l'utilizzo di chip dedicati. Questi dispositivi, come ad esempio il Google Edge TPU, sono progettati per l'esecuzione di modelli di machine learning e offrono prestazioni elevate e un'efficienza energetica ottimale. Il secondo approccio prevede l'utilizzo di FPGA, dispositivi programmabili che consentono di implementare acceleratori personalizzati e ottimizzati per specifici modelli di machine learning. Questa soluzione offre una maggiore flessibilità e la possibilità di adattare l'acceleratore alle esigenze specifiche dell'applicazione.

## 1.4 Obiettivi

Il primo obiettivo di questa tesi è la valutazione delle prestazioni dei dispositivi edge nell'esecuzione di algoritmi di machine learning. La relazione tra la latenza di esecuzione, potenza di calcolo e consumo energetico verrà valutata attraverso l'utilizzo di un software di benchmarking standardizzato, MLPerf-Tiny[3].
I risultati ottenuti da questi test saranno poi confrontati con quelli ottenuti utilizzando delle tecniche di ottimizzazione, come l'utilizzo di librerie ottimizzate o l'utilizzo di acceleratori hardware.
Inoltre, verrà proposto un metodo per calcolare i parametri necessari per l'implementazione di un acceleratore hardware su FPGA a partire dalle specifiche di una rete neurale 1D e delle caratteristiche del dispositivo target.

## 1.5 Struttura della tesi

Il lavoro di tesi è stato strutturato come segue:

- Nel capitolo 2 verranno presentati i concetti di base relativi a TensorFlow Lite Micro e al benchmark MLPerf-Tiny.
- Nel capitolo 3 saranno descritti i dettagli implementativi sulle diverse piattaforme utilizzate, oltre ad una prima analisi dei risultati.
- Nel capitolo 4 verranno confrontati i risultati ottenuti e verrà valutata l'efficacia delle diverse soluzioni proposte.
- Nel capitolo 5 sarà proposto un metodo per derivare i parametri necessari per l'implementazione di un acceleratore hardware su FPGA.

# Capitolo 2

# LiteRT Micro and MLPerf-Tiny Benchmark

## 2.1 Tensorflow

Tensorflow[4] è una libreria open-source, sviluppata da Google, che offre un'ampia gamma di strumenti per la creazione e l'addestramento e il deployment di grafi computazionali. Questi grafi, rappresentati come *"dataflow graphs"*, descrivono le operazioni matematiche necessarie per eseguire algoritmi di machine learning, in particolare le reti neurali.
Una delle caratteristiche distintive di TensorFlow è la sua flessibilità, che permette di utilizzare la libreria per creare una vasta gamma di modelli di machine learning, dai più semplici modelli di regressione lineare ai più complessi modelli di deep learning. La scalabilità di TensorFlow è garantita dalla sua capacità di eseguire calcoli su una vasta gamma di hardware, dalle CPU alle GPU, fino a cluster di macchine interconnesse. Inoltre, TensorFlow offre strumenti per la distribuzione dei modelli su dispositivi mobili e embedded, rendendo possibile l'implementazione di soluzioni di intelligenza artificiale anche in contesti con risorse limitate.

## 2.2 LiteRT for Microcontrollers

TensorFlow Lite for Microcontrollers, ora LiteRT for Microcontrollers, rappresenta una soluzione efficace per l'implementazione di modelli di machine learning su dispositivi embedded a bassa potenza. Questo framework, compatibile con una vasta gamma di microcontrollori e microprocessori, consente di superare le sfide tecniche legate all'esecuzione di algoritmi di intelligenza artificiale in ambienti con risorse limitate.

LiteRT for Microcontrollers si propone di risolvere alcuni dei problemi tecnici legati all'implementazione di modelli di machine learning su microcontrollori. In assenza di un framework unificato, gli sviluppatori sarebbero costretti a creare firmware specifici e ottimizzati per ciascuna piattaforma hardware. Questo processo, oltre ad essere complesso e dispendioso in termini di tempo e risorse, genera

problematiche legate alla gestione di diverse versioni del codice[5]. Inoltre, un altro problema potrebbe essere rappresentato dalle diverse performance ottenute su piattaforme diverse, rendendo difficile la scelta della piattaforma migliore per un determinato progetto.

LiteRT for Microcontrollers semplifica notevolmente questo processo offrendo un'interfaccia unificata per l'esecuzione di modelli di machine learning. Ciò consente di utilizzare lo stesso modello su diverse piattaforme hardware senza la necessità di modificare il codice sorgente, garantendo una maggiore portabilità e flessibilità nello sviluppo.

Per utilizzare questa libreria, è necessario generarne una versione specifica per la piattaforma target. Questo processo viene effettuato tramite un tool disponibile nella repository ufficiale di LiteRT for Microcontrollers[6]. Successivamente, il modello di machine learning deve essere esportato in un formato compatibile con LiteRT Micro, ovvero un file *.tflite*.

A partire da un modello LiteRT esportato in formato *.tflite*, il framework genera un file *.cc*.

Questo file, chiamato *Flatbuffer*, contiene tutte le informazioni necessarie per l'esecuzione del modello, come pesi, bias e la sequenza ordinata delle operazioni matematiche da eseguire. Questa metodo di rappresentazione del modello, ereditato da LiteRT, permette di ridurre la quantità di memoria necessaria ma richiede che il compilatore utilizzato per la compilazione del codice sorgente supporti C++11.

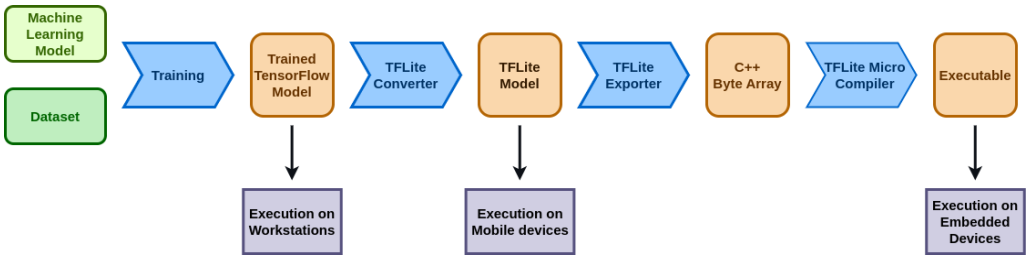



Figura 2.1: Flusso di lavoro per l'esecuzione di un modello di machine learning su un microcontrollore utilizzando LiteRT for Microcontrollers

L'esecuzione di un modello di machine learning su un microcontrollore tramite LiteRT for Microcontrollers coinvolge una serie di passaggi cruciali, orchestrati per ottimizzare l'utilizzo delle risorse limitate di questi dispositivi.
A livello di codice, possiamo identificare quattro fasi principali:

- **Decodifica delle operazioni**: Questa fase si concentra sull'interpretazione del modello di machine learning. Un oggetto della classe MicroOpResolver analizza il modello, identificando le operazioni matematiche necessarie per l'inferenza e associando ciscuna di esse ad istruzioni eseguibili dal microcontrollore.

**Esempio di decodifica delle operazioni**

```
static tflite::MicroMutableOpResolver<6> resolver;
resolver.AddFullyConnected();

resolver.AddConv2D();
resolver.AddDepthwiseConv2D();
resolver.AddReshape();
resolver.AddSoftmax();
resolver.AddAveragePool2D();
```

In questo esempio, il resolver è stato configurato per supportare sei operazioni matematiche: FullyConnected, Conv2D, DepthwiseConv2D, Reshape, Softmax e AveragePool2D.

- **Inizializzazione della memoria**: La gestione della memoria è un aspetto critico nei microcontrollori, dove le risorse sono limitate. Per evitare l'allocazione dinamica della memoria, che può portare a frammentazione e instabilità, LiteRT for Microcontrollers utilizza un approccio di allocazione statica di celle di memoria contigue. La classe MicroInterpreter si occupa di allocare la memoria necessaria per memorizzare i risultati intermedi e le variabili temporanee prima dell'esecuzione del modello.

**Esempio di inizializzazione della memoria**

```
static tflite::MicroMutableOpResolver<6> resolver;
tflite::MicroInterpreter interpreter;
alignas(8) const unsigned char model[] = {... }

constexpr int kTensorArenaSize = 100 * 1024;
uint8_t tensor_arena[kTensorArenaSize];

interpreter = tflite::MicroInterpreter(model, resolver, tensor_arena,
                                       kTensorArenaSize);
interpreter->AllocateTensors();
```

In questo esempio, la memoria necessaria per l'esecuzione del modello è allocata staticamente in un array di dimensione *kTensorArenaSize*. Il modello è caricato in memoria e l'interprete è inizializzato con il modello, il resolver e l'array di memoria.

- **Caricamento degli input**: Prima di eseguire il modello, è necessario caricare i dati di input. LiteRT for Microcontrollers utilizza un array di tensori per memorizzare i dati di input e i risultati intermedi. La classe MicroInterpreter si occupa di caricare i dati di input nei tensori corrispondenti, garantendo che i dati siano memorizzati in modo corretto e accessibili durante l'esecuzione del modello.

**Esempio di caricamento degli input**

```
int8_t inputs[...];
TfLiteTensor* input = interpreter->input(0);

for (int i = 0; i < input->bytes / sizeof(inputT); i++){
    input->data.int8[i] = inputs[i];
}
```

É inoltre necessario applicare eventuali trasformazioni ai dati di input, come la normalizzazione o la conversione in un formato compatibile con il modello.

- **Esecuzione delle operazioni**: É la fase di effettiva esecuzione del modello. La classe MicroInterpreter, utilizzando le implementazioni associate a ciascuna operazione nella fase di decodifica, esegue sequenzialmente le operazioni matematiche definite nel modello. L'interprete gestisce il flusso di dati tra le diverse operazioni, garantendo che i risultati intermedi vengano memorizzati correttamente e utilizzati per i calcoli successivi. Questa fase rappresenta il cuore dell'inferenza, dove il modello elabora i dati di input per generare i risultati finali.

**Esempio di esecuzione delle operazioni**

```
interpreter->Invoke();
```

- **Recupero dei risultati**: Una volta completata l'inferenza, i risultati saranno memorizzati nei tensori di output. La classe MicroInterpreter fornisce metodi per accedere ai tensori di output e recuperare i risultati dell'inferenza.

**Esempio di caricamento degli input**

```
TfLiteTensor* output = interpreter->output(0);

for (int i = 0; i < interpreter->outputs_size(); i++){
    // Elaborazione dei risultati
    ...
}
```

## 2.3 MLPerf-Tiny benchmark

Per valutare le prestazioni dei dispositivi edge è stato scelto di utilizzare il benchmark MLPerf-Tiny[3].
MLPerf Tiny è una suite di benchmark specificamente progettata per quantificare le prestazioni di dispositivi embedded a bassa potenza, quali microcontrollori, DSP e acceleratori hardware, nell'esecuzione di inferenze su modelli di machine learning di piccole dimensioni.

Il benchmark si propone come una valida alternativa a più complesse suite di benchmark come MLPerf, MLMark® o CoreMark®, i quali sono orientati a dispositivi di dimensioni e complessità superiori.

Per quantificare concretamente le prestazioni dei dispositivi, il benchmark utilizza i dati raccolti durante la fase di inferenza di un modello, escudendo dalla valutazione i tempi di caricamento del modello e di inizializzazione delle variabili. Questo approccio permette di ottenere risultati più realistici e più vicini ad un utilizzo reale del dispositivo. Considerando infatti tutta la fase di inferenza e non solo l'esecuzione di un singolo layer, vengono inclusi nella valutazione anche gli effetti dovuti alla banda delle memorie, alla cache e all'ottimizzazione delle operazioni matematiche.

### 2.3.1 Modelli di riferimento

La suite è composta da da 4 modelli di riferimento basati su reti neurali convoluzionali (CNN), figura 2.2, e reti neurali completamente connesse (FC), figura 2.3, selezionati per rappresentare un ampio spettro di applicazioni tipiche dell'Internet of Things (IoT), come il riconoscimento vocale, la classificazione di immagini e la detection di anomalie.

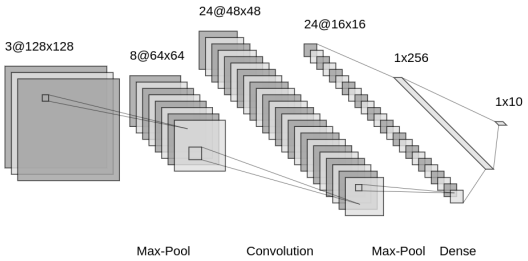



Figura 2.2: Esempio di struttura di una rete neurale convoluzionale

In modo da garantire un'ampia compatibilità con i dispositivi target, i modelli sono stati pre-allenati e quantizzati a 8 bit. Per ogni modello è stato definito un parametro di accuratezza, di cui viene riportato il valore ottenuto durante la fase di test.
I modelli selezionati per il benchmark sono i seguenti:

- **Anomaly Detection**
  Con anomaly detection si intende l'identificazione di pattern anomali in un

segnale audio. La rete neurale sono supervisionata utilizzata in questo benchmark è di grande utilità in applicazioni industriali, dove può essere utilizzata per rilevare tempestivamente guasti in un sistema meccanico. Il modello è composto da un autoencoder che elabora uno spettrogramma a 128 bande di un segnale audio. Il parametro di accuratezza utilizzato per valutare le prestazioni del modello è l'Area sotto la curva ROC (AUC). L'accuratezza ottenuta con una quantizzazione a 8 bit è di 0,86.

- **Image Classification**
  Il modello di image classification è una CNN progettata per classificare immagini a colori di dimensioni 32x32 pixel. Il dataset utilizzato per il training è un sottoinsieme del dataset CIFAR-10, contenente 60000 immagini a colori appartenenti a 10 categorie diverse.

  Per testare le prestazioni del modello vengono utilizzate 200 immagini e il parametro di accuratezza utilizzato è la *Top-1 accuracy*[1]. Per quanto riguarda il modello quantizzato a 8 bit, l'accuratezza ottenuta è del 0.85.

- **Keyword Spotting**
  KeyWord Spotting è una tecnica utilizzata per riconoscere parole chiave in un segnale audio. Questa tecnica è molto utilizzata per rendere possibile l'interazione vocale con dispositivi IoT, come ad esempio gli assistenti vocali. Come requisito fondamentale è necessario che l'implementazione abbia un consumo energetico molto basso, in modo da poter essere eseguito in background su dispositivi a batteria.

  Il modello utilizzato per il benchmark è una depthwise-separable CNN ed è stata allenata utilizzando il database Speech Commands v2, contenente 105829 campioni audio appartenenti a 30 categorie diverse. Per valutare le prestazioni del modello viene utilizzato il parametro di accuratezza *Top-1 accuracy* e vengono utilizzati 1000 campioni audio. L'accuratezza ottenuta con una quantizzazione a 8 bit è del 0.90.

- **Visual Wake Word**
  Visual Wake Word è una tecnica utilizzata per rilevare la presenza di una figura umana in un immagine. Questa tecnica è molto utilizzata in applicazioni di sorveglianza e sicurezza.

  Il modello utilizzato per il benchmark è una CNN progettata per classificare immagini a colori di dimensioni 96x96 pixel. Il dataset utilizzato per il training è un sottoinsieme del dataset MSCOCO 2014.

  Anche in questo caso il parametro di accuratezza utilizzato per valutare le prestazioni del modello è la *Top-1 accuracy* e l'accuratezza ottenuta con una quantizzazione a 8 bit è del 0.80.

[1] La *Top-1 accuracy* è una metrica di valutazione per modelli di classificazione che misura la frequenza con cui il modello assegna alla classe corretta la probailità più alta.

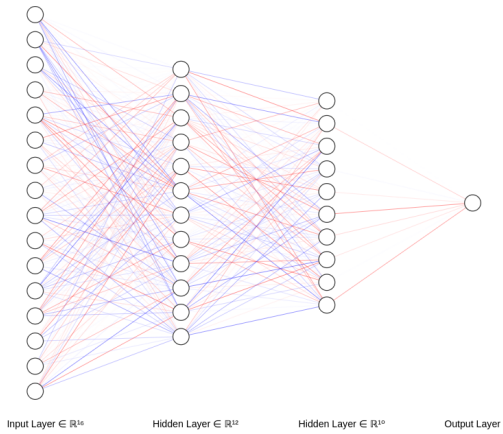



Figura 2.3: Esempio di struttura di una rete neurale completamente connessa

Nella tabella seguente sono riportate le specifiche dei modelli utilizzati per il benchmark.

| **Model** | **Input** | **Output** | **Size [kB]** | **MACs** |
|---|---|---|---|---|
| Anomaly Detection | 1x640 | 1x640 | 277.0 | 264.2k |
| Image Classification | 32x32x3 | 1x10 | 98.5 | 12.5M |
| KeyWord Spotting | 49x10x1 | 1x12 | 53.9 | 2.7M |
| Visual Wake Word | 96x96x3 | 1x2 | 333.3 | 7.5M |

Come si può notare dalla tabella, i modelli utilizzati per il benchmark sono di piccole dimensioni, con un numero di parametri che varia da 53.9 kB a 333.3 kB, compatibili con le limitazioni di memoria presenti sui microcontrollori.

### 2.3.2 Metriche di valutazione

Il benchmark identifica tre metriche principali per valutare le prestazioni dei dispositivi:

- **Latenza**: definita come il tempo necessario per eseguire un'iterazione del modello e viene espressa in millisecondi. Viene misurata per ogni iterazione utilizzando un timestamp inviato dal dispositivo al pc host all'inizio e alla fine di ogni iterazione. Il valore utilizzato viene calcolato come la media delle misurazioni ottenute su almeno 10 iterazioni del modello o un tempo di inferenza minimo di 10 secondi.

- **Consumo energetico**: definito come l'energia consumata durante l'esecuzione del modello e viene espressa in millijoule. Il consumo energetico di una singola iterazione viene calcolato come somma dei consumi istantanei misurati, mentre il valore utilizzato come metrica è calcolato come la media dei consumi ottenuti in tutte le iterazioni del modello.
  Questi due valori sono misurati utilizzando una scheda di misurazione descritta in dettaglio nel paragrafo 2.4.

- **Accuratezza**: il parametro che descrive l'accuratezza dipende dal modello utilizzato.

La scelta di utilizzare un timestamp generato dal dispositivo sotto test (DUT) per misurare la latenza è utile per evitare di considerare ritardi dovuti alla comunicazione seriale tra il microcontrollore e il pc o dovuti alla latenza introdotta dal sistema operativo. Questo approccio però richiede che il DUT sia in grado di tenere traccia del tempo trascorso dall'avvio del benchmark. La precisione minima richiesta per il timestamp è di 1 µs.

Nell'ambito di questa tesi, i valori di latenza e consumo energetico sono stati calcolati come media dei valori ottenuti su tutte le iterazioni del modello. Questa metodologia è stata adottata al fine di limitare l'incertezza dovuta alla non perfetta sincronizzazione tra i segnali di trigger di inizio e fine della fase di inferenza e le misurazioni di corrente e tensione effettuate dalla scheda di misurazione. Per garantire chiarezza e leggibilità, i grafici presentati in questo elaborato mostreranno solo un numero limitato di iterazioni.

### 2.3.3 Modalità di esecuzione e pubblicazione dei risultati

Il benchmark MLPerf-Tiny prevede due modalità di esecuzione e di pubblicazione dei risultati: *closed-division* e *open-division*.

La modalità *closed-division* viene utilizzata per ottenere risultati che siano confrontabili tra dispositivi diversi. Per pubblicare i risultati ottenuti in questa modalità è necessario utilizzare le reti neurali e i dataset forniti dalla suite di benchmark, senza apportare modifiche. Non è possibile quindi allenare nuovamente i modelli o modificare le configurazioni di esecuzione.

La modalità *open-division* viene invece utilizzata per permette ai partecipanti di mostrare le prestazioni massime che un dato dispositivo può raggiungere. In questa modalità è permesso effettuare delle modifiche ai modelli, ai dataset o alle metriche utilizzate durante il training, purchè queste modifiche siano dichiarate e documentate.

### 2.3.4 Struttura della libreria di benchmark

Il framework MLPerf Tiny mette a disposizione degli sviluppatori una libreria, da integrare nel proprio progetto, che fornisce un insieme di Application Programming Interface (API) per facilitare l'esecuzione dei benchmark. Tale libreria offre funzionalità essenziali come la decodifica dei comandi ricevuti dal pc host, la gestione dei timestamp e l'inizializzazione dei tensori di input e output.

La libreria adotta un approccio ibrido, fornendo sia funzioni pre-implementate che funzioni con solo il prototipo. Queste ultime, definite *submitter_implemented* devono essere implementate dallo sviluppatore in base all'architettura hardware

specifica e al compilatore utilizzato. Tale approccio consente di astrarre le funzionalità comuni, garantendo la portabilità del benchmark su diverse piattaforme, e al contempo di adattare il codice alle peculiarità di ciascun sistema.

L'interazione tra il firmware scritto dallo sviluppatore e la libreria di benchmark avviene tramite l'utilizzo di due funzioni principali:

- `ee_benchmark_initialize()`
- `ee_serial_callback(char ch)`

La prima funzione viene utilizzata per richiamare tutte le procedure necessari per l'inizializzazione del framework di benchmark e di LiteRT Micro. Questa funzione deve essere chiamata all'avvio del dispositivo.

La seconda invece, viene utilizzata per gestire i comandi ricevuti dal pc host. Questa funzione viene chiamata ogni volta che il dispositivo riceve un byte dal pc host.

## 2.4 Sistema di misurazione del consumo energetico

### 2.4.1 Standard MLPerf-Tiny

Il sistema di misurazione del consumo energetico definito da MLPerf-Tiny è composto da due dispositivi: una scheda per di interfaccia tra il DUT e il PC host e un circuito di alimentazione che possa misurare la corrente erogata. Il primo dispositivo, con alimentazione separata rispetto al DUT, viene utilizzato per escludere dalla misurazione l'energia necessaria per il pilotaggio della linea di comunicazione verso il PC host. Per quanto riguarda l'alimentatore invece, è necessario che questo sia in grado di comunicare tramite USB il valore di tensione e corrente erogata, in modo che il software di benchmarking possa calcolare i consumi istantanei.

Il software utilizzato per eseguire il benchmark è riceverà in input anche i valori di corrente e tensione misurati dal sistema di misurazione del consumo energetico e ne calcolerà la media per ottenere il consumo energetico medio del modello.

### 2.4.2 Sistema di misurazione del consumo energetico

Per lo scopo di questo lavoro è stato utilizzato un sistema di misurazione dei consumi diverso da quello definito da MLPerf-Tiny, in quanto non era disponibile l'hardware necessario per utilizzare il sistema di misurazione definito dal benchmark.
Per realizzare un sitema di misurazioni efficace è stato necessario rispettare due requisiti fondamentali: la capacità di eseguire misurazioni ad una frequenza elevata e la possibilità di misurare con precisione tensione e corrente.

Per misurare con precisione il consumo energetico durante l'inferenza è necessaria una frequenza di acquisizione sufficientemente elevata. Nel caso del benchmark

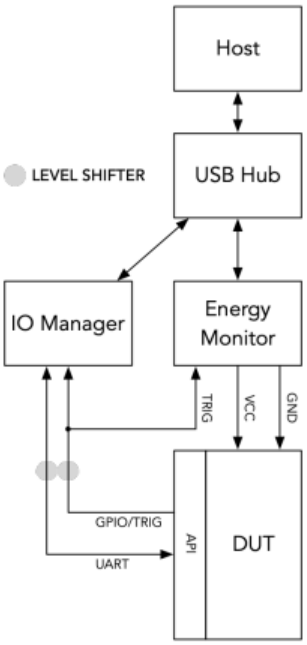



Figura 2.4: Schema del sistema di misurazione del consumo energetico MLPerf-Tiny

MLPerf-Tiny, il caso peggiore è rappresentato dell'esecuzione del modello più piccolo, eseguito sul dispositivo più veloce tra quelli utilizzati. Non conoscendo a priori l'intervallo di tempo in questione, è stato stimato, a partire dai risultati di dispositivi analoghi pubblicati per le versioni precedenti del benchmark, un tempo di inferenza minimo di circa 1 ms.
É stato quindi progettato un sistema di misurazione del consumo energetico in grado di effettuare acquisizioni ad una frequenza di almeno 1 kHz.
La precisione nella misurazione di corrente e tensione è essenziale per ottenere una stima affidabile del consumo energetico. Sebbene MLPerf Tiny non imponga un requisito minimo di precisione, è evidente che una maggiore accuratezza si traduca in una migliore valutazione delle prestazioni energetiche dei dispositivi.
Si è scelto di utilizzare un sensore di corrente e tensione che garantisse un trade-off tra precisione e velocità di acquisizione.

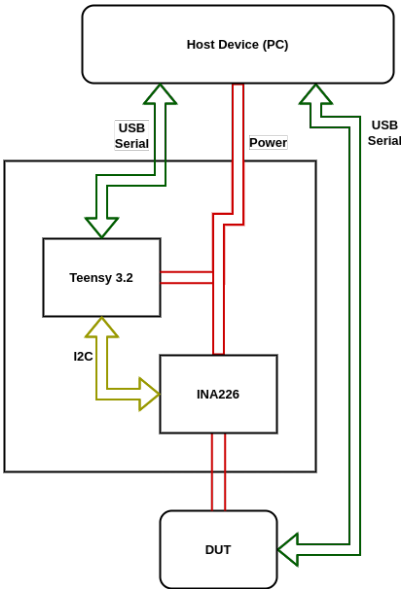



Figura 2.5: Schema del sistema di misurazione del consumo energetico utilizzato

Basato sulla scheda di prototipazione Teensy 3.2 e su un modulo di misurazione del consumo energetico INA226 [7], il circuito (figura 2.6) è stato progettato per poter misurare i consumi in contemporanea ai test effettuati per misurare la latenza di esecuzione e per inviare i dati in tempo reale al pc host, attraverso una connessione seriale ad alta velocità. Il microcontrollore presente sulla scheda è programmato per ricevere i valori di corrente assorbita e tensione di alimentazione acquisiti dal modulo INA226. Il consumo viene calcolato utilizzando le due misurazioni e l'intervallo di tempo che intercorre tra due misurazioni. Per evitare un flusso di dati eccessivo tra la scheda di misurazione dei consumi e il pc host, l'INA226 è stato configurato in modo tale da effettuare un'acquisizione dell misure ogni 204 µs.

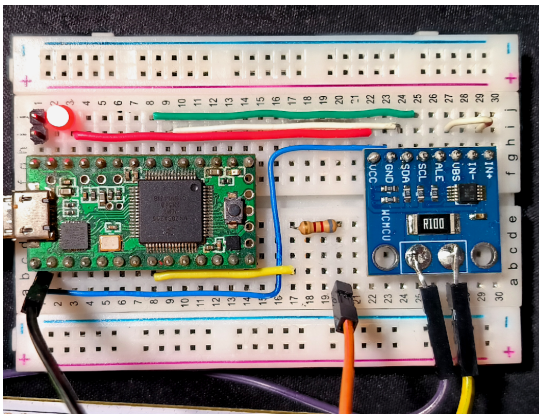


Figura 2.6: Foto della scheda di misurazione del consumo energetico

Poiché il protocollo di comunicazione utilizzato dal software del benchmark non è pubblico, è stato necessario sviluppare un software ad-hoc per l'elaborazione dei

dati acquisiti dalla scheda di misurazione dei consumi. Per ridurre il carico computazionale sul dispositivo host e permettere un'analisi a posteriori dei dati raccolti, sono stati realizzati due software separati rispettivamente per la ricezione e per la visualizzazione dei dati. Il primo si occupa di ricevere tramite interfaccia seriale i campioni e di scriverli in un file di log salvato sul pc. Il secondo script elabora i dati salvati per mostrarli in un'interfaccia grafica semplice e intuitiva (figura 2.7). Oltre a visualizzare i grafici di corrente, tensione ed energia, il software esegue automaticamente il calcolo della latenza e del consumo energetico per ciascuna iterazione del modello.

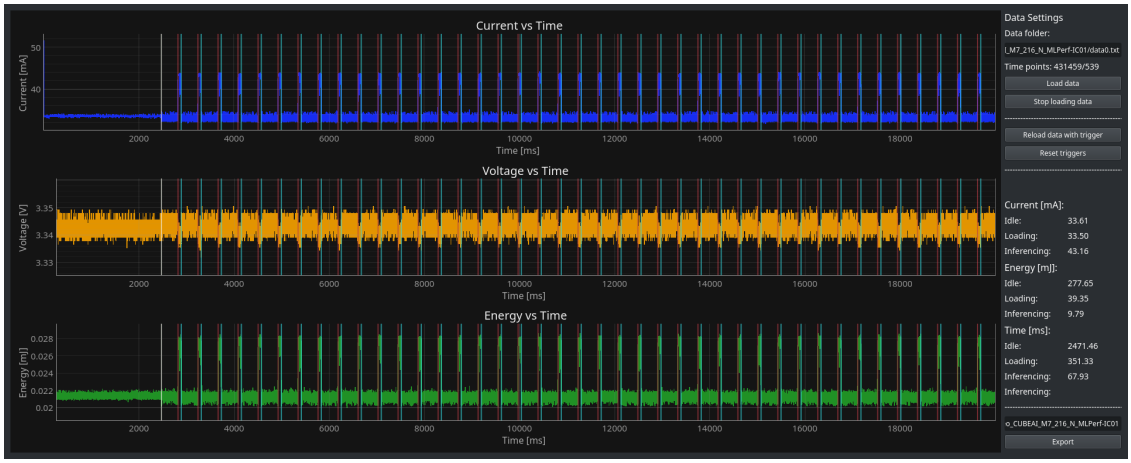



Figura 2.7: Software di misurazione del consumo energetico

I grafici riportati nell'immagine precedente mostrano rispettivamente (dall'alto verso il basso) l'andamento della corrente assorbita, della tensione di alimentazione e dell'energia consumata in ogni istante di tempo. Nella tabella a lato dell'immagine sono riportate le medie dei valori delle tre fasi dell'esecuzione dell'algoritmo. L'intervallo denominato *Idle* rappresenta il periodo di tempo che precede l'inizio della fase di test e dunque una fase in cui il microcontrollore non esegue operazioni. I valori calcolati per questo intervallo verranno utilizzati per calcolare l'aumento di consumi dovuto all'esecuzione dell'algoritmo. Gli altri due intervalli presenti sono chiamati *Loading* e *Inference* e rappresentano rispettivamente la fase di scambio dati tra DUT e pc host e la fase di inferenza.

# Capitolo 3

# Implementazione sui dispositivi

La flessibilità offerta dalla libreria LiteRT for microcontrollers ha reso possibile scegliere dispositivi con architetture differenti in modo da ottenere dei risultati rappresentativi delle diverse opzioni disponibili sul mercato dei dispositivi edge.
Di seguito, verranno presentati i dispositivi selezionati e le metodologie implementate per l'esecuzione degli algoritmi di benchmark.

## 3.1 Nucleo STM32H745

La prima piattaforma hardware testata è un microcontrollore (MCU) STM32H745 di STMicroelectronics.

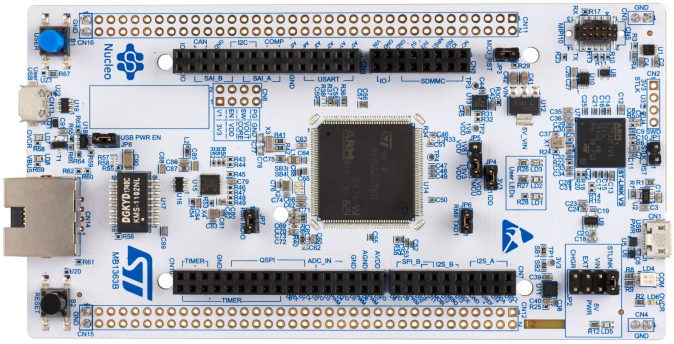


Figura 3.1: Scheda di sviluppo Nucleo-STM32H745ZI-Q

Tale scelta è motivata da diversi fattori, tra cui l'ampia diffusione e il solido supporto di cui godono i microcontrollori STM32, sia da parte del produttore che della community. Questo garantisce la disponibilità di numerose risolre, documentazione e assistenza, facilitando lo sviluppo di applicazioni e la risoluzione dei problemi. Inoltre, la presenza di schede di sviluppo dedicate semplifica l'utilizzo dei microcontrollori e delle loro periferiche, evitando la necessità di progettare schede custom. Un ulteriore vantaggio di questo microcontrollore è rappresentato dalla sua architettura dual-core, composta da un core Cortex®-M7 e un core Cortex®-M4. Questa

caratteristica consente di valutare le prestazioni di due architetture diverse senza la necessità di cambiare hardware.
Infine, STMicroelectronics offre *STM32Cube.AI* [8], un tool che promette di ottimizzare l'esecuzione di modelli LiteRT, presentandosi come un'alternativa potenzialmente più efficiente rispetto alle librerie standard di LiteRT Micro. L'adozione di un microcontrollore STM32H7 permette quindi di condurre i test su una piattaforma rappresentativa, facilmente accessibile e supportata da strumenti di sviluppo dedicati.

La scheda di sviluppo utilizzata è la Nucleo-STM32H745ZI-Q. Il microcontrollore integrato nella scheda presenta le seguenti caratteristiche principali:

- Core Arm® Cortex®-M7 a 32bit con FPU e frequenza di clock massima di 480 MHz
- Core Arm® Cortex®-M4 a 32bit con FPU e frequenza di clock massima di 240 MHz
- 2x MB di memoria flash
- 1x MB di RAM
- 4x UARTs e 1 LPUART
- 2x 32-bit timers
- 2x SysTick timers

La configurazione dual-core del microcontrollore STM32H7, sebbene non strettamente richiesta dal benchmark, si presta alla simulazione di scenari applicativi realistici. In tali contesti, generalmente, un core può essere dedicato a task real-time come la gestione delle periferiche e l'interazione con l'ambiente esterno (ad esempio, acquisizione dati da sensori, controllo di attuatori), mentre l'altro core può eseguire in background algoritmi computazionalmente intensi (come quelli di machine learning).
La programmazione e il debug della scheda di sviluppo avvengono tramite l'interfaccia ST-LINK integrata nel dispositivo. Questa interfaccia, sfruttando la connessione USB, permette non solo la programmazione del firmware, ma anche la comunicazione seriale (UART) con il microcontrollore. Questo canale di comunicazione bidirezionale è stato utilizzato per inviare comandi e dati al dispositivo, nonché per ricevere i risultati dei test eseguiti. Il software di benchmark utilizza di default una velocità di comunicazione di 9600 baud. Tuttavia, data la possibilità di configurare il baudrate del microcontrollore è stato scelto di incrementare la velocità di comunicazione fino a 115200 baud, riducendo notevolmente il tempio necessario alla trasmione dei dati.

Nelle schede di sviluppo Nucleo è presente un jumper di debug che permette di interrompere la linea di alimentazione del microcontrollore. Questa funzionalità è stata sfruttata per inserire la scheda di misurazione dei consumi in serie all'alimentazione del MCU. In questo modo è stato possibile eseguire le misurazioni di

corrente senza includere i consumi degli altri componenti presenti sulla scheda e non utilizzati durante i test.

Per la misurazione accurata della latenza di esecuzione dei modelli, è stato impiegato un timer hardware a 32 bit con risoluzione di 1 µs. Tuttavia, questa configurazione limita il tempo massimo misurabile a circa 72 minuti, dopo i quali il timer va in overflow. Per superare questa limitazione, è stato implementato un approccio ibrido che utilizza in parallelo un secondo timer, il *SysTick*, con risoluzione di 1 ms. La combinazione dei due timer consente di ottenere sia l'elevata risoluzione del timer hardware a 32 bit che l'ampio intervallo di misurazione del *SysTick*, garantendo una misurazione precisa della latenza anche per test di lunga durata.

Al fine di poter eseguire i test utilizzando anche le librerie ottimizzate sviluppate da STMicroelectronics, per la configurazione del microcontrollore e la scrittura del codice è stato utilizzato l'IDE STM32CubeIDE.
Il plugin *STM32Cube.AI*, installabile allinterno dell'IDE, offre una serie di funzionalità che semplificano notevolmente il processo di sviluppo, dall'ottimizzazione dei modelli neurali fino alla loro esecuzione efficiente sull'hardware target. Le principali funzionalità offerte dal plugin sono:

- Conversione dei modelli TensorFlow in formato compatibile con il microcontrollore
- Ottimizzazione del modello per l'esecuzione su microcontrollori STM32
- Ottimizzazioni per aumentare le prestazioni o ridurre i consumi energetici

### 3.1.1 Nucleo STM32H745 e LiteRT for Microcontrollers

In questo paragrafo verrà descritta l'implementazione dei modelli di benchmark eseguiti utilizzando la libreria LiteRT for Microcontrollers. Come descritto nel paragrafo 2.2, è possibile (e consigliato) compilare il codice eseguibile della libreria per una specifica architettura hardware.
In questo caso, per eseguire i test su entrambi i core del MCU è stato necessario compilare il codice per entrambe le architetture. Oltre alla scelta dell'architettura, è possibile configurare la libreria per sfruttare le istruzioni DSP e MVE presenti nei core Cortex®-M. Queste istruzioni permettono di incrementare maggiormente le prestazioni sfruttando al massimo le potenzialità del core. Per attivare queste ottimizzazioni è necessario abilitare la libreria CMSIS-NN [9].
Non sono stati eseguiti test utilizzando la libreria non ottimizzata in quanto sia prevedibile che le prestazioni sarebbero state nettamente inferiori rispetto a quelle ottenute con l'ottimizzazione abilitata.

Per quanto riguarda l'utilizzo delle API MLPerf-Tiny, l'inizializzazione è stata eseguita in coda alle funzioni di inizializzazione scritte da CubeIDE (sezione *User code 2*), mentre la chiamata alla funzione callback per la gestione dei byte ricevuti viene effettuata all'interno del loop principale, a seguito della ricezione di un byte.

**Chiamata alla funzione di callback**

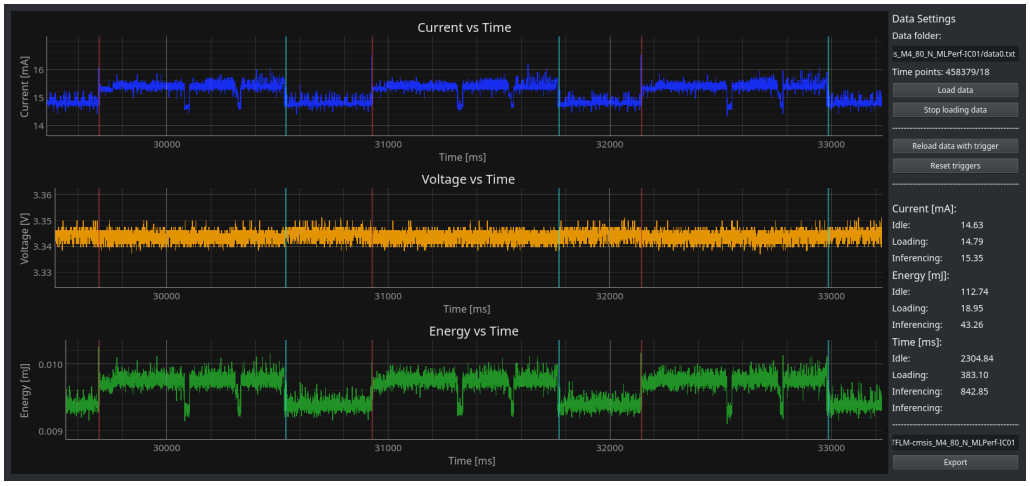
```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1){
/* USER CODE END WHILE */
    HAL_StatusTypeDef retVal;
    retVal = HAL_UART_Receive (&huart3, UART_rxBuffer, 1, 100);

    if (retVal == HAL_OK) {
        ee_serial_callback(UART_rxBuffer[0]);
    }
    /* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */
```

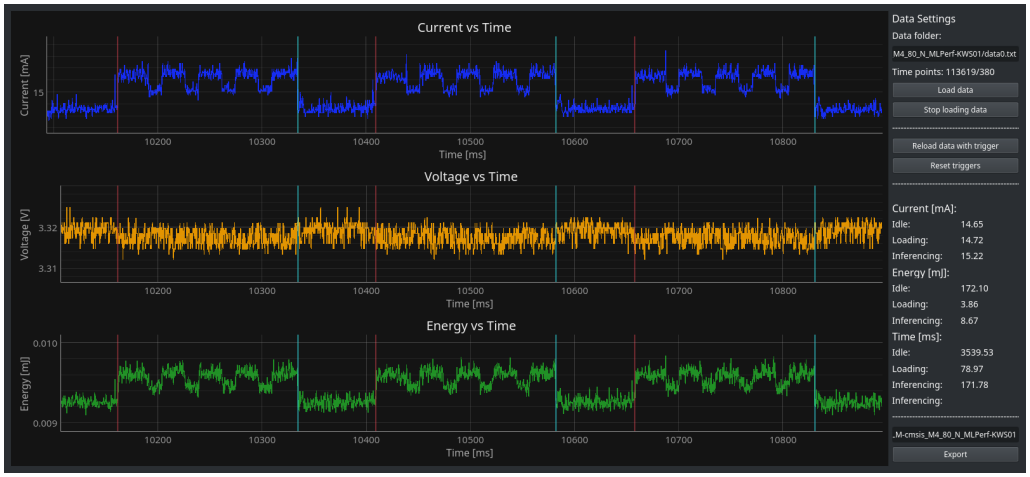
### Core Cortex®-M4

Il primo test è stato eseguito sul core Cortex®-M4 del microcontrollore STM32H745, operante a una frequenza di clock di 80 MHz. Questa, pur essendo inferiore alla frequenza massima supportata dal core, rappresenta un compromesso che consente di valutare le prestazioni del sistema in condizioni operative tipiche per applicazioni embedded. L'architettura Cortex®-M4, infatti, è ampiamente diffusa in dispositivi che richiedono un bilanciamento tra efficienza energetica e capacità di elaborazione, come ad esempio sistemi di controllo industriale, dispositivi IoT e wearable.
Vengono riportati di seguito i grafici generati tramite il software utilizzato per elaborare i dati acquisiti durante i test. Per una migliore analisi dei risultati, sono state utilizzate due linee verticali di colori distinti per delimitare le diverse fasi di funzionamento. In particolare, la linea rossa indica l'inizio della fase di inferenza, durante la quale il modello di machine learning viene eseguito sul microcontrollore, mentre una linea azzurra indica la fine dell'inferenza e l'inizio della fase di scambio dati.

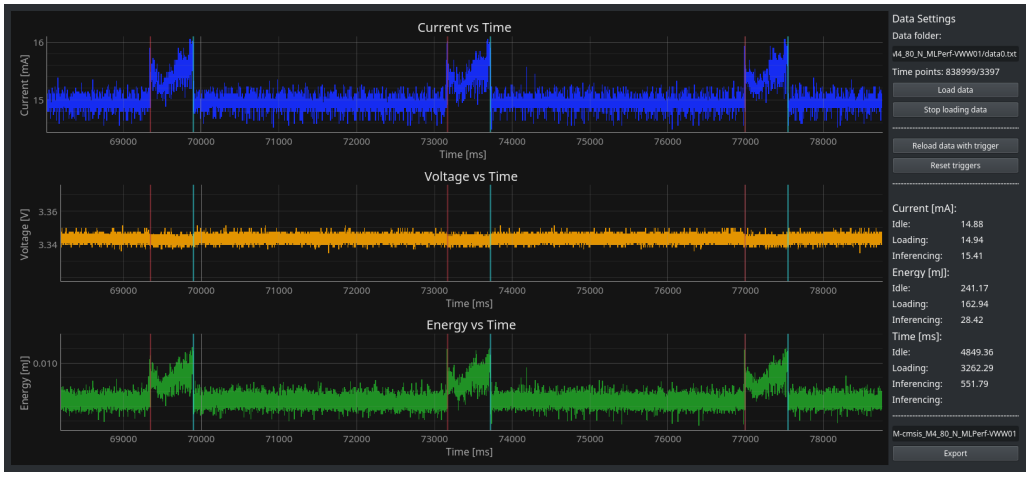

(a) Anomaly Detection Network




(b) Image Classification Network




(c) Keywork Spotting Network




(d) Visual Wakeword Network

Figura 3.2: Risultati dei test eseguiti sul core M4 a 80 MHz

L'analisi dei grafici rivela una correlazione diretta tra la complessità del modello, misurata in termini di numero di operazioni eseguite durante l'inferenza, e l'aumento del tempo necessario per l'esecuzione. Questo incremento si traduce in un conseguente aumento dell'energia consumata dal dispositivo. È interessante notare come l'intesità della corrente assorbita durante la fase di elaborazione non sembra dipendere dalla rete utilizzata ma sia piuttosto una caratteristica intrinseca del microcontrollore.
Nei grafici 3.2b e 3.2c è possibile notare come la fase di inferenza sia suddivisa in diverse sotto-fasi caratterizzate da diversi livelli di assorbimento di corrente. Queste variazioni sono attribuibili alla diversa complessità computazionale dei layer che compongono la rete neurale: layer più complessi, che richiedono un numero di operazioni al secondo maggiore, determinano picchi di consumo energetico, mentre layer più semplici si traducono in un consumo inferiore.

I test successivi saranno finalizzati ad approfondire queste osservazioni, indagando la relazione tra consumo energetico e configurazione del microcontrollore.

Al fine di facilitare l'analisi comparativa, le metriche calcolate per la fase di inferenza, oltre ai valori medi di corrente assorbita in condizioni di inattività (*idle*), sono stati riportati nella tabella sottostante.

| **Modello** | **Corrente Idle (mA)** | **Tempo (ms)** | **Corrente (mA)** | **Avg Power (mW)** | **Energy (mJ)** |
|---|---|---|---|---|---|
| Anomaly Detection | 15.01 | 18.13 | 15.83 | 52.24 | 0.95 |
| Image Classification | 14.63 | 823.03 | 15.325 | 43.97 | 42.19 |
| Keyword Spotting | 14.65 | 172.90 | 15.21 | 50.16 | 8.72 |
| Visual WakeWord | 14.87 | 551.72 | 15.40 | 50.82 | 28.40 |

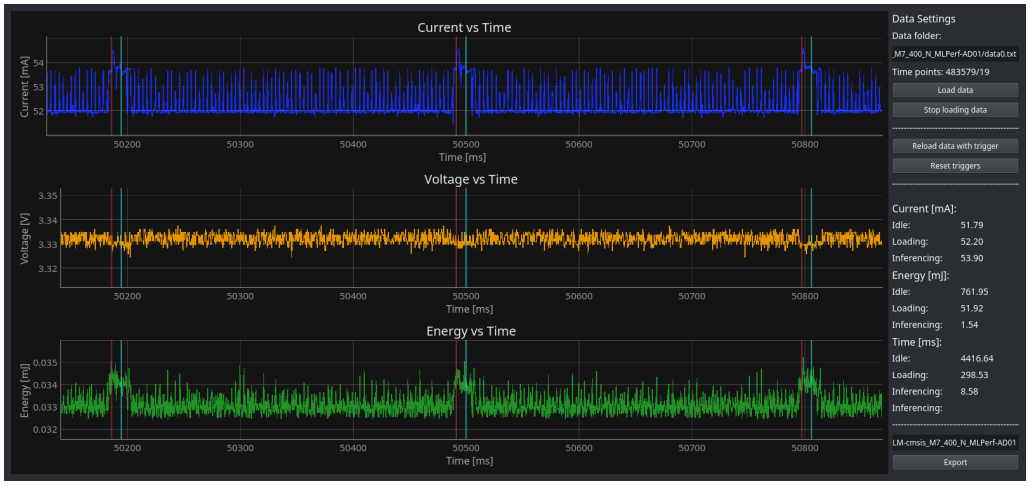
Tabella 3.1: Risultati dei test eseguiti sul core M4 a 80 MHz

Le metriche calcolate per la fase di caricamento visibili nelle immagini non saranno utilizzate per confrontare le prestazioni dei diversi dispositivi in quanto sono relative all'esecuzione del benchmark e non ad un eventuale utilizzo del dispositivo in un'applicazione reale.

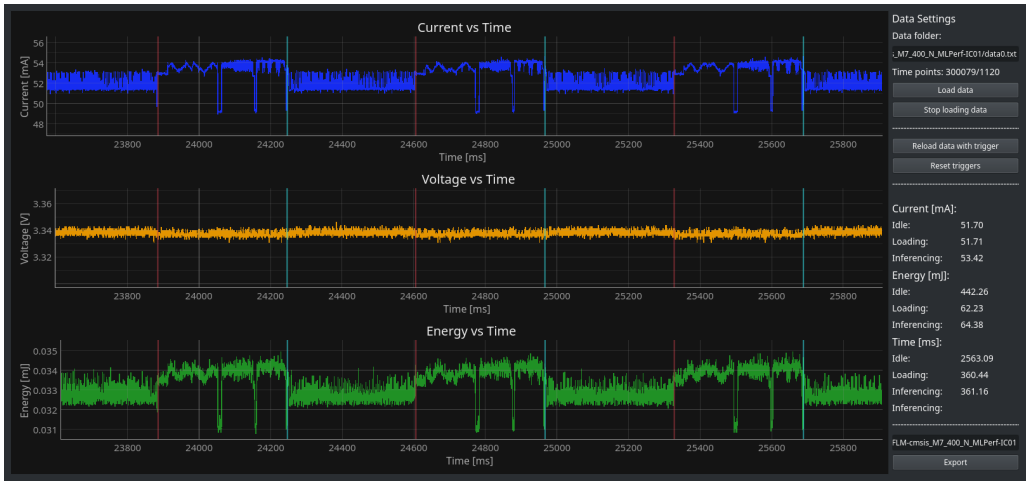
### Core Cortex®-M7 a 400MHz

Per la valutazione del core Cortex®-M7 del microcontrollore STM32H745, sono stati condotti due set di test a differenti frequenze di clock.
Inizialmente si prevedeva di utilizzare la frequenza di clock massima a cui può operare il microcontrollore, ovvero 480 MHz. Tuttavia a causa di alcune problematiche emerse durante la configurazione del dispositivo, si è optato per una frequenza di clock leggermente inferiore, pari a 400 MHz. Nonostante questa limitazione, i risultati ottenuti in queste condizioni sono comunque indicativi delle prestazioni massime che possono essere ottenute con questo dispositivo.

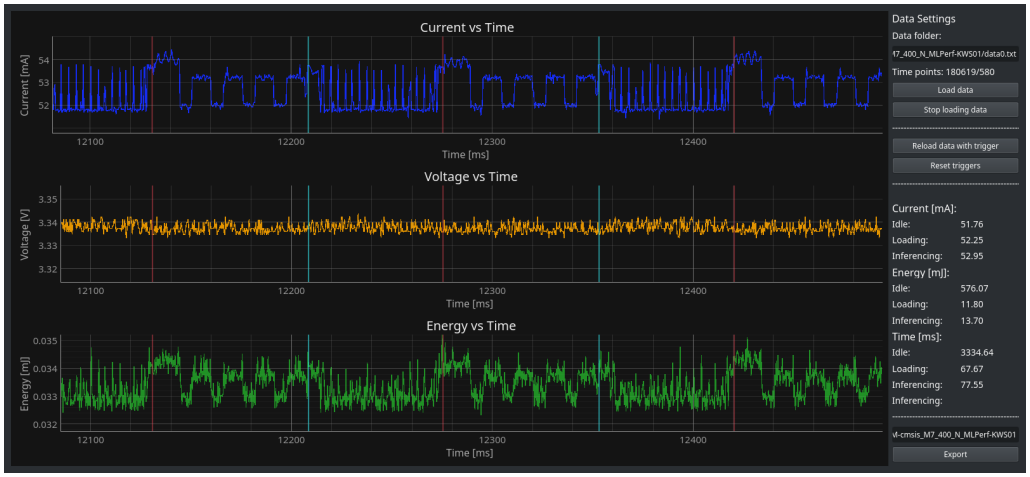
Rispetto ai risultati ottenuti nel test precedente, ci si aspetta che, a fronte di un aumento della potenza di calcolo e della frequenza di clock, corrisponda sia una riduzione significativa della latenza che un aumento del consumo energetico, sia nella fasse di inattività e di caricamento dei dati, sia durante le inferenze.



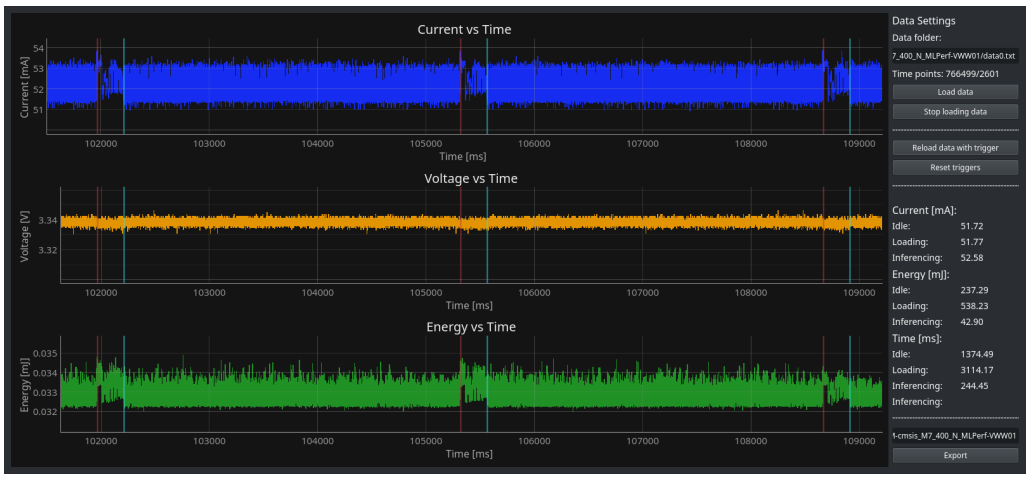

(a) Anomaly Detection Network




(b) Image Classification Network




(c) Keywork Spotting Network




(d) Visual Wakeword Network

Figura 3.3: Risultati dei test eseguiti sul core M7 a 400 MHz

Analogamente ai grafici relativi ai test eseguiti sul core M4, anche in questo caso si osserva una differenza nel consumo energetico tra le fasi di inferenza e di idle/-scambio dati. In queste condizioni, però, le differenze risultano essere meno marcate. Questo differenza può essere attribuita alla maggiore potenza richiesta per l'alimentazione del core utilizzato. Inoltre, l'elevata frequenza di clock è cause di picchi di assorbimento maggiori e, di conseguenza, rumore maggiore nelle misurazioni della corrente.
Anche in questo caso, nei grafici relativi ai modelli di Image Classification e Keyword Spotting, si possono notare delle fasi di inferenza suddivise in diversi stadi, anche se la loro individuazione risulta meno evidente.

| Modello | Corrente Idle (mA) | Tempo (ms) | Corrente (mA) | Avg Power (mW) | Energy (mJ) |
|---|---|---|---|---|---|
| Anomaly Detection | 51.79 | 8.58 | 53.90 | 177.87 | 1.54 |
| Image Classification | 51.70 | 361.16 | 53.42 | 176.29 | 64.38 |
| Keyword Spotting | 51.76 | 77.55 | 52.95 | 174.74 | 13.70 |
| Visual WakeWord | 51.72 | 244.45 | 52.58 | 173.51 | 42.90 |

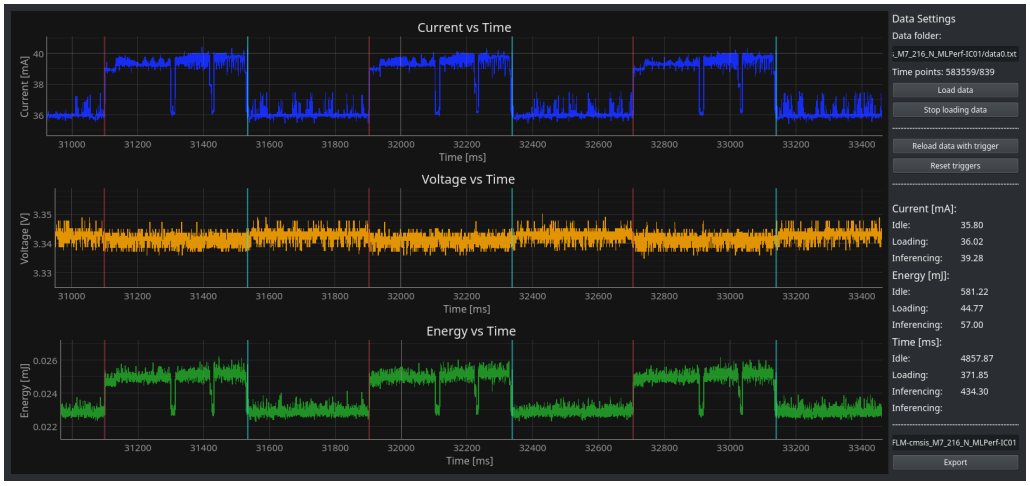
Tabella 3.2: Risultati dei test eseguiti sul core M7 a 400 MHz

Dai valori riportati in tabella 3.1.1 si può notare che i risultati ottenuti mantengono lo stesso andamento di quelli ottenuti nel test precedente. La previsione fatta in precedenza si è rivelata corretta. Si può notare infatti come la latenza dei diversi modelli si sia quasi dimezzata e, nonostante questo, i consumi siano aumentati di circa il 55%.

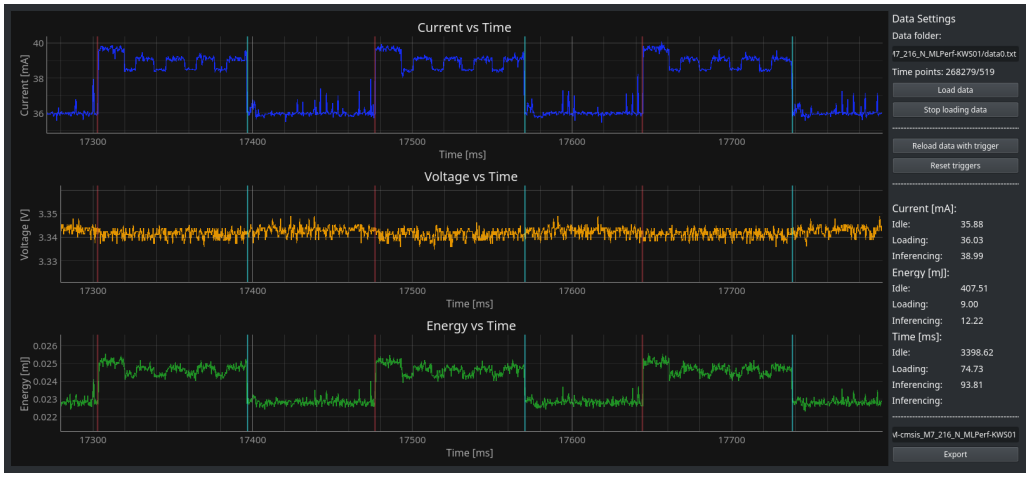
### Core Cortex®-M7 a 216MHz

Per il secondo test effettuato sul core M7 del microcontrollore è stata scelta una frequenza intermedia rispetto a quelle utilizzate nei test precedenti. Questa scelta mira a verificare se l'aumento delle prestazioni e dei consumi energetici sia lineare rispetto alla frequenza di clock del core.

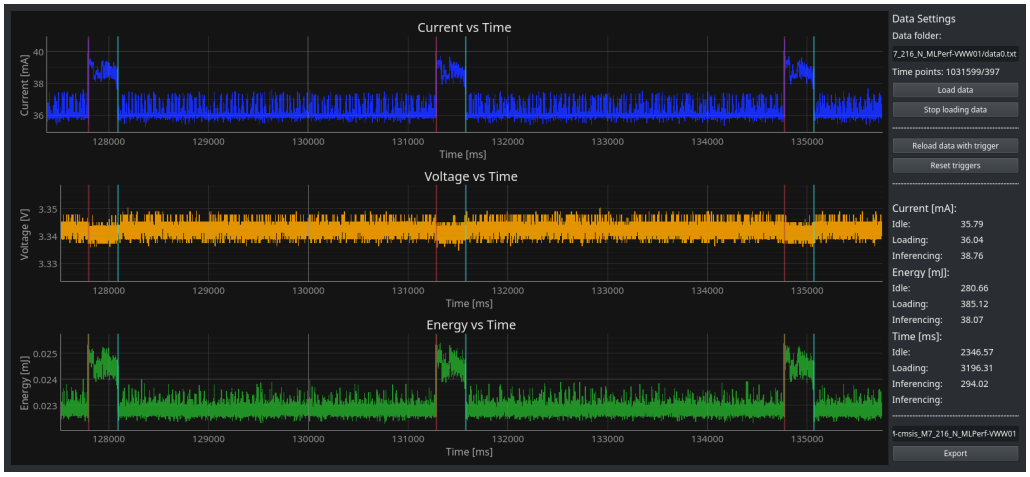

(a) Anomaly Detection Network




(b) Image Classification Network




(c) Keywork Spotting Network




(d) Visual Wakeword Network

Figura 3.4: Risultati dei test eseguiti sul core M7 a 216 MHz

Possiamo notare come i grafici risultino meno rumorosi rispetto a quelli ottenuti con una frequenza di clock maggiore e anche la differenza tra le fasi risulta essere nuovamente più marcata.

| **Modello** | **Corrente Idle (mA)** | **Tempo (ms)** | **Corrente (mA)** | **Avg Power (mW)** | **Energy (mJ)** |
|---|---|---|---|---|---|
| Anomaly Detection | 37.95 | 9.54 | 41.06 | 135.50 | 1.31 |
| Image Classification | 35.80 | 434.30 | 39.28 | 129.62 | 57.00 |
| Keyword Spotting | 35.88 | 93.81 | 38.99 | 128.67 | 12.22 |
| Visual WakeWord | 35.79 | 294.02 | 38.76 | 129.91 | 38.07 |

Tabella 3.3: Risultati dei test eseguiti sul core M7 a 216 MHz

Come previsto, i risultati ottenuti con la frequenza di clock intermedia si collocano tra quelli ottenuti con le frequenze più alta e più bassa. Sia la latenza che il consumo energetico hanno raggiunto valori intermedi, confermando l'influenza diretta della frequenza di clock su entrambe le metriche. Effettuando un'analisi comparativa con i risultati precedenti, si osserva un aumento dei tempi di esecuzione di circa il 20% a fronte di un a riduzione dei consumi di circa il 12%. Questo dato conferma che la modifica della frequenza di clock, a parità di configurazione del sistema, consente di ottenere un trade-off tra prestazioni e consumo energetico.

La riduzione della frequenza di clock rispetto al test precedente ha anche comportato una diminuzione sia della corrente assorbita in fase di inattività che della corrente assorbita durante le inferenze. Questo risultato evidenzia come la frequenza di clock sia un fattore determinante nel consumo energetico del microcontrollore, in tutte le fasi di funzionamento.

### 3.1.2 Nucleo STM32H743 e Cube.AI

Per valutare l'impatto delle ottimizzazioni specifiche per l'architettura STM32, il benchmark è stato eseguito utilizzando il plugin *Cube.AI*. In questo caso, il codice per l'inizializzazione della rete neurale e per l'esecuzione delle inferenze è stato generato automaticamente da *Cube.AI*. Questa automazione semplifica il processo di sviluppo e consente di sfruttare le ottimizzazioni specifiche del plugin.

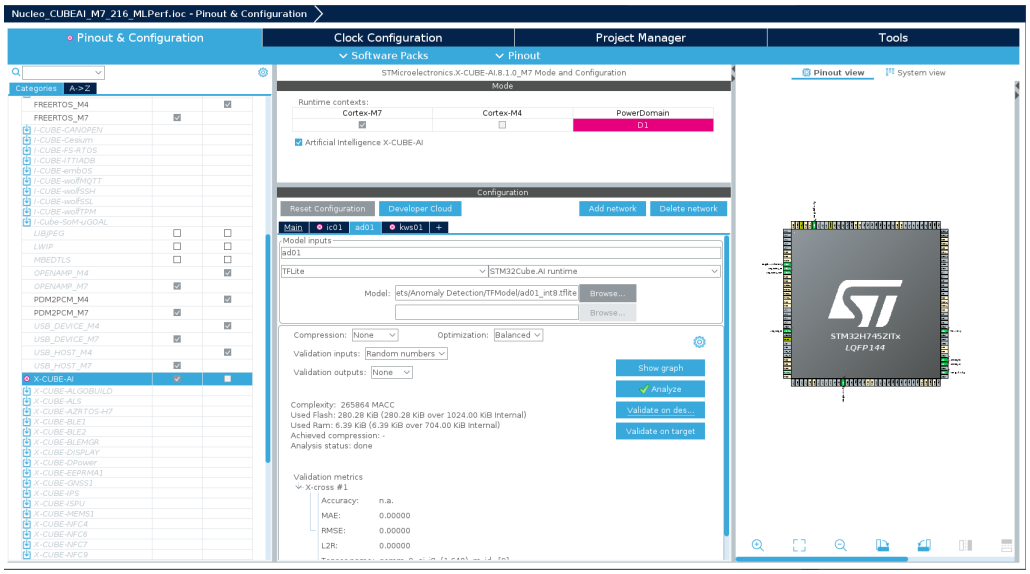
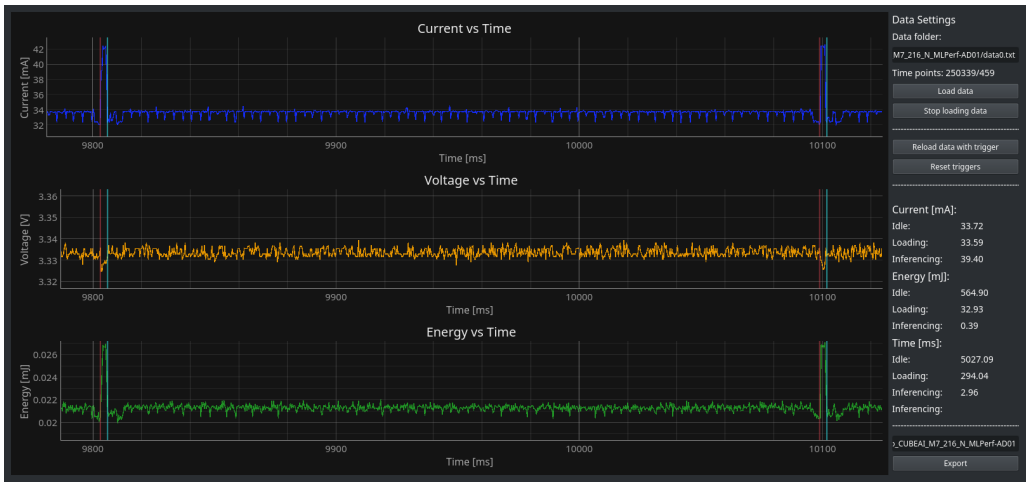


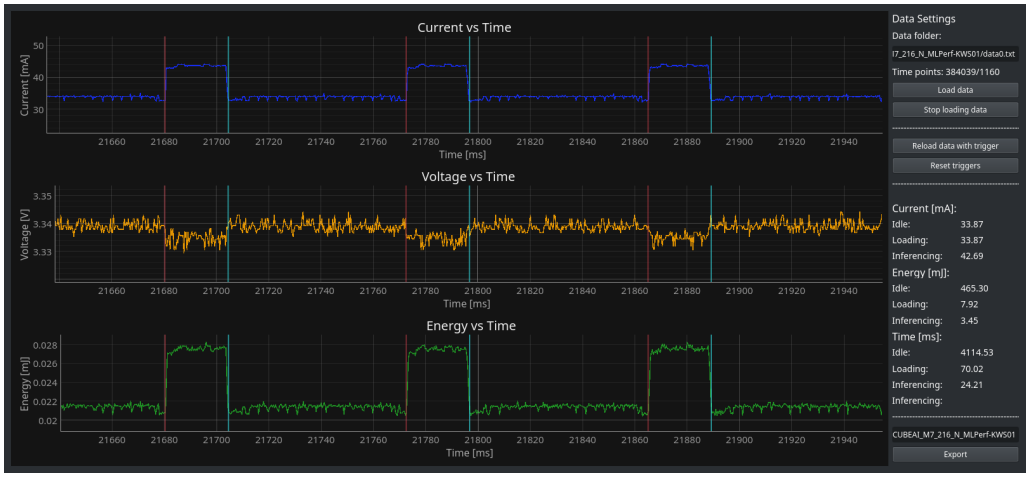

Figura 3.5: Screenshot delle funzionalità offerte dal plugin Cube.AI

Considerando le potenziali ottimizzazioni introdotte da Cube.AI, si è ipotizzato un miglioramento delle prestazioni rispetto all'utilizzo di LiteRT Micro. Per questo motivo, si è scelto di eseguire i test a una frequenza di clock di 216 MHz. Si pensava infatti che, utilizzando questo valore di frequenza, si sarebbero ottenute prestazioni intermedie rispetto ai due test precedenti.

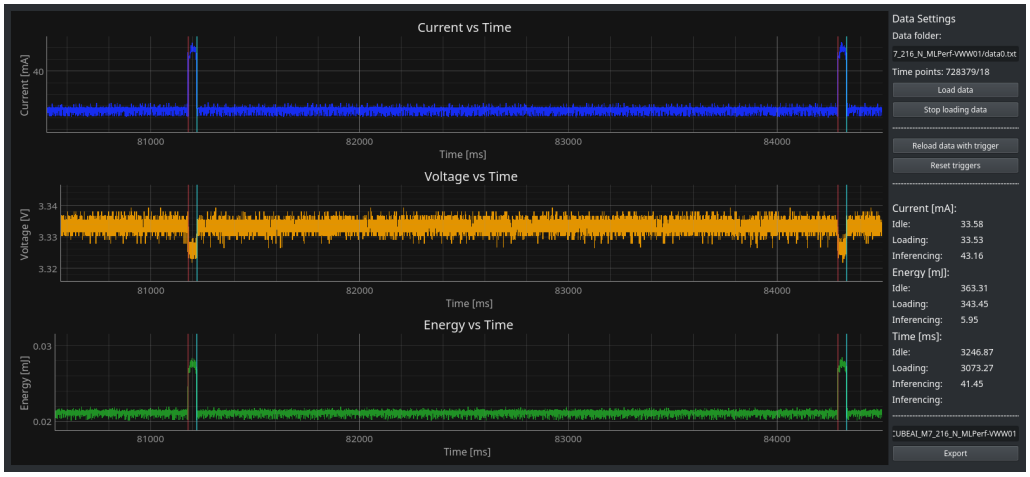



(a) Anomaly Detection Network


(b) Image Classification Network




(c) Keywork Spotting Network




(d) Visual Wakeword Network

Figura 3.6: Risultati dei test eseguiti con Cube.AI sul core M7 a 216 MHz

Dai grafici si può già notare una differenza significativa rispetto ai test eseguiti con LiteRT for Microcontrollers. Nello specifico, si osserva una riduzione significativa del rapporto tra tempo di inferenza e tempo di trasferimento dati, parametro che era rimasto pressoché costante a prescindere dalla frequenza di clock utilizzata. Inoltre, l'assenza di stadi intermedi nei profili di corrente ed energia durante l'inferenza indica una probabile ristrutturazione della rete neurale operata dal plugin *Cube.AI* durante la fase di ottimizzazione. Tale modifica potrebbe contribuire alla riduzione del tempo di inferenza e al miglioramento dell'efficienza energetica.

| **Modello** | **Corrente Idle (mA)** | **Tempo (ms)** | **Corrente (mA)** | **Avg Power (mW)** | **Energy (mJ)** |
|---|---|---|---|---|---|
| Anomaly Detection | 33.72 | 2.96 | 39.40 | 130.02 | 0.39 |
| Image Classification | 33.61 | 67.90 | 43.16 | 142.43 | 9.79 |
| Keyword Spotting | 33.87 | 24.21 | 42.69 | 140.88 | 3.45 |
| Visual WakeWord | 33.58 | 41.45 | 43.16 | 142.43 | 5.95 |

Tabella 3.4: Risultati dei test eseguiti con Cube.AI sul core M7 a 216 MHz

L'analisi comparativa delle prestazioni con la libreria LiteRT for Microcontrollers evidenzia un comportamento peculiare; nonostante il consumo di corrente del microcontrollore si mantenga sostanzialmente costante in entrambe le fasi operative, si registra una drastica riduzione della latenza di esecuzione.

A titolo esemplificativo, nel modello di Image Classification, la latenza si riduce di circa l'85% rispetto all'implementazione con LiteRT for Microcontrollers alla medesima frequenza.
Sorprendentemente, le prestazioni superano persino quelle ottenute con una frequenza di clock di 400 MHz, con una riduzione della latenza dell'80%.

Pertanto, sebbene il consumo di corrente rimanga invariato, la diminuzione della latenza si traduce in un notevole risparmio energetico complessivo. Questo risultato sottolinea l'efficacia dell'ottimizzazione nell'incrementare le prestazioni senza impattare significativamente sul consumo energetico.

## 3.2 Coralmicro

Nell'ambito della ricerca di soluzioni per l'implementazione efficiente di algoritmi di intelligenza artificiale su microcontrollori, si è valutata l'influenza di un acceleratore hardware dedicato su consumi e prestazioni.

La scelta del dispositivo è stata guidata dalla necessità di compatibilità con il benchmark MLPerf-Tiny e la piattaforma di misurazione dei consumi preesistente. I requisiti fondamentali includevano il supporto del framework *LiteRT* for Microcontrollers e la presenza di interfacce UART e GPIO rispettivamente per la comunicazione e la generazione dei segnali di trigger.

La *Dev Board Micro* di Coral, divisione di Google dedicata allo sviluppo di soluzioni AI per dispositivi edge, è stata selezionata come piattaforma di test. Tale scheda integra un microcontrollore dual-core NXP, un acceleratore hardware *Coral Edge TPU* e una serie di periferiche di I/O per l'interazione con l'ambiente esterno.

Le specifiche tecniche della scheda sono le seguenti:

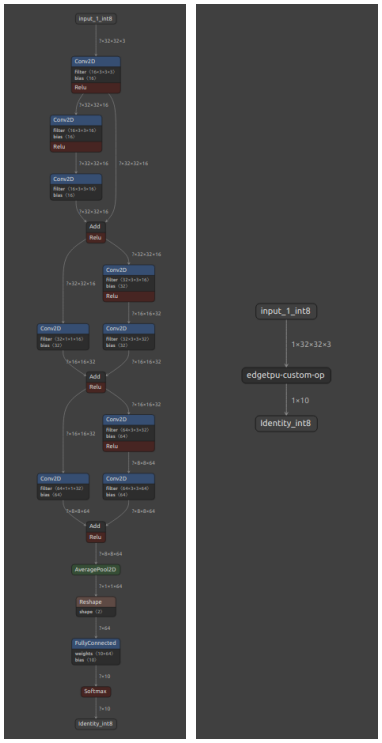
- NXP i.MX RT1176 dual core Cortex-M7 and Cortex-M4 a 800 MHz
- Coral Edge TPU da 4 TOPS (int8)
- 64 MB di memoria RAM
- 128 MiB di memoria flash


Figura 3.7: Coral Dev Board Micro

La programmazione del microcontrollore può essere effettuata in C/C++ (tramite qualsiasi IDE), sfruttando la libreria *coralmicro* fornita da Coral. Basata su FreeRTOS, questa libreria offre funzionalità di gestione di task, code e semafori, tipiche di un sistema operativo real-time, oltre a API per l'interazione con le diverse periferiche integrate (camera, microfono, EdgeTPU) e per l'esecuzione di modelli di Machine Learning tramite *LiteRT*.

Dal manuale presente online e dagli esempi forniti è emerso che la Dev Board Micro è stata progettata principalmente per essere connessa ad un PC host attraverso una connessione ethernet simulata tramite USB. Per utilizzare la scheda e permetteere uno scambio di dati tramite protocollo seriale è stato necessario risolvere alcuni problemi presenti nel codice sorgente. In particolare è stato necessario modificare il codice del task FreeRTOS che si occupa di gestire il buffer circolare della comunicazione seriale (appendice A.1).
Il caricamento dei modelli sulla scheda avviene tramite uno script Python, reperibile nel repository ufficiale [10], che combina il codice eseguibile e il modello *LiteRT* (non convertito in flatbuffer) in un unico file binario.

La Coral Dev Board Micro supporta due modalità di esecuzione dei modelli TensorFlow: modelli standard e modelli compilati per l'EdgeTPU. I modelli standard non richiedono particolari ottimizzazioni e possono essere eseguiti indistintamente su entrambi i core del microcontrollore. Al contrario, per essere eseguiti sull'EdgeTPU è necessario che i layer siano stati quantizzati a 8 bit e che il modello sia stato compilato utilizzando il tool *edgetpu-compiler*. Questo strumento si occupa di separare i layer che soddisfano i requisiti dell'acceleratore e li compatta in un unico layer, la restante parte del modello verrà estratta dal file originale e verrà eseguita sulla CPU. É anche possibile forzare l'esecuzione di uno o più layer sull CPU, ma questo comporta un aumento dei consumi e dei tempi di esecuzione. Per questo motivo non sono stati eseguiti test utilizzando questa funzione.




(a) (b)

Figura 3.8: Anomaly Detection Network

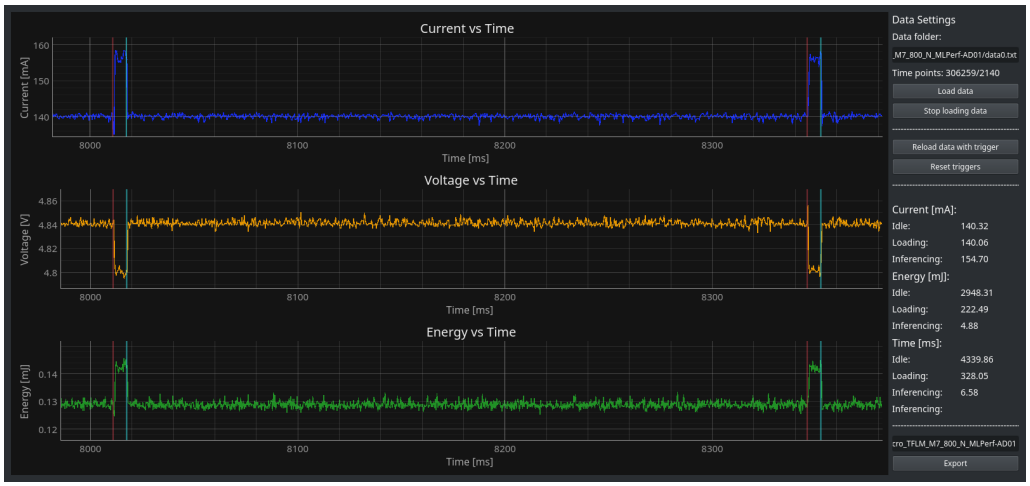
In figura 3.8 è possibile vedere la differenza tra un modello standard (Anomaly Detection) la versione compilata per l'EdgeTPU.
In questo caso, in cui la rete originale era stata interamente quantizzata a 8 bit, il modello compilato per l'EdgeTPU è composto da un solo layer, chiamato *edgetpu-custom-op*.

La misurazione dei consumi è risultata essere più complessa rispetto ai casi precendenti: al contrario delle schede di sviluppo ST, la Coral Dev Board Micro non prevede un sistema per la misurazione delle corrente assorbita dal microcontrollore e dall'acceleratore. Per ovviare a questo problema è stato necessario collegare la scheda di misurazione dei consumi in serie all'alimentazione dell'intera scheda. Ciò ha permesso di eseguire le misurazioni ugualmente, ma con lo svantaggio di non poter isolare i consumi dei diversi componenti presenti.

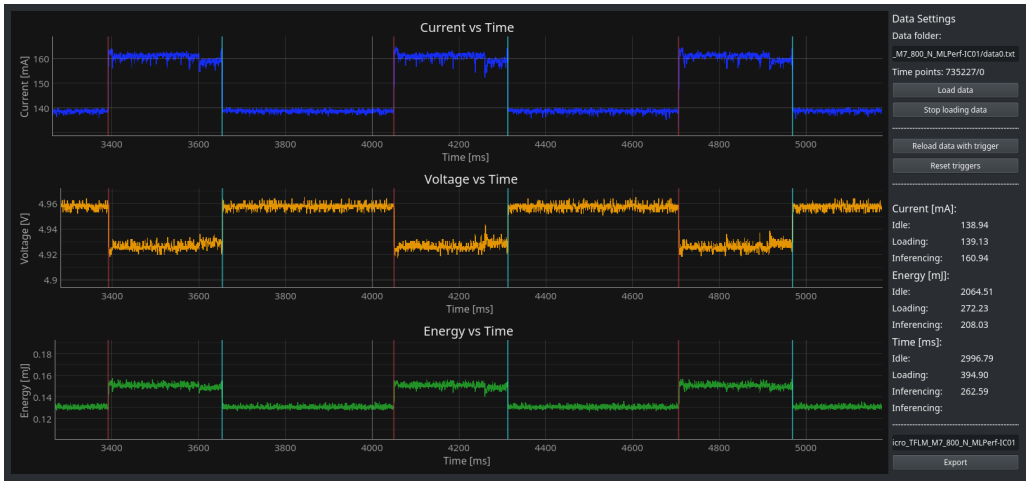
I valori riportati, quindi, sono da considerarsi un upper bound dei consumi del microcontrollore e dell'acceleratore.

### 3.2.1 Esecuzione dei modelli su MCU

Nella prima iterazione dei test, i modelli *LiteRT for Microcontrollers* sono stati eseguiti sfruttando solamente il core Cortex-M7 del MCU. Dato che non è prevista la possibilità di modificare la frequenza di clock, è stato eseguito solo un test alla frequenza massima di 800 MHz.

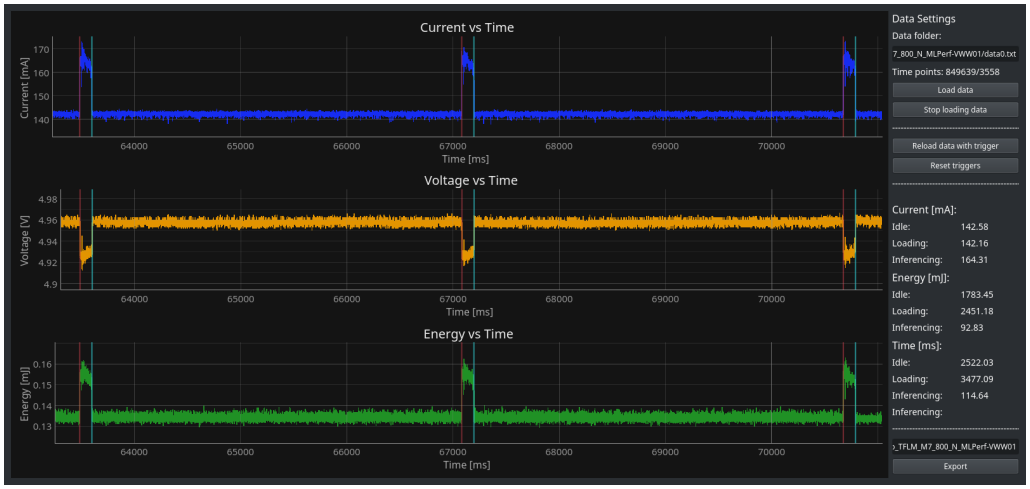



(a) Anomaly Detection Network




(b) Image Classification Network


(c) Keywork Spotting Network




(d) Visual Wakeword Network

Figura 3.9: Risultati dei test di esecuzione dei modelli sulla CPU a 800MHz

| **Modello** | **Corrente Idle (mA)** | **Tempo (ms)** | **Corrente (mA)** | **Avg Power (mW)** | **Energy (mJ)** |
|---|---|---|---|---|---|
| Anomaly Detection | 140.32 | 6.58 | 154.70 | 742.56 | 4.88 |
| Image Classification | 138.94 | 262.59 | 160.94 | 772.51 | 208.03 |
| Keyword Spotting | 143.07 | 40.76 | 168.70 | 809.76 | 33.24 |
| Visual WakeWord | 142.58 | 114.64 | 164.31 | 788.69 | 92.83 |

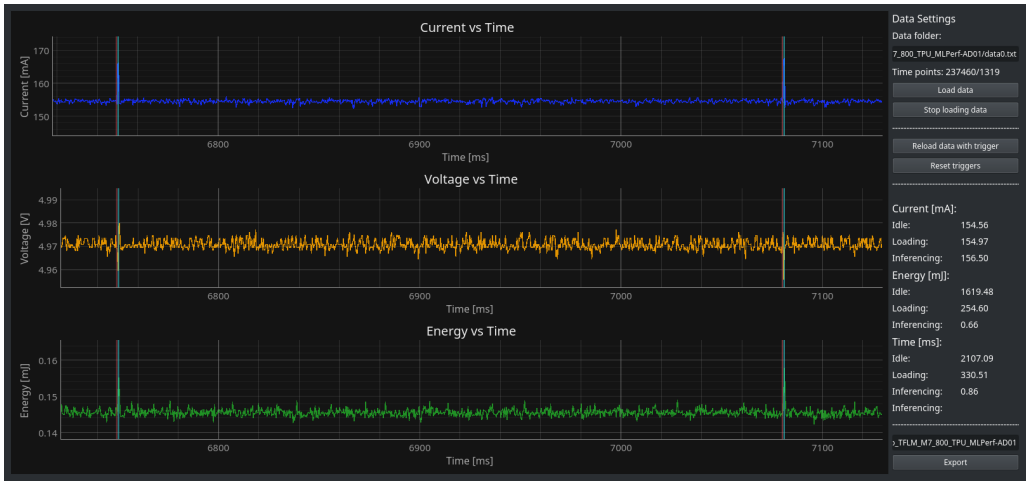
Tabella 3.5: Risultati dei test di esecuzione dei modelli sulla CPU a 800MHz

É opportuno confrontare i risultati appena ottenuti con quelli presenti in tabella 3.1.1 in quanto in entrambi i casi viene utilizzato un core con la stessa architettura. L'analisi comparativa evidenzia come un incremento del 100% della frequenza di clock determini una riduzione del tempo di inferenza del 50% e un aumento del consumo di corrente del 220%.

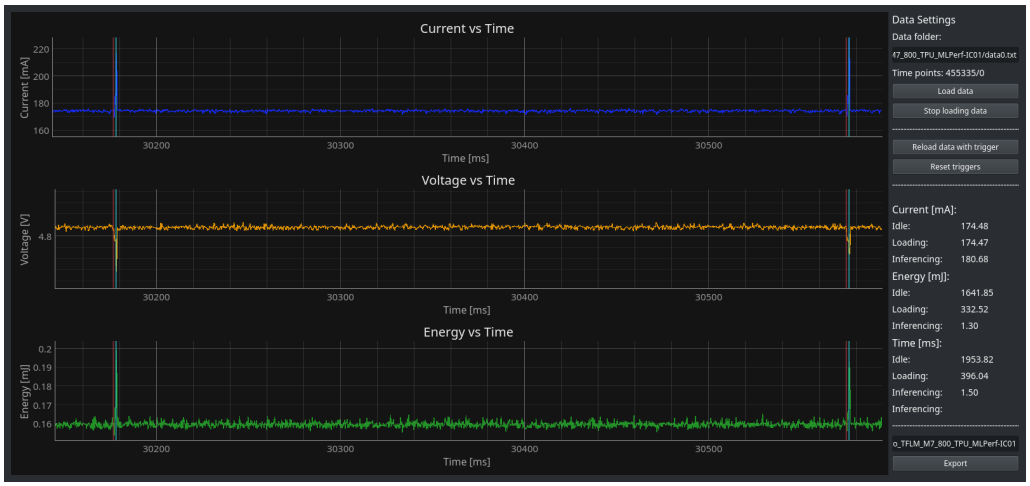
Come atteso, si osserva un consumo energetico significativamente maggiore rispetto alla scheda Nucleo. Questi valori sono dovuti ad una misurazione che include anche gli altri componenti presenti sulla scheda, come l'acceleratore EdgeTPU, le memorie esterne e i circuiti per la gestione dell'alimentazione. Nel caso in cui si volesse utilizzare solo il microcontrollore, i consumi sarebbero inferiori a quelli misurati. Nel capitolo 6 viene presentata anche una comparazione diretta tra gli incrementi di consumi tra le fasi di idle e di inferenze, andando oltre la semplice analisi dei valori assoluti. Questo permetterà di avere una visione più chiara dei consumi aggiuntivi dovuti all'esecuzione dei modelli.

### 3.2.2 Esecuzione dei modelli su TPU

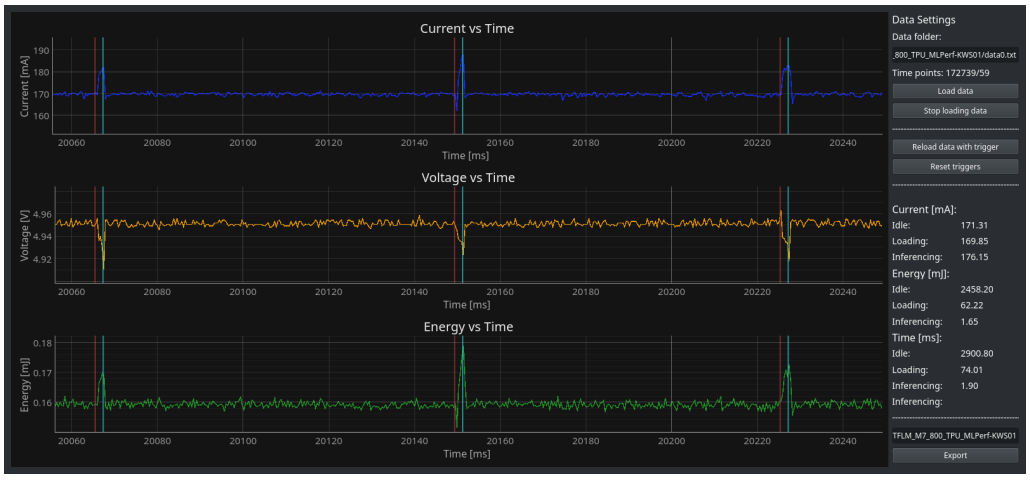
L'esecuzione di inferenza su EdgeTPU richiede la conversione delle reti neurali in un formato specifico. Fortunatamente, i modelli pre-addestrati forniti dal benchmark MLPerf-Tiny, essendo già quantizzati a 8 bit, hanno consentito una compilazione diretta tramite *edgetpu-compiler*, senza la necessità di ulteriori passaggi di pre-elaborazione.




(a) Anomaly Detection Network




(b) Image Classification Network




(c) Keywork Spotting Network


(d) Visual Wakeword Network

Figura 3.10: Risultati dei test di esecuzione dei modelli sulla TPU

Analogamente a quanto accaduto nel test utilizzando il framework CUBE.AI, anche in questo caso si osserva un cambiamento nel rapporto tra il tempo di esecuzione e il tempo di caricamento dati. Questo fenomeno è dovuto all'accelerazione hardware fornita dalla TPU, che permette di eseguire le operazioni di inferenza in tempi molto più brevi rispetto alla CPU. Nei grafici riportati in figura 3.10 si osserva come il tempo di esecuzione sia diventato trascurabile rispetto al tempo di caricamento dei dati, che rappresenta la maggior parte del tempo totale di esecuzione.

| **Modello** | **Corrente Idle (mA)** | **Tempo (ms)** | **Corrente (mA)** | **Avg Power (mW)** | **Energy (mJ)** |
|---|---|---|---|---|---|
| Anomaly Detection | 154.56 | 0.86 | 156.50 | 766.85 | 0.66 |
| Image Classification | 174.48 | 1.50 | 180.68 | 885.33 | 1.30 |
| Keyword Spotting | 171.31 | 1.90 | 176.15 | 863.14 | 1.65 |
| Visual WakeWord | 157.45 | 4.44 | 173.08 | 848.10 | 3.78 |

Tabella 3.6: Risultati dei test di esecuzione dei modelli sulla TPU

Si può notare dai risultati e dai grafici che durante l'esecuzione dei test relativi delle reti più semplici e veloci da eseguire, quali Anomaly Detection e Image Classification, i picchi di corrente assorbita non sono perfettamente allineati con i

segnali di trigger. Questo probabilmente è dovuto alle limitazioni della frequenza di campionamento della scheda di misurazione dei consumi.
Rispetto ai test eseguiti sulla CPU, la corrente assorbita in entrambe le fasi è aumentata, a causa dell'attivazione dell'acceleratore hardware. Considerando solo i due test considerati attendibili, ovvero Keyword Spotting e Visual WakeWord, si osserva un'elevata riduzione sia dei consumi che del tempo di esecuzione rispetto all'esecuzione sulla CPU, nonostante la corrente assorbita sia nello stesso ordine di grandezza.
É possibile quindi affermare che l'utilizzo dell'EdgeTPU permette di ottenere un miglioramento significativo delle prestazioni, riducendo i tempi di esecuzione e i consumi energetici.

# Capitolo 4

# Analisi dei risultati

In questo capitolo verrà fornita una panoramica comparativa delle diverse soluzioni presentate nei capitoli precedenti. L'obiettivo è quello di evidenziare le performance, i punti di forza e le limitazioni di ciascuna soluzione, al fine di fornire una valutazione oggettiva. Nell'appendice A.2 sono riportate le tabelle con tutti i risultati numerici ottenuti durante i test oltre ai grafici relativi ai consumi energetici e ai tempi di esecuzione.

## 4.1 Accuratezza

Nell'analisi delle prestazioni, il primo parametro considerato è l'accuratezza, valutata mediante due metriche distinte: AUC per il modello di Anomaly Detection e accuratezza classica per i restanti modelli.

Coerentemente con la modalità *closed division* del benchmark adottato, i risultati ottenuti non sono utilizzati per confronto con lavori esterni, ma servono a verificare la correttezza dell'implementazione.

La Figura 4.1 mostra i valori di accuratezza raggiunti dalle diverse soluzioni implementate, confermando la corretta implementazione di tutti i modelli in ciascuna configurazione, senza differenze significative tra i valori di accuratezza ottenuti.

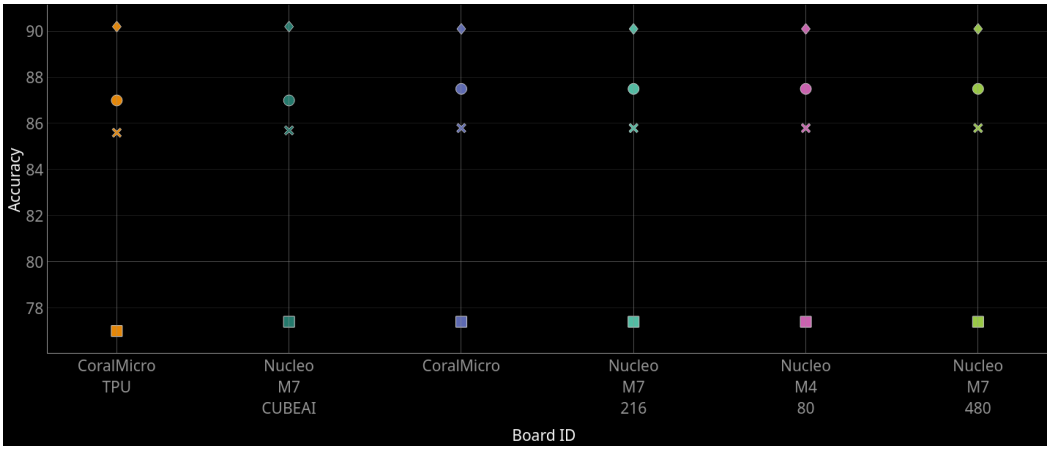



Figura 4.1: Accuratezza delle diverse soluzioni

## 4.2 Latenza

Sebbene una bassa latenza sia generalmente desiderabile, il suo impatto sulle prestazioni complessive del sistema dipende fortemente dal contesto applicativo. In scenari reali, l'ottimizzazione della latenza non si traduce sempre in un miglioramento delle prestazioni globali. Ad esempio, in un sistema embedded in cui l'algoritmo di ML opera in parallelo con l'acquisizione dati, una latenza inferiore al tempo di acquisizione non apporterà benefici tangibili. In questo caso, il fattore limitante sarà la velocità di acquisizione, rendendo superflua un'ulteriore riduzione della latenza di elaborazione.

Di seguito sono riportati i valori di latenza ottenuti con le diverse soluzioni implementate. I quattro test effettuati sono rappresentati per ogni configurazione con lo stesso simbolo.

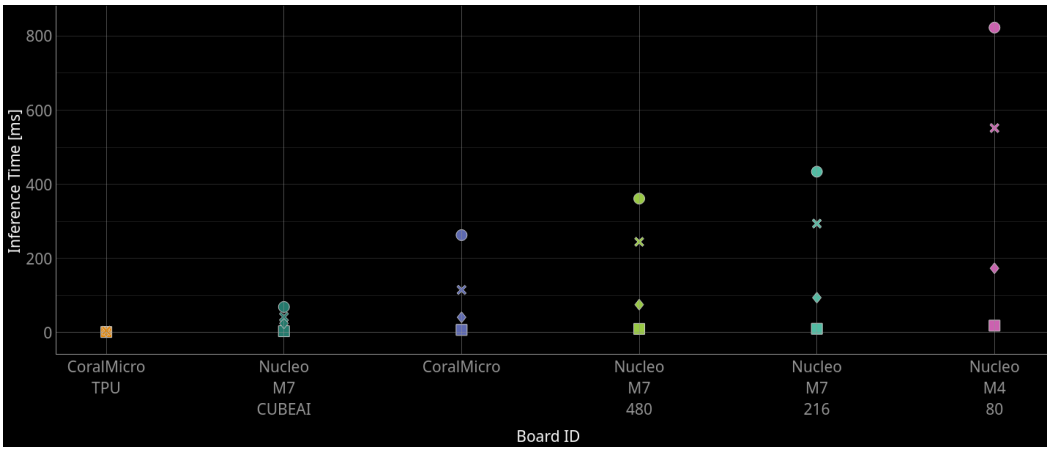



Figura 4.2: Latenza delle diverse soluzioni

L'analisi dei risultati conferma l'attesa correlazione tra latenza, potenza di calcolo e frequenza di clock: all'aumentare di questi ultimi, si osserva generalmente una riduzione della latenza. Tuttavia, l'implementazione sulla scheda Nucleo con la libreria ottimizzata *Cube.AI* rappresenta un'eccezione a questo trend. Nonostante la frequenza di clock inferiore (un quarto rispetto alla Coral Dev Board Micro), la scheda Nucleo con *Cube.AI* raggiunge valori di latenza molto inferiori, con un aumento di prestazioni che dipende dalla rete considerata. Questo risultato evidenzia l'importanza di librerie ottimizzate ad-hoc per l'architettura hardware specifica. Tali librerie, sfruttando in modo efficiente le risorse disponibili, consentono di ottenere prestazioni superiori rispetto al semplice aumento della frequenza di clock.

È tuttavia fondamentale sottolineare che lo sviluppo di librerie ottimizzate richiede una conoscenza approfondita dell'architettura del dispositivo, rendendo questo approccio non sempre percorribile.

Come previsto, le migliori prestazioni in termini di latenza sono state ottenute con l'acceleratore hardware. Oltre a ridurre significativamente la latenza, l'acceleratore consente al microcontrollore di eseguire in parallelo altre operazioni, aumentando l'efficienza complessiva del sistema.

## 4.3 Corrente assorbita

L'analisi della corrente assorbita conferma la correlazione diretta tra consumo energetico e frequenza di funzionamento. Come evidenziato nel grafico in Figura 4.3, un aumento della frequenza di clock si traduce in un incremento proporzionale della corrente assorbita.

È interessante notare che, nonostante le ottimizzazioni implementate nella libreria *Cube.AI*, non si osserva una riduzione dei consumi rispetto all'implementazione con LiteRT. Questo risultato suggerisce che le ottimizzazioni di *Cube.AI*, pur migliorando significativamente la latenza, non impattano in modo altrettanto rilevante sull'efficienza energetica.

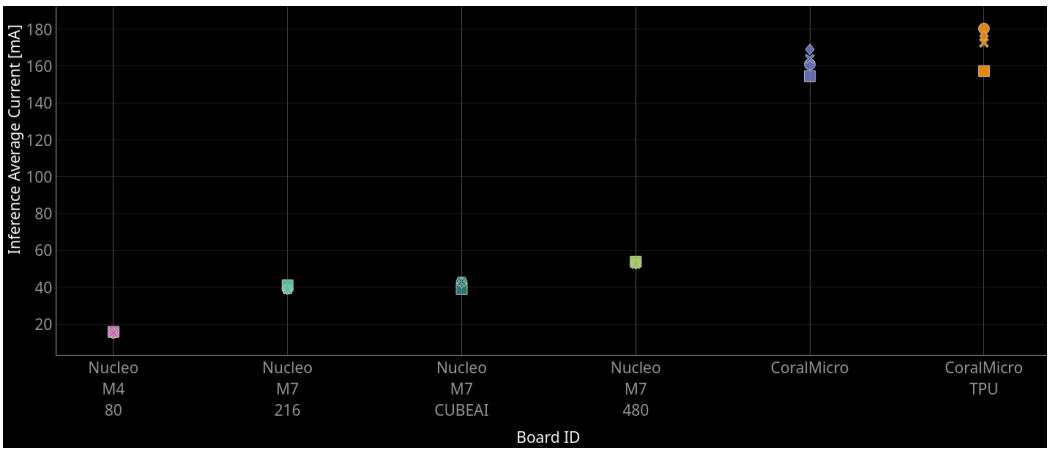



Figura 4.3: Consumi delle diverse soluzioni

L'immagine in questione evidenzia la problematica discussa nel Capitolo 3.2 relativa alla misurazione dei consumi della Coral Dev Board Micro. I valori ottenuti, significativamente superiori alle aspettative, non consentono un confronto diretto con gli altri dispositivi.

Per condurre un'analisi comparativa più accurata, pur non essendo tale metrica utilizzabile per il benchmark, si è calcolato il delta di corrente tra la fase di inattività (idle) e la fase di inferenza (figura 4.4). Questa metodologia permette di isolare il contributo energetico dell'esecuzione dei modelli, escludendo i consumi statici della scheda e delle periferiche.
Osservando il grafico in figura 4.4 si possono formulare le seguenti osservazioni:

- L'implementazione che sfrutta solamente la CPU della Coral Dev Board Micro è quella che presenta il consumo energetico più a causa dell'elevata frequenza di clock.

- La libreria CUBE.AI è caratterizzata da consumi inferiori nella fase di inattività ma un delta di corrente elevato. Questa caratteristica è probabilmente correlata alle strategie di ottimizzazione che migliorano le prestazioni modificando il dataflow dell'algoritmo in modo da qumentare il numero di operazioni al secondo.

- L'implementazione con l'acceleratore hardware mostra un delta di corrente ridotto, grazie all'utilizzo di un'architettura dedicata all'esecuzione di algoritmi di Machine Learning.
- I risultati ottenuti con la Nucleo a 400 MHz sono parzialmente falsati a causa del rumore introdotto dall'elevata frequenza di clock. Confrontando manualmente i risultati si può notare che la differenza di corrente dovrebbe essere superiore a quella ottenuta con la Nucleo a 216 MHz.

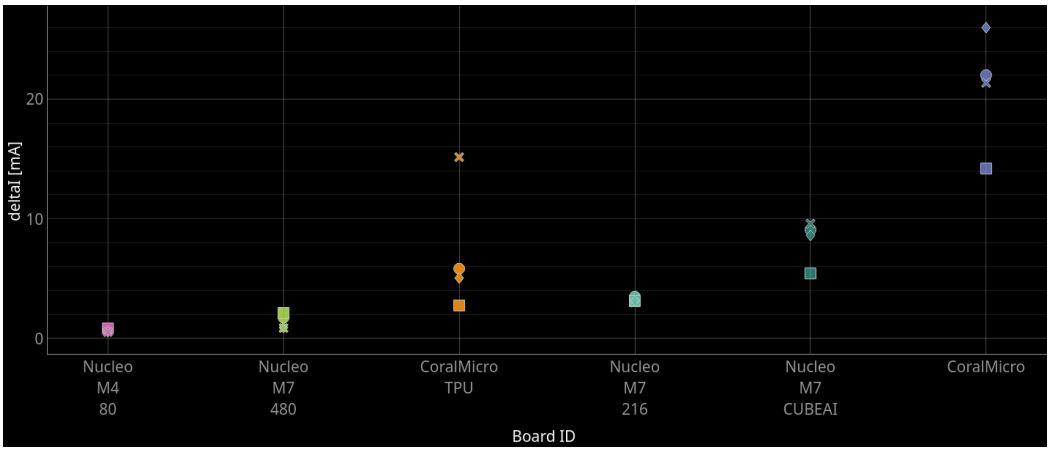



Figura 4.4: Differenza di corrente tra fase di inattività e inferenza

## 4.4 Consumo energetico

Il consumo energeto dipende sia dall'intensità della corrente assorbita che dalla durata della fase di inferenza. In un dispositivo edge alimentato a batteria, il consumo energetico è un parametro chiave per valutare l'autonomia del sistema.

I consumi energetici delle diverse soluzioni sono stati calcolati utilizzando i valori di corrente assorbita presentati nel grafico 4.3

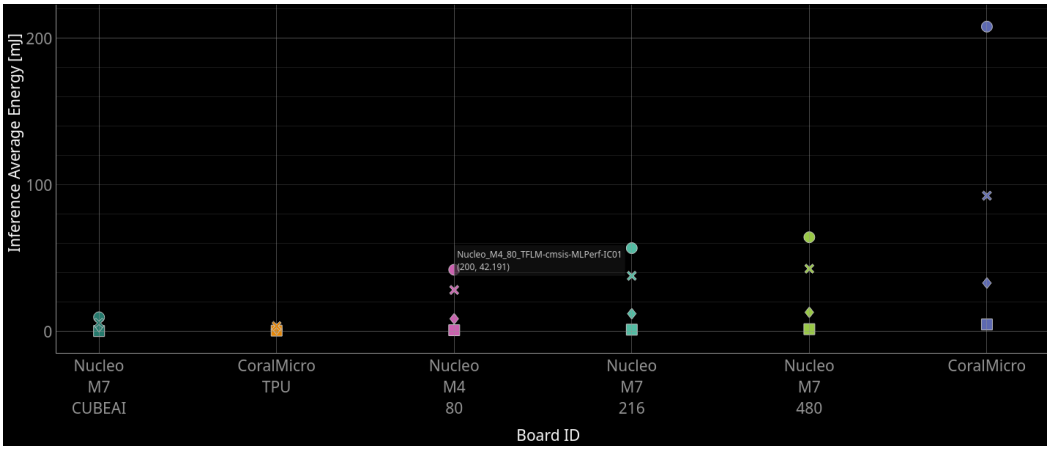



Figura 4.5: Consumo energetico delle diverse soluzioni

Come era possibile prevedere, la soluzione che utilizza il tool *Cube.AI* ha un consumo energetico molto ridotto rispetto alle altre soluzioni. Anche i risultati ottenuti con l'acceleratore hardware sono molto buoni, nonostante il problema riscontrato durante la misurazione dei consumi.

Dagli altri quattro risultati si può dedurre che, nonostante la latenza si riduca all'aumentare della frequenza di clock, l'incremento della corrente assorbita comporta un aumento del consumo energetico.

Analogamente a quanto fatto per la corrente, è stato calcolato l'aumento di energia richiesta per l'esecuzione dell'algoritmo di ML.

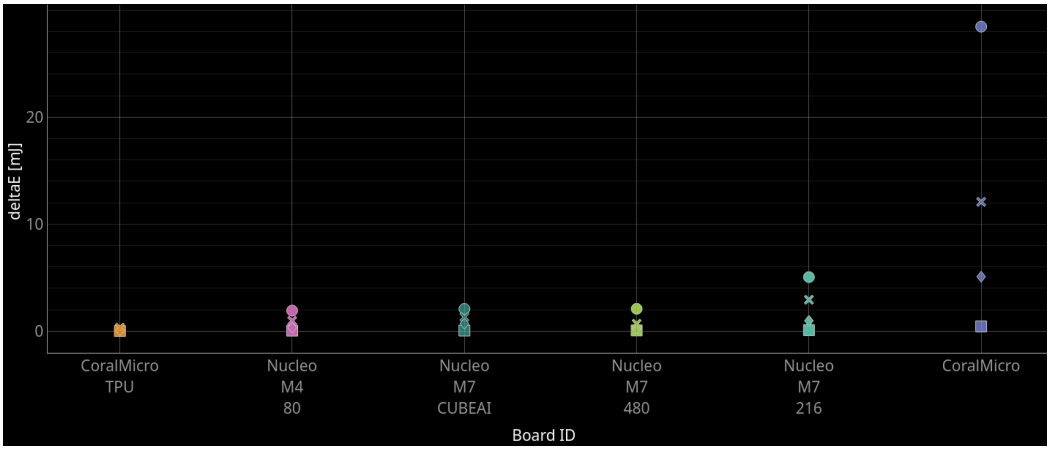



Figura 4.6: Differenza di energia tra fase di inattività e inferenza

Da questo grafico, relativo al benchmark *Image Classification* si può notare che l'energia richiesta per l'esecuzione dell'algoritmo su EdgeTPU è inferiore rispetto alle altre soluzioni, confermando la superiorità dell'acceleratore hardware in termini di efficienza energetica.

## 4.5 Considerazioni finali

Dai risultati ottenuti da questo studio è possibile affermare che le performance che possono essere ottenute da un sistema dipendono molto dalle specifiche dell'hardware utilizzato e dalle librerie di supporto.

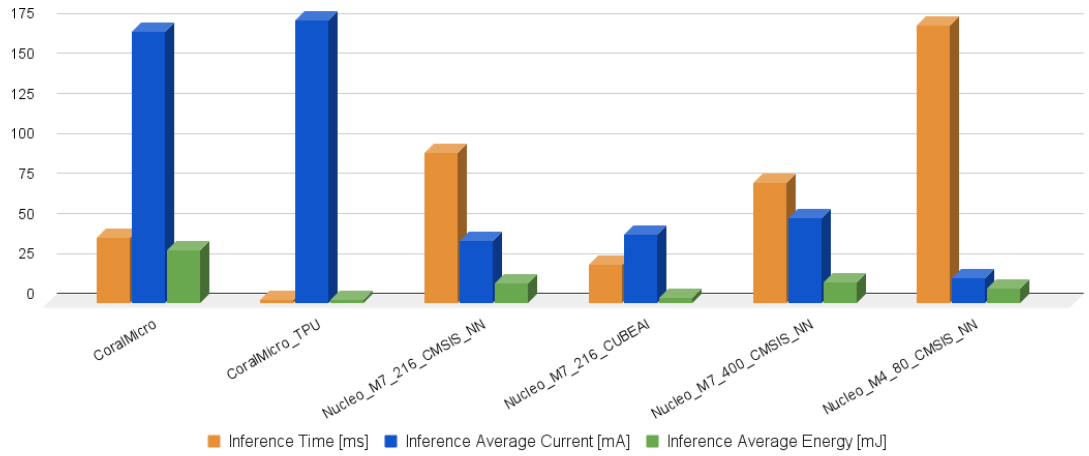



Figura 4.7: Confronto dei risultati per il benchmark KeyWord Spotting

L'esecuzione efficiente di modelli di Deep Learning su microcontrollori richiede un'attenta valutazione del trade-off tra prestazioni e consumo energetico. Librerie standard come LiteRT, pur offrendo portabilità, sono vincolate dalle risorse computazionali limitate del microcontrollore. Un aumento della frequenza di clock può migliorare le prestazioni, ma a discapito di un maggiore consumo energetico. Questo incremento può non essere un problema rillevante in caso di dispositivi alimentati dalla rete elettrica ma risulta problematico in dispositivi alimentati a batteria, dove l'autonomia è un fattore cruciale.

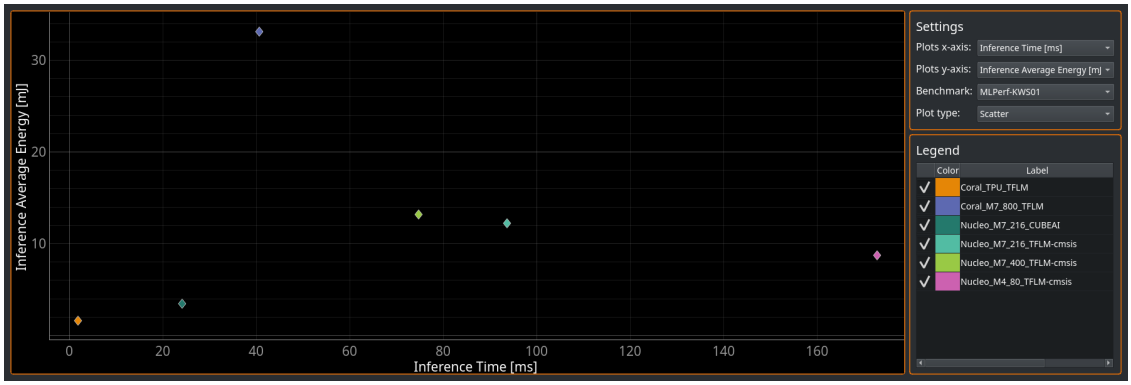



Figura 4.8: Grafico Energia consumata in funzione della latenza per il benchmark KeyWord Spotting

L'utilizzo di librerie ottimizzate come *Cube.AI* consente di ottenere prestazioni elevate mantenendo gli stessi consumi del microcontrollore,ma richiede una cono-

scenza approfondita dell'hardware utilizzato. Al contrario, l'utilizzo di acceleratori hardware consente di ottenere prestazioni elevate con consumi energetici ridotti, ma richiede l'utilizzo di reti neurali quantizzate e aumenta il costo e la camplessità del sistema.

# Capitolo 5

# Acceleratore hardware su FPGA

La fase successiva alla valutazione comparativa delle soluzioni commerciali consiste nello sviluppo e nella verifica di un acceleratore hardware personalizzato. In ambito accademico, l'implementazione di tale progetto si basa comunemente sull'utilizzo di una FPGA (Field Programmable Gate Array) per motivi di convenienza economica. L'adozione di una FPGA offre flessibilità in termini di implementazione, consentendo di realizzare soluzioni general purpose per un'ampia gamma di applicazioni o soluzioni specifiche ottimizzate per casi d'uso particolari e/o con vincoli di risorse. Dal punto di vista didattico, l'utilizzo di una FPGA permette di sperimentare con diverse tecniche di progettazione hardware e di apprendere i concetti di base dell'architettura di un acceleratore hardware. La scelta della tipologia di implementazione richiede un'analisi approfondita dei requisiti prestazionali e delle esigenze applicative, unitamente a una valutazione delle risorse disponibili sulla FPGA.
Il presente lavoro si propone di investigare la fattibilità di un acceleratore hardware implementato sulla FPGA integrata nella scheda VirtLAB[11].

## 5.1 VirtLAB

La VirtLAB è una piattaforma di sviluppo, concepita e realizzata presso il Politecnico di Torino, che fornisce agli studenti un ambiente per la progettazione, implementazione e validazione di progetti di ricerca e didattici. Questo dispositivo è stato progettato come alternativa economica all'insieme di strumenti necessari per lo sviluppo di codice o hardware per i progetti universitari.

La scheda è costituita due sezioni identiche,chiamate *Master* e *User*, che integrano ciascuna un microcontrollore e una FPGA dello stesso tipo, ottimizzando costi e complessità di progettazione.

La sezione *Master* è ideata per avere un funzionamento analogo agli strumenti di laboratorio, con il microcontrollore che implementa un DSO (Digital Storage Oscilloscope) e un generatore di forme d'onda arbitrarie mentre la FPGA, coadiuvata da una memoria HyperRAM, realizza un analizzatore logico e un generatore di pattern digitali.

Inoltre, il microcontrollore Master gestisce la configurazione di entrambe le FPGA, la cui natura volatile (RAM-based) necessita di una fase di programmazione ad ogni power-up. Per evitare di dover caricare manualmente il bitstream ad ogni accensione, questo viene salvato in una memoria Flash dedicata, connessa allo stesso microcontrollore.
La sezione *User* invece è concepita per lo sviluppo e la validazione di soluzioni hardware e firmware.
La comunicazione tra le due sezioni è possibile grazie ad un bus a 32 bit in comune tra i due microcontrollori e le due FPGA. Questo bus, durante l'uso in laboratorio, viene utilizzato principalmente come interfaccia tra software o hardware di controllo (implementato dal docente) e il progetto sviluppato dagli studenti. Il bus inoltre è accessibile esternamente tramite dei pin header per l'interfacciamento con altri dispositivi.

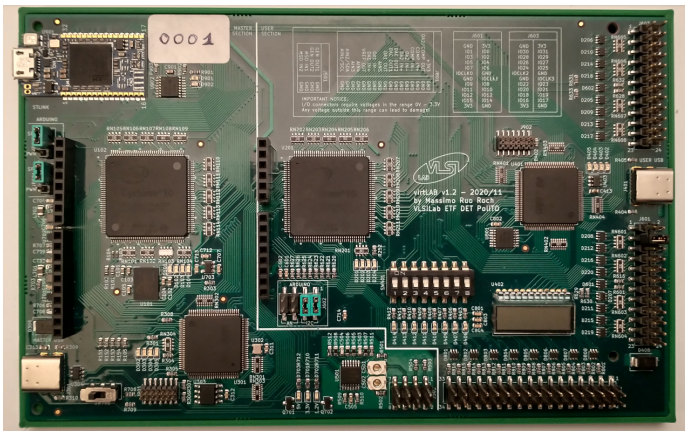



Figura 5.1: VirtLAB

La programmazione dei microcontrollori avviene tramite un modulo STLink con interfaccia USB. Sono poi presenti altre due porte USB, una utilizzata per inviare le configurazioni delle FPGA al microcontrollore Master e un'altra, collegata al microcontrollore User, che permette una comunicazione seriale ad alta velocità tra quest'ultimo e un dispositivo esterno.

Alcune delle specifiche del microcontrollore presente sulla scheda (STM32L4) sono le seguenti:

- ARM Cortex-M4 a 80 MHz;
- 512 kBytes di Memoria Flash;
- 320 kBytes di Memoria RAM;
- SPI, I2C, UART, CAN;
- 14 canali DMA;

Sono presenti altre funzionalità che non sono state riportate in quanto non rilevanti per il progetto.

Per quanto riguarda la FPGA, la VirtLAB è utilizza due FPGA Cyclone 10 CL025, i quali hanno le seguenti specifiche:

- clock massimo di 480 MHz;
- 24,624 kLE;
- 594 kbit di memoria interna (74.25 kByte);
- 66 moltiplicatori a 18x18 bit;
- 4 PLL;

Da considerare che un moltiplicatore da 18x18 bit può anche essere separato in due moltiplicatori 8x8 bit. Per lo scopo di questo progetto, in cui si vuole implementare un acceleratore che esegue reti quantizzate, questa possibilità permette di avere fino a 132 moltiplicatori. Utilizzando il tool di sintesi di Quartus Prime è possibile stimare che l'implementazione di un moltiplicatore 8x8 bit richieda anche 13 LUT e 17 registri (questi valori dipendono dalla configurazione selezionata), tale numero è da tenere in considerazione per il calcolo della memoria richiesta per l'implementazione dell'acceleratore.

## 5.2 Analisi preliminari

Nella progettazione di un acceleratore hardware, un parametro critico è la capacità della memoria interna della FPGA. Tale parametro determina la possibilità di memorizzare interamente i dati necessari per l'inferenza all'interno dell'acceleratore, o la necessità di ricorrere a memoria esterna e alla tecnica del *tiling*.

La tecnica del tiling consiste nella suddivisione dei dati in blocchi di dimensioni ridotte, al fine di limitare la quantità di informazioni da memorizzare on-chip. Questo approccio consente l'utilizzo di dispositivi con risorse di memoria limitate, ma introduce un aumento della complessità del sistema e una potenziale riduzione delle prestazioni dovuta all'utilizzo di memorie esterne con latenza e banda limitate.

Considerando le specifiche riportate in Tabella 2.3.1, si evince che la memoria interna della FPGA non sia sufficiente a contenere l'intero set di pesi e bias di una rete neurale. È quindi necessario valutare se tale memoria sia adeguata per memorizzare i dati relativi all'esecuzione di un singolo layer per volta.

Di seguito sono riportate le dimensioni massime richieste per la memorizzazione di pesi e bias dei layer delle quattro reti neurali di test, considerando due diverse strategie: la prima strategia prevede di mantenere fisse le dimensioni dei blocchi di memoria. In questo caso la dimensione della memoria sarà pari alla somma del del massimo numero di pesi,bias,input e output da memorizzare.

| Model | Input | Pesi | bias | Output | Totale [B] |
|---|---|---|---|---|---|
| Anomaly Detection | 1x640 | 640x128 | 640 | 1x640 | 83840 |
| Image Classification | 1x32x32x16 | 64x3x3x64 | 64 | 1x32x32x16 | 69696 |
| KeyWord Spotting | 1x25x5x64 | 64x1x1x64 | 64 | 1x25x5x64 | 20160 |
| Visual Wake Word | 1x48x48x16 | 256x1x1x256 | 256 | 1x48x48x16 | 139520 |

Tabella 5.1: Memoria in configurazione "fixed-size"

La seconda strategia invece considera dei blocchi di memoria che possono avere dimensione variabile in base al layer eseguito. In questo caso la dimensione della memoria sarà pari alla somma di pesi,bias,input e output del layer con le dimensioni massime.

| Model | Input | Pesi | bias | Output | Totale [B] |
|---|---|---|---|---|---|
| Anomaly Detection | 1x128 | 640x128 | 640 | 1x640 | 83328 |
| Image Classification | 1x32x32x16 | 64x3x3x64 | 64 | 1x32x32x16 | 69696 |
| KeyWord Spotting | 1x25x5x64 | 64x1x1x64 | 64 | 1x25x5x64 | 20160 |
| Visual Wake Word | 1x3x3x256 | 256x1x1x256 | 256 | 1x3x3x256 | 70400 |

Tabella 5.2: Memoria in configurazione "variable-size"

La seconda strategia, pur riducendo in alcuni casi la memoria richiesta, introduce una maggiore complessità a livello di sistema. Pertanto, per la progettazione dell'acceleratore è stato scelto di considerare la prima strategia.

Considerando la capacità di memoria disponibile sulla FPGA e riservando un margine per l'implementazione della logica di controllo, si osserva che solo la rete neurale per il Keyword Spotting potrebbe essere interamente allocata on-chip. Per le reti neurali rimanenti, è necessario l'utilizzo della tecnica del tiling e di una memoria esterna.

Le possibili soluzioni per implementare le reti all'interno della FPGA sono le seguenti:

- Utilizzo di una memoria esterna collegata al bus a 32 bit presente sulla VirtLAB e utilizzo di MCU e FPGA della sezione User;
- Utilizzo il MCU della sezione User e la FPGA delle sezione master sfruttando la memoria HyperRAM ad essa collegata.

Si è optato per la seconda soluzione, in quanto la memoria HyperRAM offre prestazioni superiori in termini di velocità rispetto alla memoria esterna connessa al bus a 32 bit, oltre a non richiedere lo sviluppo di una scheda esterna.

## 5.3 Eyexam

La progettazione di un acceleratore per reti neurali su FPGA richiede la definizione di un'architettura hardware in grado di eseguire le operazioni di inferenza in modo efficiente. Un'architettura ampiamente utilizzata per questo scopo è il *Systolic Array*[[12]][[13]] (figura 5.2), una struttura composta da una matrice di elementi di calcolo (PE), capace di eseguire operazioni in parallelo.

Il nome deriva dall'analogia con il sistema circolatorio, in cui i dati fluiscono attraverso la matrice in modo sincrono e ritmico, elaborati dai PE in maniera analoga al flusso sanguigno nel corpo umano. Questa architettura si presta particolarmente all'implementazione di reti neurali, consentendo l'esecuzione parallela di operazioni su più dati e riducendo il tempo di inferenza.

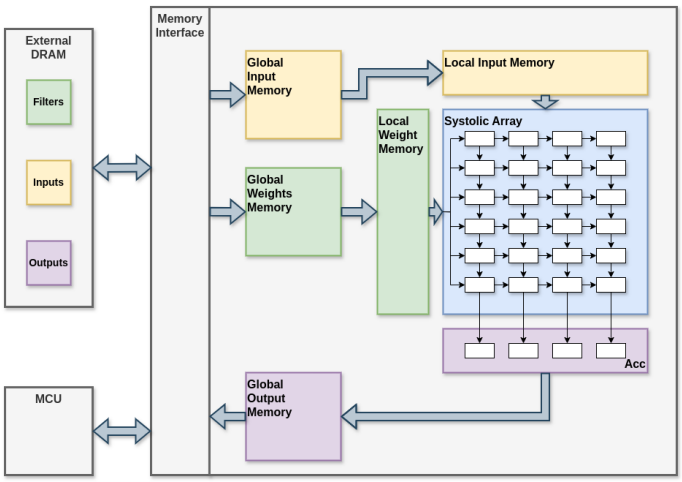



Figura 5.2: Acceleratore basato su "Systolic Array"

Il cuore dell'acceleratore è costituito dalla matrice di Processing Element che forma il Systolic Array, responsabile dell'esecuzione delle operazioni matematiche necessarie per l'inferenza della rete neurale. Ciascun PE è un'unità di elaborazione in grado di eseguire operazioni di moltiplicazione e accumulo (MAC) su due operandi. Il flusso di dati all'interno della matrice segue uno schema preciso: dopo ogni operazione MAC, i dati in ingresso e il risultato parziale vengono trasferiti sincronicamente al PE adiacente per l'esecuzione dell'operazione successiva.

Il processo di elaborazione inizia con il caricamento graduale di input e pesi dalla memoria esterna alla memoria globale interna. La logica di controllo dell'acceleratore organizza questi dati in set ordinati che vengono poi trasferiti alle rispettive memorie locali. Da qui, i dati vengono inviati sincronicamente ai PE per l'esecuzione delle operazioni MAC.

Il flusso dei risultati invece segue un percorso inverso: i risultati parziali generati dai PE vengono accumulati nella memoria locale. Una volta completato il calcolo, gli output finali vengono trasferiti alla memoria globale e infine alla memoria esterna. Per progettare un acceleratore basato su Systolic Array è necessario definire i seguenti parametri:

- **Dimensione della matrice**: da intendersi sia come numero di PE che come forma. La dimensione della matrice determina il numero di operazioni che possono essere eseguite in parallelo. Una matrice più grande permette di eseguire più operazioni in parallelo, ma richiede una quantità maggiore di risorse.

- **Tipologia di dataflow**: a seconda del tipo di dataflow utilizzato cambia il funzionamento della matrice e le strutture, sia di controllo che di memorizzazione necessarie.

- **Dimensione delle memorie**: l'architettura del Systolic Array dipende da diversi blocchi di memoria, come la memoria dei pesi, la memoria degli input e la memoria dei bias. La dimensione di queste memorie determina la quantità di dati che possono essere memorizzati e quindi elaborati in parallelo.

Per valutare questi parametri è stato scelto di utilizzare una procedura molto simile a quella utilizzata dal framework *Eyexam* descritto nel paper *Eyeriss v2*[14]. L'obiettivo originale di Eyexam è quello di valutare le prestazioni di un'architettura hardware utilizzando un processo a 7 step, partendo da un caso ideale fino a raggiungere l'implementazione reale.
Eyexam, per valutare le prestazioni di un'architettura, concentra l'attenzione su due fattori principali: il numero di PE utilizzati rispetto a quelli presenti e per quanti cicli i PE siano utilizzati per eseguire le operazioni.

## 5.4 Procedura

La metodologia proposta in *Eyeriss v2* presenta una generalizzazione intrinseca che ne consente l'applicazione a reti neurali operanti su tensori di dimensione arbitraria. Tuttavia, in questo elaborato, l'analisi preliminare si è concentrata esclusivamente su architetture neurali che elaborano tensori unidimensionali (1D). Si prevede che in studi futuri la procedura verrà estesa a reti neurali che operano su tensori di dimensionalità superiore.
Per quanto riguarda i dataflow, Eyexam considera quattro tipologie di dataflow:

- **Weight Stationary (WS)**: in questo schema, i pesi della rete neurale sono mantenuti costanti all'interno della memoria interna dei *Processing Elements* (PE), mentre gli input vengono inviati in modo sequenziale. Questo tipo di dataflow è particolarmente adatto per reti con pochi pesi e molti input. I risultati parziali sono accumulati in una memoria esterna ai PE.

- **Input Stationary (IS)**: in questo caso gli input della rete neurale sono salvati all'interno dei PE, mentre i pesi vengono inviati in modo sequenziale. Questo tipo di dataflow è particolarmente adatto per reti con pochi input e molti pesi. Anche in questo caso i risultati parziali sono accumulati in una memoria esterna ai PE.

- **Output Stationary (OS)**: ad ogni PE viene assegnata l'elaborazione di un output. Input e pesi vengono inviati in modo sequenziale affinché ogni PE possa eseguire l'operazione richiesta. Non richiede una memoria esterna per i risultati parziali in quanto questi vengono accumulati all'interno dei PE. Necessita di una logica più complessa per la creazione delle sequenze di input e pesi.

- **Row Stationary (RS)**: in questa strategia, i tensori di input e i tensori dei pesi vengono partizionati in righe, consentendo di decomporre un'operazione N-dimensionale in una serie di operazioni 1D eseguibili in parallelo.

In questo progetto è stato scelto di prendere in considerazione solamente i primi tre tipi di dataflow, in quanto sono quelli più utilizzati e più intuitivi da implementare.

Per la descrizione di questi dataflow sarà utilizzata la notazione a "loop-nest", in cui le iterazioni eseguite in parallelo sono indicate con il termine *parallel-for*. Il numero di loop presenti nella descrizione dipende dalle caratteristiche hardware dell'acceleratore. Nel caso descritto in questo capitolo, possiamo considerare quattro serie di iterazioni che, dall'interno verso l'esterno, sono:

- **Local Memory Loop**: identificano le iterazioni necessarie per spostare tutti i dati presenti nella memoria locale ai PE. Una dimensione maggiore per la memoria locale relativa agli input comporta un aumento dell'intervallo di tempo durante il quale gli stessi pesi vengono riutilizzati e viceversa;

- **PE (Parallel)Loop**: esecuzione in parallelo delle operazioni all'interno dei PE;

- **Global Memory Loop**: sono i cicli necessari per spostare blocchi di dati tra la memoria locale e la memoria globale e viceversa;

- **External Memory Loop**: sono le iterazioni durante le quali i dati vengono spostati tra la memoria esterna e la memoria globale.

Lo pseudocodice per l'implementazione dei tre dataflow considerati è il seguente:

### Input Stationary Dataflow

```
int inputs[E];
int weights[R];
int outputs[E-R+1]

for (e3 = 0; e3 < E3; e3++) {
 for (r3 = 0; r3 < R3; r3++) {
  for (e2 = 0; e2 < E2; e2++) {
   for (r2 = 0; r2 < R2; r2++) {
    parallel-for (e1 = 0; e1 < E1; e1++) {
     parallel-for (r1 = 0; r1 < R1; r1++) {
      for (e0 = 0; e0 < E0; e0++) {
       for (r0 = 0; r0 < R0; r0++) {
        inputIdx = e0 + e1*E0 + e2*E1*E0 + e3*E2*E1*E0;
        weightIdx = r0 + r1*R0 + r2*R1*R0 + r3*R2*R1*R0
        outputIdx = inputIdx - weightIdx;
        outputs[outputIdx] += inputs[inputIdx] * weights[weightIdx];
}}}}}}}}
```

### Weight Stationary Dataflow

```
int inputs[E];
int weights[R];
int outputs[E-R+1]

for (r3 = 0; r3 < R3; r3++) {
 for (e3 = 0; e3 < E3; e3++) {
  for (r2 = 0; r2 < R2; r2++) {
   for (e2 = 0; e2 < E2; e2++) {
    parallel-for (r1 = 0; r1 < R1; r1++) {
     parallel-for (e1 = 0; e1 < E1; e1++) {
      for (r0 = 0; r0 < R0; r0++) {
       for (e0 = 0; e0 < E0; e0++) {
        inputIdx = e0 + e1*E0 + e2*E1*E0 + e3*E2*E1*E0;
        weightIdx = r0 + r1*R0 + r2*R1*R0 + r3*R2*R1*R0
        outputIdx = inputIdx - weightIdx;
        outputs[outputIdx] += inputs[inputIdx] * weights[weightIdx];
        }}}}}}}}
```

### Output Stationary Dataflow

```
int inputs[H];
int weights[R];
int outputs[E];

for (e3 = 0; e3 < E3; e3++) {
 for (r3 = 0; r3 < R3; r3++) {
  for (e2 = 0; e2 < E2; e2++) {
   for (r2 = 0; r2 < R2; r2++) {
    parallel-for (e1 = 0; e1 < E1; e1++) {
     parallel-for (r1 = 0; r1 < R1; r1++) {
      for (e0 = 0; e0 < E0; e0++) {
       for (r0 = 0; r0 < R0; r0++) {
        weightIdx = r0 + r1*R0 + r2*R1*R0 + r3*R2*R1*R0
        outputIdx = e0 + e1*E0 + e2*E1*E0 + e3*E2*E1*E0
        inputIdx = outputIdx + weightIdx
        outputs[outputIdx] += inputs[inputIdx] * weights[weightIdx];
}}}}}}}}
```

È importante osservare che nelle configurazioni *Weight Stationary* e *Input Stationary*, l'indice $E$ è impiegato per denotare gli elementi del tensore di input, mentre nella configurazione *Output Stationary* lo stesso indice $E$ viene utilizzato per riferirsi agli elementi del tensore di output. Tale convenzione è stata adottata per garantire una maggiore chiarezza e leggibilità nella struttura del codice, semplificando l'implementazione dei cicli iterativi che gestiscono l'elaborazione dei dati. In particolare, questa scelta consente di mantenere una struttura del codice omogenea per i diversi dataflow.
Dai dataflow riportati è possibile vedere come, in alcuni casi, gli indici utilizzati per accedere ai dati siano possano essere "out-of-bounds". In questo caso l'operazione non viene effettivamente eseguita dai PE. Questo tempo di esecuzione "sprecato" è uno dei due fattori che Eyexam considera per valutare le prestazioni di un'architettura.
Nel prossimo paragrafo sarà descritta la procedura utilizzata, sia in modo teorico, sia utilizzando degli esempi pratici. Per i calcoli pratici saranno utilizzati i dati presenti nella tabella 5.3.

| **Network Informations** | | | |
|---|---|---|---|
| **Layer ID** | **Inputs** | **Weights** | **Outputs** |
| 0 | 1x640 | 1x10 | 1x640 |
| **Memory Informations** | | | |
| **Memory Type** | **Size** | **Bandwidth** | **Buswidth** |
| External | 8Mb | 50Mbps | 1B |
| On-Chip | 64kB | 200Mbps | 1B |
| **Processing Elements** | | | |
| **Number** | **Matrix Shape** | **Pipeline** | **Frequency** |
| 132 | 16x8 | 13 | 200MHz |

Tabella 5.3: Specifiche utilizzate per gli esempi

### 5.4.1 Step 1 - Layer Shape and Size

La fase iniziale della procedura prevede l'analisi di uno scenario ideale in cui l'acceleratore hardware è in grado di elaborare l'intero layer della rete neurale in un unica iterazione.Per raggiungere tale obiettivo, è necessario dimensionare l'acceleratore in modo che il numero di Processing Element sia pari al numero di operazioni matematiche richieste dal layer.
Formalmente, questa condizione si traduce nell'impostazione dei parametri di parallelismo *R1* ed *E1* pari, rispettivamente, alla dimensione R del tensore dei pesi e alla dimensione E del tensore di input. Gli altri parametri sono posti uguali a 1.
Per estendere l'analisi ad un'intera rete neurale, è fondamentale determinare le di-

mensioni massime dei tensori di input e output tra tutti i layer che compongono il modello. Questo permette di dimensionare l'acceleratore in modo da garantire la capacità di elaborare tutti i layer presenti.

Il risultato dell'esecuzione di questo step porta ad avere i seguenti parametri:

R0 = 1, R1 = 10, R2 = R3 = 1
E0 = 1, E1 = 640, E2 = E3 = 1

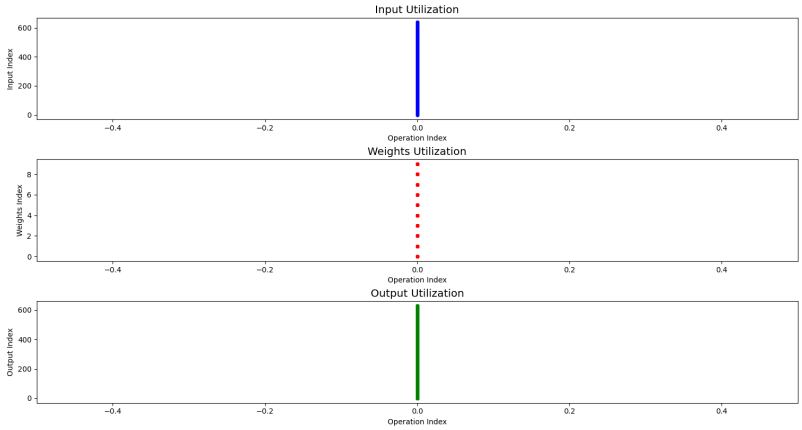



Figura 5.3: Utilizzo dei dati nella configurazione del primo step

Nella figura 5.3 è possibile vedere come i dati vengano utilizzati in questo step. In questo caso, come previsto, tutti i dati vengono utilizzati in un singolo ciclo di esecuzione.

### 5.4.2 Step 2 - Dataflow

Nella seconda fase della procedura, la scelta del dataflow impone dei vincoli sui valori dei parametri di parallelismo *R1* ed *E1*, determinando quale tensore (pesi o input) debba essere processato sequenzialmente e quale in parallelo.

Considerando il dataflow Weight Stationary, i pesi sono mantenuti stazionari all'interno delle Processing Element, mentre gli input vengono inviati sequenzialmente. In accordo con la procedura, il valore di *R1* deve consentire l'utilizzo in parallelo di tutti i pesi, mentre gli input vengono processati uno alla volta. Pertanto, il valore di *R1* rimarrà uguale a R, come nello step precedente, mentre il valore di *E1* sarà pari a 1. Dato che, per garantire che tutti gli input vengano elaborati, il prodotto di tutti i parametri $E^*$ deve essere uguale al numero di input, il valore di *E0* sarà posto uguale a E.

Weight Stationary Dataflow:
R0 = 1, R1 = 10, R2 = 1, R3 = 1
E0 = 640, E1 = 1, E2 = 1, E3 = 1

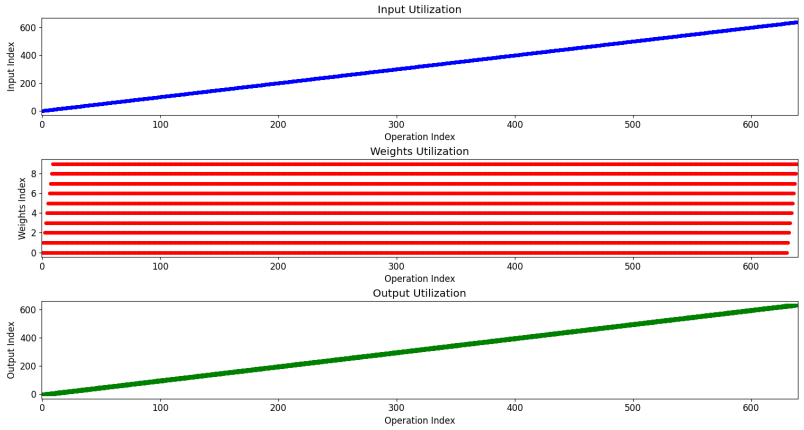



Figura 5.4: Utilizzo dei dati nella configurazione del secondo step

Anche in questo caso il grafico conferma che il tensore dei pesi viene mantenuto costante all'interno dei PE per tutta la durata dell'esecuzione, mentre gli input vengono processati uno alla volta.

### 5.4.3 Step 3 - Number of PEs

In questa fase, i valori di *R1* ed *E1* vengono opportunamente dimensionati per massimizzare il numero di Processing Element istanziabili, tenendo conto delle risorse hardware disponibili (nel caso dell'esempio riportato, 132).
Per limitare il numero di PE al valore massimo consentito, è necessario ridurre il grado di parallelismo nell'esecuzione dei calcoli. Facendo riferimento al dataflow *Weight Stationary* come esempio, il valore assegnato a *R1* viene vincolato al numero di PE disponibili. Di conseguenza, il valore di *R0* viene calcolato come il rapporto tra il numero totale di operazioni da eseguire e il numero di PE, arrotondato per eccesso (`E0 = ceil(E/E1)`). Questa operazione di "arrotondamento per eccesso" garantisce che tutte le operazioni vengano effettivamente eseguite. Lo svantaggio di questa procedura è che, in alcuni casi, il numero di operazioni da eseguire non sarà un multiplo del numero di PE, e di conseguenza in alcuni cicli di esecuzione parte dei PE non sarà utilizzata.
Si ottengono i seguenti parametri:

Weight Stationary Dataflow:
R0 = 1, R1 = 132, R2 = 1, R3 = 1
E0 = E = 640, E1 = 1, E2 = 1, E3 = 1

Utilizzando i parametri ottenuti è possibile stimare il numero di operazioni eseguite e, in particolare, di quantificare quante di queste in realtà non siano utili per il calcolo dei risultati. Se la struttura dei PE lo permette, è possibile disabilitare i Processing Elements coinvolti in queste operazioni inutili, riducendo il consumo di risorse.

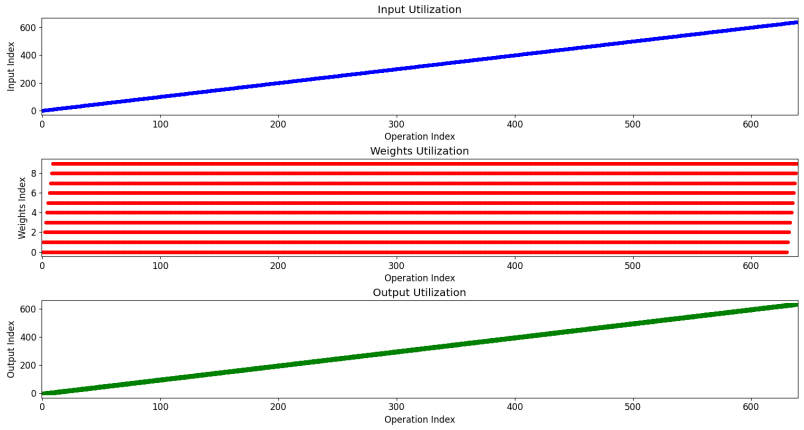



Figura 5.5: Utilizzo dei dati nella configurazione del terzo step

La figura non mostra differenze rispetto al grafico dello step precedente, in quanto il numero di PE istanziati è maggiore del numero di pesi presente nel tensore.

### 5.4.4 Step 4 - Physical dimensions of the PE array

Nello step 4, l'algoritmo di ottimizzazione introduce ulteriori vincoli imposti dalla configurazione fisica della matrice di Processing Element (PE). In particolare, i parametri *R1* ed *E1* vengono dimensionati in modo da ottenere una matrice di PE con una forma specifica.A parità di numero di PE, privilegiare una dimensione della matrice rispetto all'altra influenza il grado di parallelismo nell'elaborazione di uno specifico tensore.
Ad esempio, nel caso del dataflow Weight Stationary, dove la prima dimensione della matrice è correlata al numero di pesi processati in parallelo, un aumento di questa dimensione implica un maggiore utilizzo simultaneo dei pesi.
Tuttavia, questa scelta progettuale può comportare una riduzione del numero effettivo di PE istanziabili rispetto al massimo teorico. Inoltre, può aumentare il numero di cicli di clock inutilizzati, poiché la probabilità che il numero di operazioni da eseguire non sia un multiplo intero del numero di PE risulta incrementata. Come nel caso precedente, i parametri *R0* ed *E0* vengono calcolati in modo da garantire che tutte le operazioni vengano effettivamente eseguite.

I parametri ottenuti in questo step sono i seguenti:

Weight Stationary Dataflow:
R0 = 1, R1 = 16, R2 = 1, R3 = 1
E0 = 80, E1 = 8, E2 = 1, E3 = 1

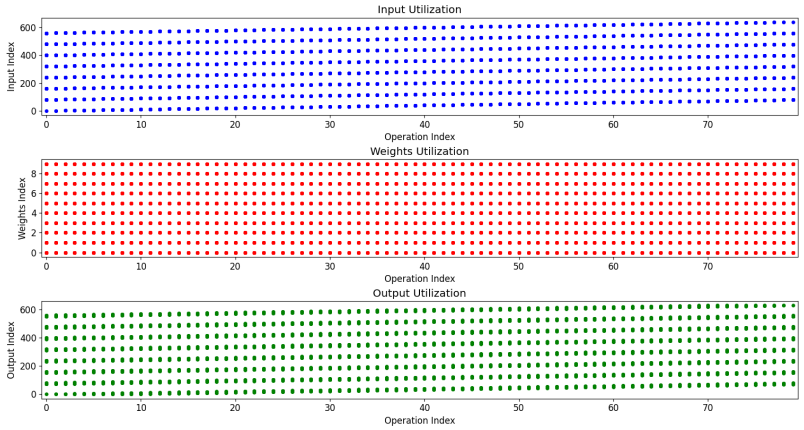



Figura 5.6: Utilizzo dei dati nella configurazione del quarto step

### 5.4.5 Step 5,6 - Storage Capacity and Data Bandwidth

A differenza della procedura originale, in cui i vincoli dimensionali e di banda delle memorie vengono trattati separatamente, questa procedura descritta in questo elaborato adotta una strategia più complessa, che considera entrambi i parametri contemporaneamente per ottenere una soluzione ottimale. Tale approccio consente di adattare dinamicamente le dimensioni dei blocchi di memoria interni per rispettare i limiti di banda, eliminando la necessità di introdurre tempi di attesa per il trasferimento dati tra le memorie. Si evitano così potenziali colli di bottiglia e si massimizza l'efficienza del sistema.
L'architettura di riferimento per l'acceleratore è quella illustrata in Figura 5.2, in cui input, pesi e output condividono la stessa memoria esterna. Questa configurazione semplifica la gestione dei dati e riduce la complessità del sistema.

Per prima cosa viene riportata un'analisi approfonditaa delle formule impiegate per determinare la quantità di memoria richiesta per l'implementazione dell'acceleratore hardware.

- **Memoria dei PE**
  Ciascun Processing Element è composto da un moltiplicatore e un accumulatore, implementati sfruttando i blocchi DSP (Digital Signal Processing block) disponibili all'interno dell'FPGA.

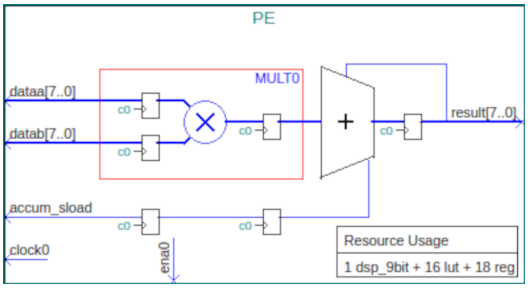



Figura 5.7: Struttura di un Processing Element

Ogni PE include anche una serie di registri per la memorizzazione temporanea dei dati e l'eventuale implementazione di pipeline per ottimizzare il flusso delle operazioni. Il numero di registri necessari per la realizzazione di un PE è specificato nel tool di sintesi di Quartus Prime (figura 5.7). Oltre a questi è utile considerare la presenza di un doppio buffer per la memorizzazione dei dati in ingresso, in modo da non dover attendere la scrittura dei dati in memoria per poter eseguire le operazioni. Il numero di registri necessari per la realizzazione di un singolo PE è specificato dal tool di sintesi Quartus Prime (figura 5.7). A questi registri è di norma aggiunto un doppio buffer per la memorizzazione dei dati, il quale consente di salvare nuovi dati in memoria senza attendere che i dati attualmente in uso siano stati processati.

La memoria totale dalla matrice di PE si ottiene moltiplicando la memoria di un singolo PE per il numero totale di PE presenti nella matrice. Formalmente, possiamo esprimere questa relazione come:

$$PEs_{mem} = R1 * E1 * PE_{mem} + 2 * R1 * E1 \tag{5.1}$$

- **Memoria locale**
  La memoria locale rappresenta il primo livello di memoria che interagisce direttamente con la matrice di Processing Element (PE). La sua funzione principale è quella di garantire un flusso costante di dati ai PE e di memorizzare i risultati parziali delle operazioni.
  Nel caso dei dataflow *Weight Stationary* e *Input Stationary*, la memoria per i risultati perziali costituisce un blocco separato, mentre nel caso del dataflow *Output Stationary* questa sarà integrata direttamente dai PE, in quanto ognuno di essi genera un risultato distinto.
  La dimensione della memoria allocata per gli input dipende dal grado di parallelismo con cui vengono processati dai PE e dal numero di operazioni eseguite sequenzialmente. Possiamo esprimerla come:

$$LocalInput_{mem} = E0 * E1 \tag{5.2}$$

Analogamente, la memoria per i pesi si calcola con la stessa logica, variando solo i parametri specifici:

$$LocalOutput_{mem} = R0 * R1 \tag{5.3}$$

Il dimensionamento della memoria per i risultati parziali richiede un'analisi più approfondita. Questa memoria deve essere in grado di contenere tutti i risultati intermedi generati dai PE. Il numero di tali risultati è direttamente proporzionale al numero di PE nella matrice: più PE sono presenti, maggiore sarà il numero di risultati parziali generati. Inoltre, bisogna considerare il numero di cicli necessari per completare le operazioni e l'eventuale elaborazione di più set di risultati in modo alternato.
La dimensione della memoria sarà quindi stimata considerando il numero risultati parziali presenti in tutti i set che devono essere memorizzati contemporaneamente in memoria.

In numero di risultati parziali presenti in un set e il numero di cicli necessari per iniziare il calcolo di un nuovo set possono essere calcolati come segue:

$$setSize = E0 * E1 \tag{5.4}$$

$$newSetCycles = R0 * R1 \tag{5.5}$$

Il tempo di elaborazione di un risultato dipende dalla dimensione dei tensori e dai parametri dell'acceleratore. È fondamentale stimare il numero di cicli necessari affinché tutti i pesi siano moltiplicati per i corrispondenti input. Questo valore aumenta al diminuire del parallelismo e all'aumentare della separazione dei dati tra le memorie. Inoltre, il dataflow influenza l'ordine di utilizzo dei dati e quindi il tempo di elaborazione.

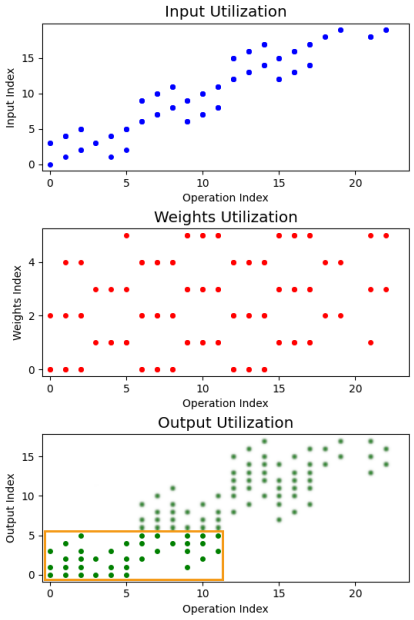



Figura 5.8:
R0=1, R1=3, R2=2, R3=1
E0=3, E1=2, E2=4, E3=1

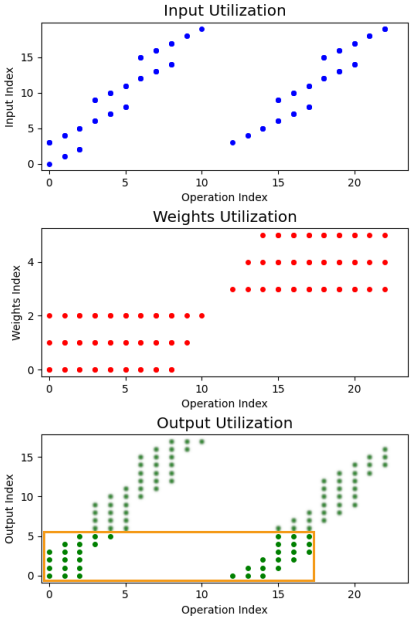



Figura 5.9:
R0=1, R1=3, R2=2, R3=1
E0=3, E1=2, E2=4, E3=1

Come si può vedere dalle figure 5.8 e 5.9, il numero di cicli necessari affinchè un set di dati di dimensione *E0*E1* venga completamente processato dipende dai parametri che definiscono il flusso dei dati.

Combinando queste informazioni, possiamo calcolare la dimensione della memoria per i risultati parziali:

$$setConcurrent = \frac{execCycles}{newSetCycles} \tag{5.6}$$

$$LocalOutput_{mem} = setSize * setConcurrent \tag{5.7}$$

Infine, nel dimensionamento dei blocchi di memoria, è essenziale prevedere un doppio buffer. Questo meccanismo consente di scrivere un nuovo set di dati in memoria senza attendere che i PE abbiano terminato di elaborare il set precedente,mascherando i tempi di trasferimento dati con il tempo di esecuzione delle operazioni.

- **Memoria globale**
  In uno scenario ideale, la sua capacità dovrebbe essere sufficiente a contenere tutti i dati necessari all'esecuzione di un intero layer della rete neurale. Se ciò non è possibile, questa funge da buffer intermedio tra la memoria esterna e la memoria locale. Analogamente alla memoria locale, anche la memoria globale è organizzata in tre blocchi distinti, dedicati rispettivamente a pesi, input e

output.
Come accennato in prercedenza, i parametri *R3* ed *E3* rappresentano il rapporto tra la dimensione dei dati contenuti nei tensori e la dimensione della memoria globale. Tuttavia, per calcolare la dimensione effettiva della memoria, è preferibile utilizzare i parametri *E0, E1, E2* e *R0, R1, R2*, in quanto questi sono stati ricavati utilizzando degli arrotondamenti per eccesso.

$$GlobalInput_{mem} = E0 * E1 * E2 \tag{5.8}$$

$$GlobalWeight_{mem} = R0 * R1 * R2 \tag{5.9}$$

Per quanto riguarda la memoria per contenere gli output, la formula cambia in base al dataflow utilizzato.

$$\begin{aligned} GlobalOutput_{mem} &= E0 * E1 * E2 && \text{se OS} \\ &= (E0 * E1 * E2) + (R0 * R1 * R2) - 1 && \text{se WS o IS} \end{aligned} \tag{5.10}$$

- **Memoria esterna**
  La dimensione della memoria esterna non è direttamente correlata ai parametri dell'acceleratore, ma svolge un ruolo cruciale nel determinare la fattibilità dell'implementazione. Infatti, la memoria esterna deve avere una capacità sufficiente a contenere tutti i dati necessari per l'esecuzione di un layer della rete neurale.
  Esistono due approcci per stimare lo spazio di memoria necessario:
  - **Calcolo diretto delle dimensioni dei tensori:** Questo metodo, semplice e intuitivo, fornisce la dimensione minima indispensabile per la memoria esterna. Tuttavia, non tiene conto del fatto che i trasferimenti di dati tra le memorie avvengono in blocchi di dimensione fissa, e ciò richiede una logica di gestione dei dati più complessa.
  - **Utilizzo dei parametri *E** e *R*:** Questo metodo, basato sui parametri ottenuti nelle fasi precedenti, calcola lo spazio necessario per memorizzare i dati presupponendo blocchi di dimensione fissa. Pur fornendo una stima più conservativa (e quindi una dimensione maggiore) rispetto al metodo precedente, semplifica l'implementazione dell'acceleratore, allineando i requisiti di memoria con le modalità di trasferimento dati.

  Poichè questa procedura è stata progettata per essere utilizzata con reti neurali formate da più layer, è necessario utilizzare il metodo basato sui parametri $E*$ e $R*$ in quanto è possibile che i tensori di dimensioni maggiori appartengano a layer diversi da quello attualmente in esame.

$$ExternalInput_{mem} = E0 * E1 * E2 * E3 \tag{5.11}$$

$$ExternalWeight_{mem} = R0 * R1 * R2 * R3 \tag{5.12}$$

$$\begin{aligned} ExternalOutput_{mem} &= E0 * E1 * E2 * E3 && \text{se OS} \\ &= (E0 * E1 * E2 * E3) \\ &+ (R0 * R1 * R2 * R3) - 1 && \text{se WS o IS} \end{aligned} \tag{5.13}$$

Segue ora l'analisi delle formule utilizzate per determinare la banda necessaria al trasferimento dei dati tra le diverse memorie del sistema. In questo caso, non è necessario definire un valore di banda specifico per ogni singola memoria.Infatti, sebbene varino sia il volume di dati trasferiti che la frequenza di trasferimento, questi due fattori sono inversamente proporzionali. Di conseguenza, il loro effetto sulla banda si compensa, mantenendo costante il valore finale della banda richiesta.

- **Input**
  La banda necessaria per il trasferimento degli input è determinata dalla modalità e dalla frequenza con cui questi vengono riutilizzati all'interno dell'acceleratore. Tali fattori sono strettamente correlati al dataflow implementato. Nel caso del dataflow *Input Stationary* un numero pari ad *E1* di input vengono mantenuti stazionari all'interno delle Processing Element, mentre i pesi vengono aggiornati ad ogni ciclo di clock. In questo scenario specifico, la banda richiesta per il trasferimento degli input può essere calcolata come segue:

  $$Input_{BW} = E1/(R0 * R2 * T_{cycle}) \tag{5.14}$$

  Nel caso del dataflow *Weight Stationary* si verifica la condizione opposta, i pesi rimangono stazionari all'interno dei PE mentre gli input vengono aggiornati ad ogni ciclo. In questo caso la banda può essere calcolata come:

  $$Input_{BW} = E1/(R0 * T_{cycle}) \tag{5.15}$$

  Infine, nel caso del dataflow *Output Stationary* ad ogni ciclo vengono inviati sia nuovi input che nuovi pesi. La banda necessaria per il trasferimento degli input può essere calcolata come:

  $$Input_{BW} = E1/(R0 * R2 * T_{cycle}) + (R1 - 1)/(E0 * T_{cycle}) \tag{5.16}$$

- **Pesi**
  Per quanto riguarda i pesi vale un discorso analogo a quello degli input. Di seguito le formule per il calcolo della banda nei diversi dataflow.
  *Input Stationary*:

  $$Weight_{BW} = R1/(E0 * T_{cycle}) \tag{5.17}$$

  *Weight Stationary*:

  $$Weight_{BW} = R1/(E0 * E2 * T_{cycle}) \tag{5.18}$$

  *Output Stationary*:

  $$Weight_{BW} = R1/(E0 * T_{cycle}) \tag{5.19}$$

- **Output**
  Per gli output è necessario nuovamente calcolare la dimensione dei blocchi calcolati in parallelo e ogni quanti cicli un nuovo blocco di output viene generato (formula 5.5).
  Utilizzando questi due valori è possibile calcolare la banda necessaria per il trasferimento degli output.

$$Output_{BW} = setSize/(newSetCycles * T_{cycle}) \tag{5.20}$$

La banda di memoria complessiva richiesta dal sistema dipende da diversi fattori, tra cui l'organizzazione della memoria e le sue capacità di accesso. Nello specifico, la presenza di una memoria unificata per i diversi tipi di dati (pesi, input, output) e la possibilità di accedere simultaneamente a più locazioni di memoria influenzano la banda necessaria. Nel caso di un'architettura con una singola DRAM dotata di una sola porta di accesso, la banda totale richiesta sarà pari alla somma delle singole bande necessarie per il trasferimento di ciascun tipo di dato (pesi, input, output). Questo perché l'accesso ai dati avviene sequenzialmente, un dato alla volta, attraverso l'unica porta disponibile.
Per individuare la configurazione ottimale dei parametri dell'acceleratore, si è optato per un approccio iterativo. Partendo da un set di valori iniziali, l'algoritmo li modifica progressivamente, cercando la soluzione che massimizza le prestazioni.
L'ordine di iterazione dei parametri è stato definito in modo da dare priorità alle soluzioni che, sulla base di stime preliminari, sembrano più promettenti in termini di prestazioni. Questo accorgimento permette di ridurre il numero di iterazioni necessarie per raggiungere una soluzione ottimale, accelerando il processo di ricerca. È importante sottolineare che i parametri *E0* e *R0* non vengono modificati direttamente durante le iterazioni, ma calcolati a partire dai valori degli altri parametri.

**Ricerca dei parametri ottimali**

```
for k in range(PE_Number, 0, -1):
    for E1 in range(PE[1], 0, -1):
        R1 = k - E1
        if 1 <= R1 <= PE[1]:
            E3 = 1
            while E3 <= math.ceil(E/PE[1]):
                R3 = 1
                while R3 <= math.ceil(R/PE[0]):
                    R2 = 1
                    while E2 <= math.ceil(E/(E1*E3)):
                        R2 = 1
                        while R2 <= math.ceil(R/(R1*R3)):
                            E0 = math.ceil(E/(E1*E2*E3))
                            R0 = math.ceil(R/(R1*R2*R3))

                            ##########################################
                            # Controllo della memoria e della banda
                            ##########################################

                            R2 += 1
                        E2 += 1
                    R3 += 1
                E3 += 1
```

Nei loop più interni vengono modificati i parametri *R2* e *E2* che, insieme ai parametri *R0* e *E0*, determinano la dimensione dei blocchi di memoria locale, nello specifico quante in quante serie di operazioni sequenziali verranno utilizzati i dati presenti in memoria.
Mantenendo costante la dimensione della memoria globale rimarrà costante anche il numero di accessi alla memria esterna.
Nei successivi due loop vengono modificati i parametri *R3* e *E3* che determinano la dimensione dei blocchi di memoria globale. In questo caso il numero di accessi alla memoria esterna aumenterà, incrementando la latenza e il consumo di energia.
Infine, nei loop più esterni vengono modificati i parametri *R1* e *E1* che determinano il numero di PE utilizzati e di conseguenza il numero di operazioni eseguite in parallelo. Questi due parametri sono modificati per ultimi in quanto la riduzione del numero di elementi di calcolo comporta un notevole aumento del tempo di esecuzione e una riduzione complessiva dell'efficienza dell'acceleratore. Questa modifica ciene quindi considerata solamente se non è possibile ottenere un risultato ottimale con i parametri precedenti.

Per ciascuna configurazione di parametri generata durante il processo iterativo, vengono stimati l'occupazione di memoria e la banda richiesta. Questi valori vengono poi confrontati con i limiti imposti dalle specifiche dell'acceleratore hardware. Se la configurazione risulta essere implementabile, ovvero se l'occupazione di memoria e la banda richiesta rientrano nei limiti consentiti, l'algoritmo procede a calcolare il tempo necessario per l'esecuzione di un layer della rete neurale. Inoltre, viene

determinato il numero totale di operazioni che saranno eseguite, includendo anche quelle ridondanti.

| **Configurazioni implementabili** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **R0** | **R1** | **R2** | **R3** | **E0** | **E1** | **E2** | **E3** | **Cicli di esecuzione** | **% mem occupata** |
| 1 | 16 | 1 | 1 | 20 | 8 | 4 | 1 | 80 | 3.65 |
| 1 | 16 | 1 | 1 | 16 | 8 | 5 | 1 | 80 | 3.45 |
| 1 | 16 | 1 | 1 | 10 | 8 | 8 | 1 | 80 | 3.16 |
| 1 | 16 | 1 | 1 | 8 | 8 | 10 | 1 | 80 | 3.16 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 1 | 16 | 1 | 1 | 1 | 8 | 80 | 1 | 80 | 2.73 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 1 | 16 | 1 | 1 | 3 | 8 | 27 | 1 | 81 | 2.90 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 2 | 5 | 1 | 1 | 6 | 7 | 16 | 1 | 192 | 1.76 |

Tabella 5.4: Alcune configurazioni in ordine di tempo di esecuzione crescente

La tabella 5.4.5 mostra alcune delle configurazioni che soddisfano i vincoli imposti dalle specifiche hardware. Come si può notare il tempo di esecuzione cresce con la riduzione del numero di PE utilizzati e con l'aumento del numero di operazioni seriali.
La quantità di memoria occupata aumenta notevolmente con l'aumentare della quantità di dati contenuta nelle memorie locali ma ha un andamento descrescente con la riduzione del numero di PE utilizzati.

Tra tutte le possibili configurazioni generate, l'algoritmo di ottimizzazione seleziona quella che minimizza il tempo di esecuzione e che presenta i parametri *R3* ed *E3* più prossimi a 1 (in caso non si riescano ad effettuare distinzioni utilizzando questi si passerà a *R2* ed *E2*). Si è scelto di dare priorità ai due parametri anziché alla dimensione della memoria occupata in quanto questa non comporta una variazione di prestazioni(assodato che non superi i limiti imposti dalle specifiche hardware) mentre una riduzione del numero degli accessi alla memoria esterna può comportare una riduzione dei consumi energetici.

Come risultato della procedura appena descritta si ottiene una configurazione ottimale per l'acceleratore, in grado di eseguire le operazioni di un layer in modo efficiente.
Nel caso specifico, la configurazione ottimale ottenuta è caratterizzata dai seguenti parametri:

R0 = 1, R1 = 16, R2 = 1, R3 = 1
E0 = 20, E1 = 8, E2 = 4, E3 = 1

Lo stesso software è stato utilizzato per calcolare il tempo di esecuzione, la dimensione dei blocchi di memoria e la banda richiesta per il trasferimento dei dati.

| **Cicli** | **Operazioni** | **Operazioni non utili** |
|---|---|---|
| 80 | 6310 | 3930 |

Tabella 5.5: Parametri di esecuzione

| **Dimensione delle memorie** | | |
|---|---|---|
| **Tipologia** | **Dimensione** | |
| PEs Matrix | 1360 | Byte |
| Local memory | 138 | Byte |
| Global memory | 1299 | Byte |
| **Tot Internal** | 2797 | Byte |
| External memory | 1299 | Byte |
| **Banda delle memorie** | | |
| **Tipologia** | **Banda** | |
| Input | 160 | MBps |
| Weight | 2.5 | MBps |
| Output | 183 | MBps |

Tabella 5.6: Specifiche della configurazione ottimale

Il grafico che illustra l'utilizzo dei dati conferma il corretto funzionamento dell'acceleratore, in linea con le previsioni. Nello specifico, si osserva che:

- **I pesi rimangono costanti per l'intera durata dell'elaborazione:** Questo comportamento è coerente con l'implementazione del dataflow Weight Stationary, in cui i pesi vengono caricati nelle Processing Element (PE) e mantenuti fissi durante l'elaborazione degli input.
- **Gli input vengono elaborati a gruppi di 8 per 20 cicli:** Questo indica che l'acceleratore processa 8 input in parallelo ($E1 = 8$) per un totale di 20 cicli di clock ($E2 = 20$). Al termine di questi 20 cicli, il gruppo di input viene sostituito con un nuovo set di dati.

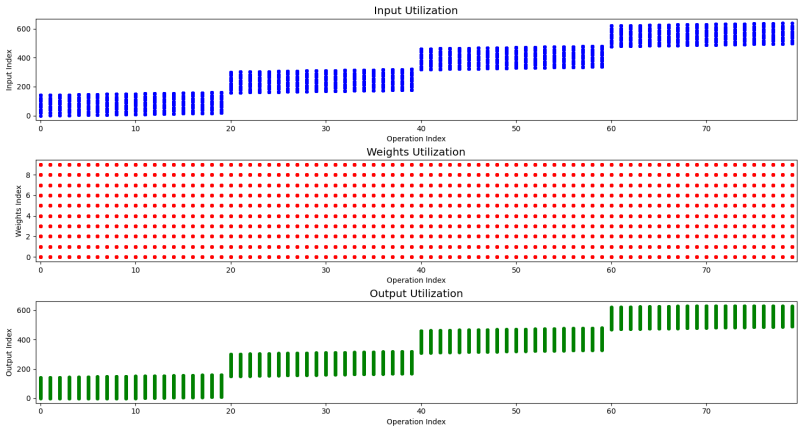



Figura 5.10: Utilizzo dei dati nella configurazione del quinto step

## 5.5 Analisi delle configurazioni

Osservando l'evoluzione delle configurazioni generate durante il processo di ottimizzazione, si nota una riduzione del parallelismo a favore di un aumento delle operazioni seriali. Questo compromesso, pur incrementando il tempo di esecuzione, è necessario per contenere la dimensione della memoria occupata e la banda richiesta. Tuttavia, l'aumento delle operazioni seriali comporta anche un incremento delle operazioni ridondanti (o "wasted operations"), ovvero operazioni che non contribuiscono alla generazione di risultati utili.

| **Step** | **Cicli di esecuzione** | **Operazioni** | **Operazioni non utili** |
|---|---|---|---|
| Step 1 | 1 | 6310 | 0 |
| Step 2 | 640 | 6310 | 90 |
| Step 3 | 640 | 6310 | 78170 |
| Step 4 | 80 | 6310 | 3930 |
| Step 5 | 80 | 6310 | 3930 |

Tabella 5.7: Confronto tra i tempi di esecuzione

La tabella 5.5 evidenzia un comportamento anomalo della configurazione descritta nello step 3. Tale anomalia è dovuta alla dimensione ridotta del tensore dei pesi, inferiore al numero di Processing Element (PE) istanziati. In questo scenario, la maggior parte delle PE rimane inattiva, non avendo dati su cui operare, causando un elevato numero di operazioni ridondanti e un conseguente spreco di risorse.

# Capitolo 6

# Conclusioni

Attraverso questo studio è stata analizzata la possibilità di eseguire algoritmi di machine learning su dispositivi embedded, spaziando da microcontrollori a basso consumo a sistemi più complessi dotati di acceleratori hardware dedicati. Sebbene l'esecuzione degli algoritmi sia risultata possibile su tutte le piattaforme considerate, le prestazioni effettive hanno evidenziato significative differenze. La scelta del dispositivo ottimale dipende fortemente dai requisiti specifici dell'applicazione. Lo studio ha confermato la correlazione tra frequenza di clock del processore, latenza di esecuzione e consumo energetico. Nel caso dei microcontrollori, privilegiare un basso consumo energetico implica l'utilizzo di frequenze di clock ridotte, a scapito delle prestazioni. L'impiego di librerie ottimizzate o di acceleratori hardware consente di mitigare questo trade-off, ottenendo performance superiori a parità di consumo.
In particolare, il framework STM32Cube.AI, integrato nell'ambiente di sviluppo STM32Cube, ha dimostrato di ridurre sia i tempi di esecuzione che i consumi energetici rispetto a soluzioni basate su librerie standard. Risultati analoghi sono stati ottenuti utilizzando l'acceleratore hardware Edge TPU presente nella Coral Dev Board Micro. Tuttavia, la difficoltà di misurare con precisione il consumo energetico di quest'ultima piattaforma ha impedito un confronto quantitativo diretto con gli altri dispositivi. Entrambe le soluzioni ottimizzate presentano però delle limitazioni:

- le librerie ottimizzate non sono sempre disponibili e la loro scrittura necessita di conoscenze approfondite dell'hardware, possibili solo per i produttori dei dispositivi;
- gli acceleratori hardware introducono un costo e una complessità aggiuntiva nella realizzazione delle schede elettroniche, rendendo questo approccio poco conveniente per piccole produzioni.

Un approccio alternativo, potenzialmente in grado di bilanciare prestazioni e flessibilità, consiste nell'utilizzare un acceleratore hardware programmabile implementato su FPGA. La progettazione di tale acceleratore richiede un'attenta analisi per garantire l'ottimizzazione rispetto all'applicazione specifica.
In questo contesto, lo studio ha approfondito gli acceleratori basati su *Systolic Array*, analizzando diverse architetture e flussi di dati. È stata inoltre definita e implementata una procedura per la configurazione dei parametri dell'acceleratore, al fine di

massimizzare le prestazioni in relazione alla rete neurale da eseguire. L'analisi preliminare si è concentrata su reti neurali con layer monodimensionali. Sviluppi futuri prevedono l'estensione della procedura a reti neurali più complesse, con l'obiettivo di stimare le prestazioni di acceleratori hardware per modelli di machine learning impiegati in applicazioni reali.

# Appendice A

# Appendice

## A.1 Buffer seriale nella libreria *coralmicro*

Durante lo sviluppo del firmware per l'esecuzione di benchmark sulla Coral Dev Board Micro, si sono verificate anomalie nella comunicazione seriale tra l'host e la scheda. Nello specifico, il microcontrollore entrava in uno stato di fault a seguito della ricezione di un determinato numero di byte, compromettendo la comunicazione e l'esecuzione del codice. L'analisi del codice sorgente della libreria coralmicro ha permesso di individuare la causa del problema: nella funzione `int ConsoleM7::Read(char* buffer, int size)`, responsabile della lettura dei byte dal buffer utilizzato per la comunicazione, si verificava un overflow del puntatore dopo un numero definito di iterazioni. Tale condizione generava un'eccezione di segmentation fault.

**Funzione di lettura del buffer seriale**

```
1  int ConsoleM7::Read(char* buffer, int size) {
2      if (!rx_task_) {
3          return -1;
4      }
5      MutexLock lock(rx_mutex_);
6      int bytes_to_return = std::min(size, static_cast<int>(rx_buffer_available_));
7
8      if (!bytes_to_return) {
9          return -1;
10     }
11     int bytes_to_read = bytes_to_return;
12     if (rx_buffer_read_ > rx_buffer_write_) {
13         memcpy(buffer, &rx_buffer_[rx_buffer_read_], kRxBufferSize - rx_buffer_read_);
14         bytes_to_read -= kRxBufferSize - rx_buffer_read_;
15         if (bytes_to_read) {
16             memcpy(buffer + (bytes_to_return - bytes_to_read), &rx_buffer_[0], bytes_to_read);
17         }
18     } else {
19         memcpy(buffer, &rx_buffer_[rx_buffer_read_], bytes_to_read);
20     }
21     rx_buffer_available_ -= bytes_to_return;
22     rx_buffer_read_ = (rx_buffer_read_ + bytes_to_return) % kRxBufferSize;
23
24     return bytes_to_return;
25 }
```

Il problema è stato risolto modificando la riga 15 e aggiungendo un controllo per evitare che il puntatore superi la dimensione del buffer prima dell'operazione di copia tra il buffer seriale e il quello di destinazione. Il codice modificato è il seguente:

**Funzione di lettura del buffer seriale**

```
1  int ConsoleM7::Read(char* buffer, int size) {
2      if (!rx_task_) {
3          return -1;
4      }
5      MutexLock lock(rx_mutex_);
6      int bytes_to_return = std::min(size, static_cast<int>(rx_buffer_available_));
7
8      if (!bytes_to_return) {
9          return -1;
10     }
11     int bytes_to_read = bytes_to_return;
12     if (rx_buffer_read_ > rx_buffer_write_) {
13         /*memcpy(buffer, &rx_buffer_[rx_buffer_read_],
14                 kRxBufferSize - rx_buffer_read_);*/
15
16         // Add a check to see if the number of bytes
17         // to read is less than the number of bytes
18         // from the read pointer to the end of the buffer.
19         if (bytes_to_read > kRxBufferSize - rx_buffer_read_) {
20         memcpy(buffer, &rx_buffer_[rx_buffer_read_], kRxBufferSize - rx_buffer_read_);
21         }
22         else {
23         memcpy(buffer, &rx_buffer_[rx_buffer_read_], bytes_to_read);
24         }
25
26         bytes_to_read -= kRxBufferSize - rx_buffer_read_;
27         if (bytes_to_read) {
28             memcpy(buffer + (bytes_to_return - bytes_to_read), &rx_buffer_[0], bytes_to_read);
29         }
30     } else {
31         memcpy(buffer, &rx_buffer_[rx_buffer_read_], bytes_to_read);
32     }
33     rx_buffer_available_ -= bytes_to_return;
34     rx_buffer_read_ = (rx_buffer_read_ + bytes_to_return) % kRxBufferSize;
35
36     return bytes_to_return;
37 }
```

# A.2 Risultati dei benchmark

## A.2.1 Tabelle dei risultati

| | Accuracy | | Idle | Load | | | Inference | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Bench-mark Network | Top1 | AUC | Avg Current [mA] | Time [ms] | Avg Current [mA] | Load Avg Energy [mJ] | Time [ms] | Avg Current [mA] | Avg Energy [mJ] | Voltage [V] |
| AD01 | 77,4 | 0,86 | 15,01 | 305,89 | 15,03 | 15,23 | 18,13 | 15,83 | 0,95 | 3,31 |
| IC01 | 87,5 | 0,98 | 14,628 | 374,94 | 14,82 | 18,57 | 823,02 | 15,33 | 42,19 | 3,35 |
| KWS01 | 90,1 | 0,99 | 14,65 | 78,37 | 14,72 | 3,83 | 172,90 | 15,21 | 8,72 | 3,32 |
| VWWS01 | 85,8 | 0,94 | 14,88 | 3259,00 | 14,93 | 162,714 | 551,72 | 15,40 | 28,40 | 3,34 |

Tabella A.1: Risultati del benchmark per la scheda Nucleo core Cortex-M4

| | Accuracy | | Idle | Load | | | Inference | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Bench-mark Network | Top1 | AUC | Avg Current [mA] | Time [ms] | Avg Current [mA] | Load Avg Energy [mJ] | Time [ms] | Avg Current [mA] | Avg Energy [mJ] | Voltage [V] |
| AD01 | 77,40 | 0,86 | 37,95 | 312,95 | 38,12 | 39,82 | 9,58 | 41,06 | 1,31 | 3,34 |
| IC01 | 87,50 | 0,98 | 35,80 | 371,61 | 36,02 | 44,75 | 433,90 | 39,28 | 56,95 | 3,34 |
| KWS01 | 90,10 | 0,99 | 35,88 | 74,83 | 36,04 | 9,01 | 93,61 | 38,99 | 12,19 | 3,34 |
| VWWS01 | 85,80 | 0,94 | 35,79 | 3192,57 | 36,07 | 384,95 | 293,73 | 38,78 | 38,06 | 3,34 |

Tabella A.2: Risultati del benchmark per la scheda Nucleo core Cortex-M7 @ 216MHz

| | Accuracy | | Idle | Load | | | Inference | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Bench-mark Network | Top1 | AUC | Avg Current [mA] | Time [ms] | Avg Current [mA] | Load Avg Energy [mJ] | Time [ms] | Avg Current [mA] | Avg Energy [mJ] | Voltage [V] |
| AD01 | 77.40 | 0.86 | 51.79 | 297.85 | 52.20 | 51.80 | 9.28 | 53.91 | 1.66 | 3.32 |
| IC01 | 87.50 | 0.98 | 51.70 | 360.08 | 51.69 | 62.14 | 361.17 | 53.43 | 64.39 | 3.34 |
| KWS01 | 90.10 | 0.99 | 51.76 | 70.45 | 52.30 | 12.31 | 74.76 | 52.67 | 13.15 | 3.34 |
| VWWS01 | 85.80 | 0.94 | 51.73 | 3111.61 | 51.70 | 537.07 | 244.52 | 52.58 | 42.91 | 3.34 |

Tabella A.3: Risultati del benchmark per la scheda Nucleo core Cortex-M7 @ 400MHz

| | Accuracy | | Idle | Load | | | Inference | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Bench-mark Network | Top1 | AUC | Avg Current [mA] | Time [ms] | Avg Current [mA] | Load Avg Energy [mJ] | Time [ms] | Avg Current [mA] | Avg Energy [mJ] | Voltage [V] |
| AD01 | 77.40 | 0.86 | 33.72 | 292.80 | 33.63 | 32.82 | 2.99 | 39.14 | 0.39 | 3.33 |
| IC01 | 87.00 | 0.98 | 33.61 | 340.14 | 33.58 | 38.14 | 68.57 | 42.68 | 9.76 | 3.34 |
| KWS01 | 90.20 | 0.99 | 33.87 | 69.75 | 33.92 | 7.89 | 24.16 | 42.48 | 3.42 | 3.34 |
| VWWS01 | 85.70 | 0.94 | 33.58 | 3068.80 | 33.54 | 343.11 | 41.47 | 43.17 | 5.95 | 3.33 |

Tabella A.4: Risultati del benchmark per la scheda Nucleo core Cortex-M7 e Cube.AI

| | Accuracy | | Idle | Load | | | Inference | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Bench-mark Network | Top1 | AUC | Avg Current [mA] | Time [ms] | Avg Current [mA] | Load Avg Energy [mJ] | Time [ms] | Avg Current [mA] | Avg Energy [mJ] | Voltage [V] |
| AD01 | 77.40 | 0.86 | 140.32 | 328.04 | 140.36 | 222.62 | 6.57 | 154.52 | 4.86 | 4.79 |
| IC01 | 87.50 | 0.98 | 138.94 | 394.90 | 139.13 | 272.23 | 262.59 | 160.94 | 208.04 | 4.92 |
| KWS01 | 90.10 | 0.99 | 143.08 | 77.75 | 144.79 | 54.92 | 40.61 | 169.05 | 33.11 | 4.82 |
| VWWS01 | 85.80 | 0.94 | 142.58 | 3475.26 | 141.88 | 2447.36 | 114.62 | 163.94 | 92.71 | 4.93 |

Tabella A.5: Risultati del benchmark per la scheda Dev Board Micro

| | Accuracy | | Idle | Load | | | Inference | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Bench-mark Network | Top1 | AUC | Avg Current [mA] | Time [ms] | Avg Current [mA] | Load Avg Energy [mJ] | Time [ms] | Avg Current [mA] | Avg Energy [mJ] | Voltage [V] |
| AD01 | 77.00 | 0.86 | 154.53 | 330.34 | 156.05 | 256.23 | 0.85 | 157.28 | 0.66 | 4.95 |
| IC01 | 87.00 | 0.98 | 174.48 | 396.07 | 174.48 | 332.55 | 1.47 | 180.31 | 1.26 | 4.78 |
| KWS01 | 90.20 | 0.90 | 171.31 | 74.41 | 169.93 | 62.63 | 1.88 | 176.35 | 1.63 | 4.92 |
| VWWS01 | 85.60 | 0.94 | 157.45 | 3480.29 | 157.94 | 2722.75 | 4.42 | 172.60 | 3.76 | 4.92 |

Tabella A.6: Risultati del benchmark per la scheda Dev Board Micro con EdgeTPU

### A.2.2 Grafici dei risultati

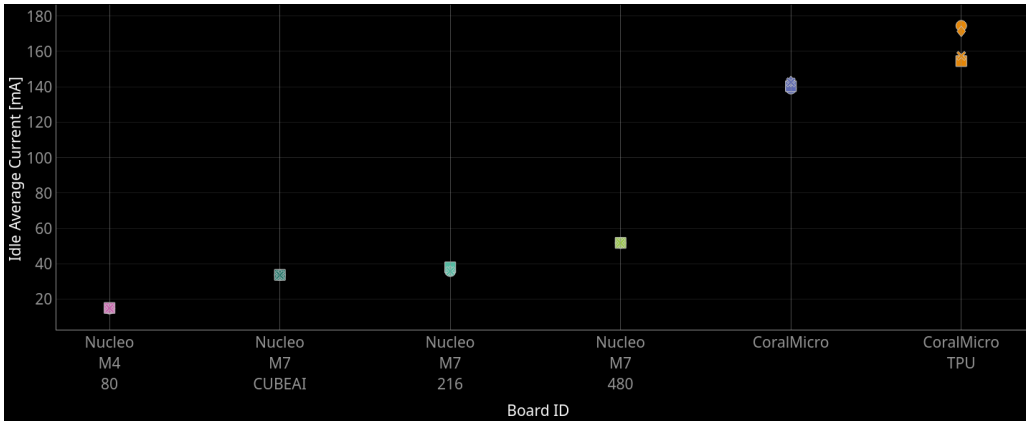



Figura A.1: Corrente assorbita nello stato di idle

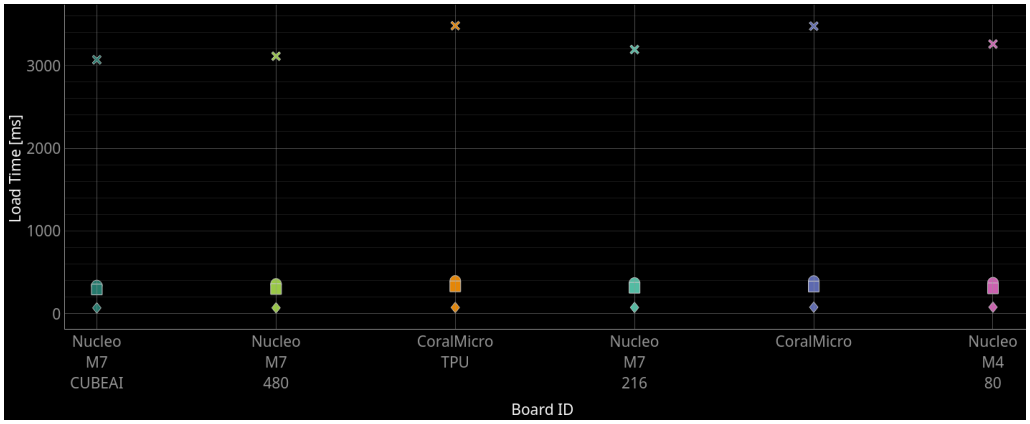



Figura A.2: Tempo di esecuzione del benchmark

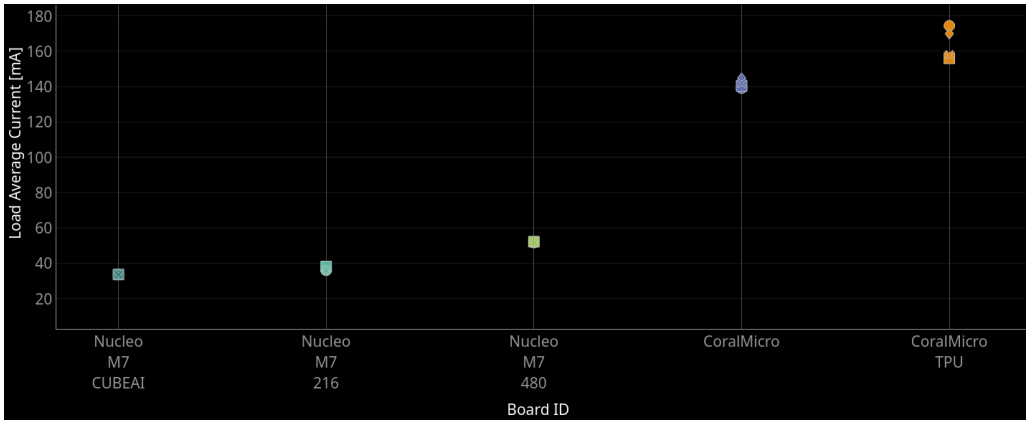



Figura A.3: Corrente assorbita durante la fase di inferenza del benchmark

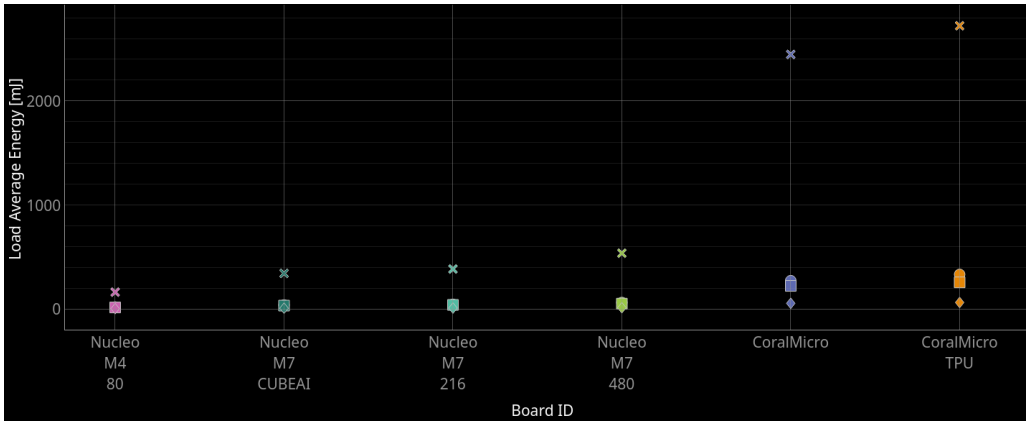



Figura A.4: Energia consumata durante la fase di inferenza del benchmark

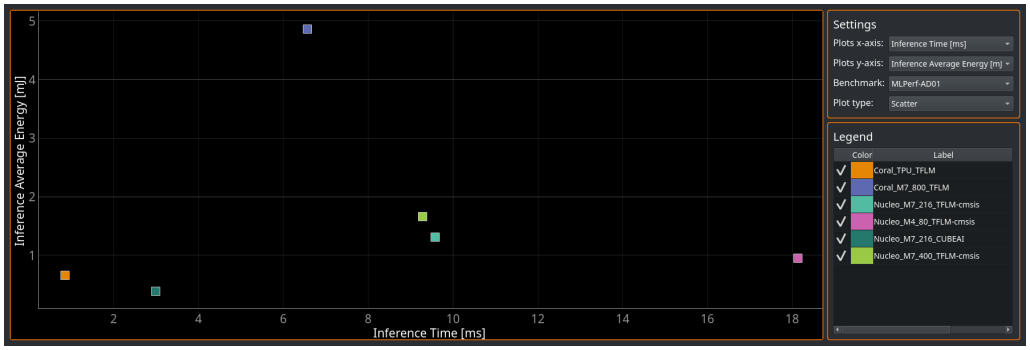



Figura A.5: Energia consumata in funzione del tempo di esecuzione per il benchmark AD01

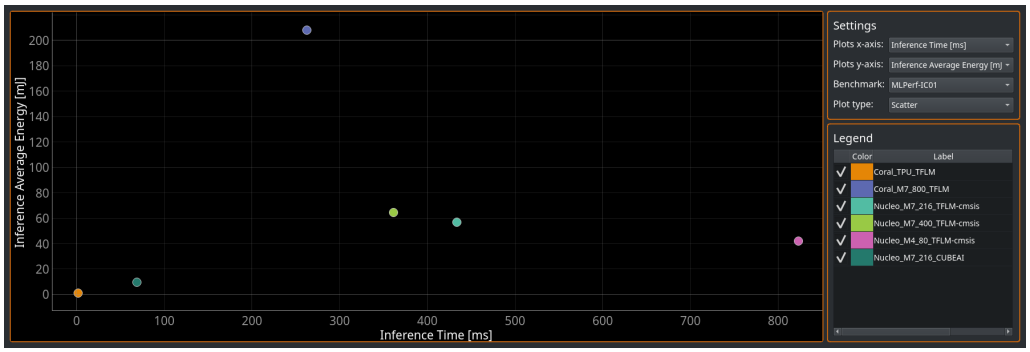



Figura A.6: Energia consumata in funzione del tempo di esecuzione per il benchmark IC01

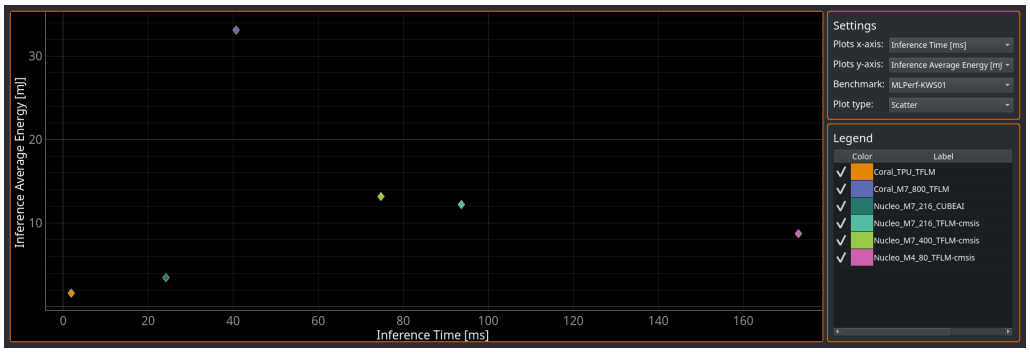



Figura A.7: Energia consumata in funzione del tempo di esecuzione per il benchmark KWS01

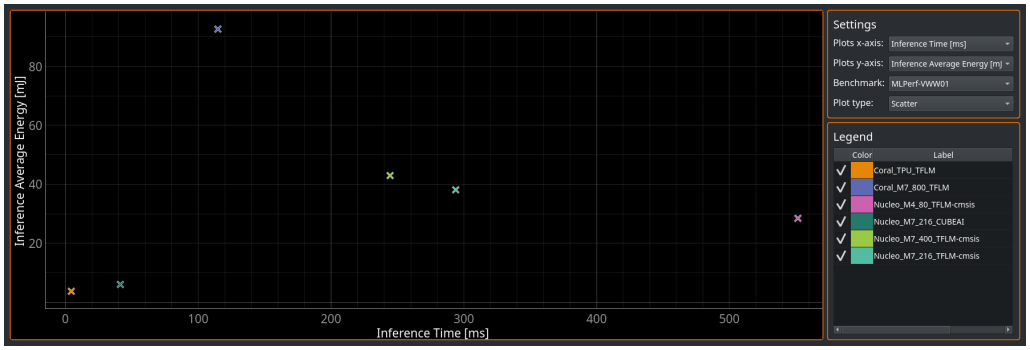



Figura A.8: Energia consumata in funzione del tempo di esecuzione per il benchmark VWWS01

## A.3 Script per l'ottimizzazione della struttura di un acceleratore 1D

### A.3.1 Variabili globali

```
"accelerator": [
    "devicekDescriptor":[
        "name": "TestDevice",
        "ExternalMemories": [
            {"name": "ExternalMemory1",
            "size": 8,
            "u_size": "Mb",
            "bandwidth": 50,
            "u_bandwidth": "Mbps",
            "busWidth": 8,
            "u_busWidth": "b",
            "containsInputs": true,
            "containsWeights": true,
            "containsOutputs": true}
        ],
        "InternalMemory":
            {"size": 594,
            "u_size": "Kb",
            "bandwidth": 200,
            "u_bandwidth": "Mbps",
            "busWidth": 8,
            "u_busWidth": "b"},
        "ProcessingElements":
            {"count": 132,
            "matrixShape": [4, 2],
            "registers": 13,
            "executionFrequency": 200,
            "u_executionFrequency": "MHz"
            }
    ],
    "networkDescriptor":[
        "name": "testNetwork",
        "layers": [
            {"name": "layer0",
            "id": 0,
            "inputShape":[80],
            "filterShape": [16],
            "outputShape": [64],
            "dataSize": 1},
            {"name": "layer1",
            "id": 1,
            "inputShape":[5],
            "filterShape": [3],
            "outputShape": [3],
            "dataSize": 1}
        ]
    ],
    "dataflowType": "WS"
]
```

### A.3.2 Step1: Layer shape and Size

```
# Consider an unbounded scenario where all operation are executed in parallel
R0 = 1          # No serialization of weights
E0 = 1          # No serialization of outputs

R1 = 1
```

```
E1 = 1
for layer in accelerator.networkDescriptor["layers"]:
    # multiply every dimension of the weights tensor
    weightsSize = 1
    for dim in layer["filterShape"]:
        weightsSize *= dim
    weightsSize *= layer["dataSize"]

    # multiply every dimension of the output tensor
    outputSize = 1
    for dim in layer["outputShape"]:
        outputSize *= dim
    outputSize *= layer["dataSize"]

    R1 = max(R1, weightsSize)
    E1 = max(E1, outputSize)

R2 = 1          # All weights are stored PEs
E2 = 1          # All outputs are stored P

R3 = 1          # All weights are stored in the internal memory
E3 = 1          # All outputs are stored in the internal memory

print("R0: {}, R1: {}, R2: {}, R3: {}".format(R0, R1, R2, R3))
print("E0: {}, E1: {}, E2: {}, E3: {}".format(E0, E1, E2, E3))
```

### A.3.3 Step2: Dataflow

```
# Change parameters to follow a desired dataflow
if dataflowType == "WS":
    E0 = E1 + R1 - 1
    E1 = 1
elif dataflowType == "OS":
    R0 = R1
    R1 = 1
elif dataflowType == "IS":
    R0 = R1
    R1 = 1
    E1 += R0 - 1
else:
    print("Invalid dataflow type")
    return False

dataflowType = dataflowType

print("Dataflow type: {}".format(dataflowType))
print("R0: {}, R1: {}, R2: {}, R3: {}".format(R0, R1, R2, R3))
print("E0: {}, E1: {}, E2: {}, E3: {}".format(E0, E1, E2, E3))
```

### A.3.4 Step3: PE Number

```
# The maximum value of the stationary parallel element is limited by the number of PEs

# Change parameters to follow a desired dataflow
if dataflowType == "WS":
    R0 = math.ceil(R1 / accelerator.hardwareDescriptor["ProcessingElements"]["count"])
    R1 = accelerator.hardwareDescriptor["ProcessingElements"]["count"]
elif dataflowType == "OS":
    E0 = math.ceil(E1 / accelerator.hardwareDescriptor["ProcessingElements"]["count"])
    E1 = accelerator.hardwareDescriptor["ProcessingElements"]["count"]
elif dataflowType == "IS":
    E0 = math.ceil(E1 / accelerator.hardwareDescriptor["ProcessingElements"]["count"])
    E1 = accelerator.hardwareDescriptor["ProcessingElements"]["count"]
else:
    print("Invalid dataflow type")
```

```
    return False

print("R0: {}, R1: {}, R2: {}, R3: {}".format(R0, R1, R2, R3))
print("E0: {}, E1: {}, E2: {}, E3: {}".format(E0, E1, E2, E3))
```

### A.3.5 Step4: PE Matrix Shape

```
# The maximum number of parallel operations is limited by the shape of the PE matrix. Usually the
↪  first dimension of the matrix is linked to the stationary element

if dataflowType == "WS":
    # E1 will be mapped to the second dimension of the PE matrix
    E1 = accelerator.hardwareDescriptor["ProcessingElements"]["matrixShape"][1]
    E0 = math.ceil(E0 / E1)

    # R1 will be mapped to the first dimension of the PE matrix
    tmpR0 = 0
    for layer in accelerator.networkDescriptor["layers"]:
        # multiply every dimension of the weights tensor
        weightsSize = 1
        for dim in layer["filterShape"]:
            weightsSize *= dim
        weightsSize *= layer["dataSize"]

        tmpR0 = max(tmpR0, weightsSize)

    R1 = accelerator.hardwareDescriptor["ProcessingElements"]["matrixShape"][0]
    R0 = math.ceil(tmpR0 / R1)

elif dataflowType == "OS":
    # R1 will be mapped to the second dimension of the PE matrix
    R1 = accelerator.hardwareDescriptor["ProcessingElements"]["matrixShape"][1]
    R0 = math.ceil(R0 / R1)

    # E1 will be mapped to the first dimension of the PE matrix
    tmpE0 = 0
    for layer in accelerator.networkDescriptor["layers"]:
        # multiply every dimension of the weights tensor
        inputsSize = 1
        for dim in layer["inputShape"]:
            weightsSize *= dim
        inputsSize *= layer["dataSize"]

        tmpE0 = max(tmpE0, inputsSize)
    E1 = accelerator.hardwareDescriptor["ProcessingElements"]["matrixShape"][0]
    E0 = math.ceil(tmpE0 / E1)

elif dataflowType == "IS":
    # R1 will be mapped to the second dimension of the PE matrix
    R1 = accelerator.hardwareDescriptor["ProcessingElements"]["matrixShape"][1]
    R0 = math.ceil(R0 / R1)

    # E1 will be mapped to the first dimension of the PE matrix
    tmpE0 = 0
    for layer in accelerator.networkDescriptor["layers"]:
        # multiply every dimension of the weights tensor
        outputsSize = 1
        for dim in layer["outputShape"]:
            outputsSize *= dim
        outputsSize *= layer["dataSize"]

        tmpE0 = max(tmpE0, outputsSize)
    E1 = accelerator.hardwareDescriptor["ProcessingElements"]["matrixShape"][0]
    E0 = math.ceil(tmpE0 / E1)
else:
    print("Invalid dataflow type")
```

```
    return False

print("R0: {}, R1: {}, R2: {}, R3: {}".format(R0, R1, R2, R3))
print("E0: {}, E1: {}, E2: {}, E3: {}".format(E0, E1, E2, E3))
```

### A.3.6 Step5: Memory Size

```
# Create some temporary variables
tmpR0 = 1
tmpR1 = R1
tmpR2 = 1
tmpR3 = 1

tmpE0 = 1
tmpE1 = E1
tmpE2 = 1
tmpE3 = 1

tmpR = 1
tmpE = 1

# Calculate the maximum size of the weights and output tensors
for layer in accelerator.networkDescriptor["layers"]:
    # multiply every dimension of the weights tensor
    weightsSize = 1
    for dim in layer["filterShape"]:
        weightsSize *= dim
    weightsSize *= layer["dataSize"]

    # multiply every dimension of the output tensor
    outputSize = 1
    for dim in layer["outputShape"]:
        outputSize *= dim
    outputSize *= layer["dataSize"]

    tmpR = max(tmpR, weightsSize)
    tmpE = max(tmpE, outputSize)

if dataflowType != "OS":
    tmpE += tmpR-1

# Init the variables
optimizations = []
solutions = 0
E3increment = 1
R3increment = 1
E3Min = math.ceil((tmpE * 2)/ accelerator.hardwareDescriptor["InternalMemory"]["size"])
E2increment = 1
R2increment = 1

checkInt = False
checkExt = False
checkInputIntBW = False
checkWeightIntBW = False
checkOutputIntBW = False
checkExtBW = False

# k is used to search first the solutions with the maximum number of PEs
for k in range(E1+R1, 0, -1):
    if len(optimizations) > 0:
        # this condition can be used to stop the search if the number of PE must be reduced but a
        ↪  solution has already been found
        pass
    for tmpE1 in range(E1, 0, -1):
        tmpR1 = k - tmpE1
        if 1 <= tmpR1 <= R1:
```

```
            E3increment = 1
            tmpE3 = E3Min
            while tmpE3 <= math.ceil(tmpE/E1):
                E3Check = True
                R3increment = 1
                tmpR3 = 1
                while tmpR3 <= math.ceil(tmpR/R1) and tmpR3 < 100:

                    E2increment = 1
                    tmpE2 = 1
                    while tmpE2 <= math.ceil(tmpE/(tmpE1*tmpE3)):

                        R2increment = 1
                        tmpR2 = 1
                        while tmpR2 <= math.ceil(tmpR/(tmpR1*tmpR3)):
                            tmpE0 = math.ceil(tmpE/(tmpE1*tmpE2*tmpE3))
                            tmpR0 = math.ceil(tmpR/(tmpR1*tmpR2*tmpR3))

                            # Check if the current parameters are valid
                            if tmpR0 * tmpR1 * tmpR2 * tmpR3 >= tmpR and tmpE0 * tmpE1 * tmpE2 *
                            ↪  tmpE3 >= tmpE:

                                # Check if, with the current parameters, all data can be stored on
                                ↪  chip and if the bandwidth is enough
                                checkInt, checkExt = checkMemory(tmpR, tmpR0, tmpR1, tmpR2, tmpR3,
                                ↪  tmpE, tmpE0, tmpE1, tmpE2, tmpE3)
                                checkInputIntBW, checkWeightIntBW, checkOutputIntBW, checkExtBW =
                                ↪  checkBandwidth(tmpR, tmpR0, tmpR1, tmpR2, tmpR3, tmpE, tmpE0,
                                ↪  tmpE1, tmpE2, tmpE3)

                                if dataflowType == "WS":
                                    opTime = (tmpE0 - 1) + (tmpR0 - 1)*tmpE0 + (tmpE2 -
                                    ↪  1)*tmpE0*tmpR0 + (tmpR2 - 1)*tmpE2*tmpE0*tmpR0 + (tmpE3 -
                                    ↪  1)*tmpR2*tmpE2*tmpE0*tmpR0 + (tmpR3 -
                                    ↪  1)*tmpE3*tmpR2*tmpE2*tmpE0*tmpR0
                                else:
                                    opTime = (tmpR0 - 1) + (tmpE0 - 1)*tmpR0 + (tmpR2 -
                                    ↪  1)*tmpE0*tmpR0 + (tmpE2 - 1)*tmpE2*tmpE0*tmpR0 + (tmpR3 -
                                    ↪  1)*tmpE2*tmpE0*tmpR0 + (tmpE3 - 1)*tmpR3*tmpE2*tmpE0*tmpR0

                                # If all the checks are passed, the solution is valid
                                if checkInt and checkExt and checkInputIntBW and checkWeightIntBW
                                ↪  and checkOutputIntBW and checkExtBW:
                                    solutions += 1

                                    # Calculate performances: Number of operation cycles
                                    if dataflowType == "WS":
                                        opTime = (tmpE0 - 1) + (tmpR0 - 1)*tmpE0 + (tmpE2 -
                                        ↪  1)*tmpE0*tmpR0 + (tmpR2 - 1)*tmpE2*tmpE0*tmpR0 + (tmpE3
                                        ↪  - 1)*tmpR2*tmpE2*tmpE0*tmpR0 + (tmpR3 -
                                        ↪  1)*tmpE3*tmpR2*tmpE2*tmpE0*tmpR0 + 1
                                    else:
                                        opTime = (tmpR0 - 1) + (tmpE0 - 1)*tmpR0 + (tmpR2 -
                                        ↪  1)*tmpE0*tmpR0 + (tmpE2 - 1)*tmpE2*tmpE0*tmpR0 + (tmpR3
                                        ↪  - 1)*tmpE2*tmpE0*tmpR0 + (tmpE3 -
                                        ↪  1)*tmpR3*tmpE2*tmpE0*tmpR0 + 1

                                    # Calculate performances: Memory occupation
                                    intMemoryOccupation = estimateFullInternalMemory(tmpR, tmpR0,
                                    ↪  tmpR1, tmpR2, tmpR3, tmpE, tmpE0, tmpE1, tmpE2, tmpE3)


                                    # Check if the current optimization is better than the previous
                                    ↪  ones
                                    found = False
                                    for opt in optimizations:
                                        # Find if this point has already been overcomed
                                        if opt["opTime"] <= opTime and opt["intMem"] <= perf:
```

```
                                        found = True
                                        break

                                if not found:
                                    for opt in optimizations:
                                    # Find if an older point has been overcomed by the current one
                                        if opt["opTime"] >= opTime and opt["intMem"] >= perf:
                                            pass
                                            optimizations.remove(opt)
                                    optimizations.append({"R0": tmpR0, "R1": tmpR1, "R2": tmpR2, "R3": tmpR3, "E0": tmpE0, "E1": tmpE1, "E2": tmpE2, "E3": tmpE3, "opTime": opTime, "intMem": intMemoryOccupation, "perf": perf})
                                else:
                                    E3Check = False

                            if not checkExtBW: # Since R2 increase the required external memory bandwidth, if the limit has been reached, there is no need to increase R2
                                break
                            else:
                                if tmpR0 > 1:
                                    R2increment = max(math.ceil(tmpR/((tmpR0-1)*tmpR1*tmpR3)) - tmpR2, 1)
                                tmpR2 += R2increment

                        if tmpE0 > 1:
                            E2increment = max(math.ceil(tmpE/((tmpE0-1)*tmpE1*tmpE3)) - tmpE2, 1)
                        tmpE2 += E2increment

                    if not checkExtBW: # Since R3 increase the required external memory bandwidth, if the limit has been reached, there is no need to increase R2
                        break
                    else:
                        if (tmpR0*tmpR1*tmpR2 > 1):
                            R3increment = max(math.ceil(tmpR/(tmpR/tmpR3-1)) - tmpR3, 1)
                        tmpR3 += R3increment

                if (tmpE0*tmpE1*tmpE2 > 1):
                    E3increment = max(math.ceil(tmpE/((tmpE/tmpE3)-1)) - tmpE3, 1)
                tmpE3 += E3increment

                if E3Check:
                    break

print("Optimizations found: {}".format(len(optimizations)))

# Sort the optimizations in ascending order of the number of operations, E3 and E2
optimizations = sorted(optimizations, key = lambda x: (x["opTime"], x["E3"], x["E2"]))

# Set the parameters to the best optimization
if len(optimizations) > 0:
    R0 = optimizations[0]["R0"]
    R1 = optimizations[0]["R1"]
    R2 = optimizations[0]["R2"]
    R3 = optimizations[0]["R3"]

    E0 = optimizations[0]["E0"]
    E1 = optimizations[0]["E1"]
    E2 = optimizations[0]["E2"]
    E3 = optimizations[0]["E3"]

    print("Optimization found:")
    print("R0: {}, R1: {}, R2: {}, R3: {}".format(R0, R1, R2, R3))
    print("E0: {}, E1: {}, E2: {}, E3: {}".format(E0, E1, E2, E3))
return optimizations
```

### A.3.7 CheckMemory

```
def checkMemory(self, R, R0, R1, R2, R3, E, E0, E1, E2, E3):
    checkInt = False
    checkExt = False

    PEMemory = estimatePEMatrixMemory(R, R0, R1, R2, R3, E, E0, E1, E2, E3)
    localInputMem, localWeightMem, localOutputMem = estimateLocalMemory(R, R0, R1, R2, R3, E, E0,
    ↪  E1, E2, E3)
    globalInputMem, globalWeightMem, globalOutputMem = estimateGlobalMemory(R, R0, R1, R2, R3, E,
    ↪  E0, E1, E2, E3)

    # If the data does not fit in the internal memory, a double buffering is needed
    if E3 != 1:
        globalInputMem *= 2
    if R3 != 1:
        globalWeightMem *= 2
    if E3 != 1 or R3 != 1:
        globalOutputMem *= 2

    externalInputMem, externalWeightMem, externalOutputMem = estimateExternalMemory(R, R0, R1, R2,
    ↪  R3, E, E0, E1, E2, E3)

    # Calculate the total available internal memory
    internalMemory = accelerator.hardwareDescriptor["InternalMemory"]["size"]

    # Calculate the total available external memory
    externalMemory = accelerator.hardwareDescriptor["ExternalMemories"][0]["size"]

    # Check if the memory needed is available
    if internalMemory >= PEMemory + localInputMem + localWeightMem + localOutputMem + globalInputMem
    ↪  + globalWeightMem + globalOutputMem:
        checkInt = True

    if externalMemory >= externalInputMem + externalWeightMem + externalOutputMem:
        checkExt = True

    return checkInt, checkExt
```

### A.3.8 CheckBandwidth

```
checkInputInt = False
checkWeightInt = False
checkOutputInt = False
checkExt = False

inputMemBW = estimateInputMemoryBandwidth(R, R0, R1, R2, R3, E, E0, E1, E2, E3)
weightMemBW = estimateWeightMemoryBandwidth(R, R0, R1, R2, R3, E, E0, E1, E2, E3)
outputMemBW = estimateOutputMemoryBandwidth(R, R0, R1, R2, R3, E, E0, E1, E2, E3)

# Calculate the internal memory bandwidth
internalMemoryBW = accelerator.hardwareDescriptor["InternalMemory"]["bandwidth"] * 1000 *
↪  accelerator.hardwareDescriptor["InternalMemory"]["busWidth"] * 8

# Calculate the external memory bandwidth
externalMemoryBW = accelerator.hardwareDescriptor["ExternalMemories"][0]["bandwidth"] * 1000 *
↪  accelerator.hardwareDescriptor["ExternalMemories"][0]["busWidth"] * 8

# Check if the bandwidth needed is available
if internalMemoryBW >= inputMemBW:
    checkInputInt = True
if internalMemoryBW >= weightMemBW:
    checkWeightInt = True
if internalMemoryBW >= outputMemBW:
    checkOutputInt = True

```

```
requiredExternalMemoryBW = 0
if R3 > 1:
    requiredExternalMemoryBW += weightMemBW
if E3 > 1:
    requiredExternalMemoryBW += inputMemBW
if R3 > 1 or E3 > 1:
    requiredExternalMemoryBW += outputMemBW

if externalMemoryBW >= requiredExternalMemoryBW:
    checkExt = True

return checkInputInt, checkWeightInt, checkOutputInt, checkExt
```

### A.3.9 EstimatePEMatrixMemory

```
def estimatePEMatrixMemory(self, R, R0, R1, R2, R3, E, E0, E1, E2, E3):
    regPerPE = accelerator.hardwareDescriptor["ProcessingElements"]["registers"]

    return R1*E1*regPerPE + R1*E1*(2+2)
```

### A.3.10 EstimateLocalMemory

```
def estimateLocalMemory(self, R, R0, R1, R2, R3, E, E0, E1, E2, E3):
    # Input Memory
    if dataflowType == "IS" or dataflowType == "WS":
        localInputMem = E0*E1
    else:
        localInputMem = E0*(E1+1)

    # Weight Memory
    localWeightMem = R0*R1

    # Output Memory
    blockOccupationTime = calculatetOutputMaxProcessingTime(R, R0, R1, R2, R3, E, E0, E1, E2, E3)
    blockNewTime = R0 * E0
    blockSize = E0*E1
    blocksConcurrent = min(math.ceil((E-R+1)/blockSize), math.ceil(blockOccupationTime /
    ↪  blockNewTime))
    localOutputMem = blockSize * blocksConcurrent

    if dataflowType == "IS" or dataflowType == "WS":
        outputNumber = E-R+1
    else:
        outputNumber = E

    if localOutputMem >= outputNumber:
        localOutputMem = outputNumber
        outputsMultiplier = 1
    else:
        outputsMultiplier = 2

    if E3 == 1 and E2 == 1:
        inputsMultiplier = 1
    else:
        inputsMultiplier = 2

    if R3 == 1 and R2 == 1:
        weightsMultiplier = 1
    else:
        weightsMultiplier = 2

    return localInputMem*inputsMultiplier, localWeightMem*weightsMultiplier,
    ↪  localOutputMem*outputsMultiplier
```

### A.3.11 EstimateGlobalMemory

```
def estimateGlobalMemory(self, R, R0, R1, R2, R3, E, E0, E1, E2, E3):
    # Input Memory
    if dataflowType == "IS" or dataflowType == "WS":
        globalInputMem = E0*E1*E2
    else:
        globalInputMem = E0*(E1+1)*E2

    # Weight Memory
    globalWeightMem = R0*R1*R2

    # Output Memory
    if dataflowType == "IS" or dataflowType == "WS":
        globalOutputMem = (E0*E1*E2)+(R0*R1*R2) - 1
    else:
        globalOutputMem = E0*E1*E2

    return globalInputMem, globalWeightMem, globalOutputMem
```

### A.3.12 EstimateExternalMemory

```
def estimateExternalMemory(self, R, R0, R1, R2, R3, E, E0, E1, E2, E3):
    # Input Memory
    if dataflowType == "IS" or dataflowType == "WS":
        externalInputMem = E0*E1*E2*E3
    else:
        externalInputMem = E0*(E1+1)*E2*E3

    # Weight Memory
    externalWeightMem = R0*R1*R2*R3

    # Output Memory
    if dataflowType == "IS" or dataflowType == "WS":
        externalOutputMem = (E0*E1*E2*E3)+(R0*R1*R2*R3) - 1
    else:
        externalOutputMem = E0*E1*E2*E3

    return externalInputMem, externalWeightMem, externalOutputMem
```

### A.3.13 EstimateInputMemoryBandwidth

```
def estimateInputMemoryBandwidth(self, R, R0, R1, R2, R3, E, E0, E1, E2, E3):
    Top = calculateTimePerOperation()

    if dataflowType == "IS":
        inputMemBW = E1/(R0*R2*Top)
    elif dataflowType == "WS":
        inputMemBW = E1/(R0*Top)
    else:
        inputMemBW = E1/(R0*R2*Top) + (R1-1)/(E0*Top)

    return inputMemBW
```

### A.3.14 EstimateWeightMemoryBandwidth

```
def estimateWeightMemoryBandwidth(self, R, R0, R1, R2, R3, E, E0, E1, E2, E3):
    Top = calculateTimePerOperation()

    if dataflowType == "IS":
        weightMemBW = R1/(E0*Top)
    elif dataflowType == "WS":
        weightMemBW = R1/(E0*E2*Top)
    else:
```

```
        weightMemBW = R1/(E0*Top)

    return weightMemBW
```

### A.3.15 EstimateOutputMemoryBandwidth

```
def estimateOutputMemoryBandwidth(self, R, R0, R1, R2, R3, E, E0, E1, E2, E3):
    Top = calculateTimePerOperation()

    dataSize = E0*E1
    dataCycles = calculatetOutputMaxProcessingTime(R, R0, R1, R2, R3, E, E0, E1, E2, E3) +1
    blockNewTime = R0 * E0
    blocksConcurrent = math.ceil(dataCycles / blockNewTime)

    outputMemBW = dataSize/((dataCycles/blocksConcurrent)*Top)

    return outputMemBW
```

### A.3.16 calculateTimePerOperation

```
def estimateOutputMemoryBandwidth(self, R, R0, R1, R2, R3, E, E0, E1, E2, E3):
    Top = calculateTimePerOperation()

    dataSize = E0*E1
    dataCycles = calculatetOutputMaxProcessingTime(R, R0, R1, R2, R3, E, E0, E1, E2, E3) +1
    blockNewTime = R0 * E0
    blocksConcurrent = math.ceil(dataCycles / blockNewTime)

    outputMemBW = dataSize/((dataCycles/blocksConcurrent)*Top)

    return outputMemBW
```

### A.3.17 calculateOutputMaxProcessingTime

```
def calculatetOutputMaxProcessingTime(self, R, R0, R1, R2, R3, E, E0, E1, E2, E3):
    time = 0
    time1 = 0
    time2 = 0
    time3 = 0
    time4 = 0

    if dataflowType == "IS":
        if R3 > 1:
            if (E0*E1*E2) < (R+E1-1):
                time2 = R0*E0*R2*E2*R3*round((R+E1-1)/(E0*E1*E2))
                rem = (R+E1-1) - E0*E1*E2*round((R+E1-1)/(E0*E1*E2))
                time3 = R0*E0*R2*math.ceil(rem/(E0*E1))
            else:
                time1 = R0*E0*R2*E2*(R3-1)
        else:
            if (E0*E1*E2) < (R+E1*R1):
                time1 = R0*E0*R2*E2*R3*round((R+E1*R1)/(E0*E1*E2))
                rem1 = (R+E1*R1) - E0*E1*E2*round((R+E1*R1)/(E0*E1*E2))
                time2 = R0*E0*R2*math.ceil(rem1/(E0*E1))
            elif (E0*E1) < (R+E1*R1):
                time1 = R0*E0*R2*round((R+E1*R1)/(E0*E1))
                rem2 = (R+E1*R1) - E0*E1*round((R+E1*R1)/(E0*E1))
                if rem2 < 0:
                    rem2 = 0
                time2 = R0*E0*R2*math.ceil(rem2/(E0))
            else:
                time1 = R0*E0*R2
        time = time1 + time2
```

```
30      elif dataflowType == "WS":
31          if R3 > 1:
32              time1 = E0*R0*E2*R2*E3*(R3-1)
33          elif R2 > 1:
34              time1 = E0*R0*E2*(R2-1)
35          else:
36              time1 = E0*R0
37
38          if (E0*E1*E2) < (R+E0*E1-1): # If the number of inputs stored in the internal memory is not
            ↪  sufficient to generate E0*E1 outputs
39              time2 = E0*R0*E2*R2*round((R+E0*E1-1)/(E0*E1*E2))
40              rem = (R+E0*E1-1) - E0*E1*E2*round((R+E0*E1-1)/(E0*E1*E2))
41              time3 = E0*R0*math.ceil(rem/(E0*E1))
42              rem = rem - E0*E1*math.ceil(rem/(E0*E1))
43              if rem < 0:
44                  rem = 0
45              time4 = E0*(rem/(E0))
46          elif (E0*E1) < (R+E0*E1-1):
47              time2 = 0
48              time3 = E0*R0*round((R+E0*E1-1)/(E0*E1))
49              rem = (R+E0*E1-1) - E0*E1*round((R+E0*E1-1)/(E0*E1))
50              if rem < 0:
51                  rem = 0
52              time4 = E0*(rem/(E0))
53
54
55          time = time1 + time2 + time3 + time4
56      elif dataflowType == "OS":
57          if R3 > 1:
58              time1 = math.ceil(E0*(R0*R2*R3)*0.5)
59              time2 = math.ceil(E0*(R0*R2*R3)*0.5) * math.ceil(E/(E0*E1))
60          else:
61              time1 = E0*(R0*R2*R3)
62
63          time = time1 + time2
64
65      return time
```

# Bibliografia

[1] Birju Tank e Vaibhav Gandhi. "A Comparative Study on Cloud Computing, Edge Computing and Fog Computing". In: gen. 2023. ISBN: 9781643683607. DOI: 10.3233/ATDE221329.

[2] Massimo Merenda, Carlo Porcaro e Demetrio Iero. "Edge Machine Learning for AI-Enabled IoT Devices: A Review". In: *Sensors* 20 (apr. 2020), p. 2533. DOI: 10.3390/s20092533.

[3] Colby R. Banbury et al. "MLPerf Tiny Benchmark". In: *CoRR* abs/2106.07597 (2021). arXiv: 2106.07597. URL: https://arxiv.org/abs/2106.07597.

[4] Martín Abadi et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems". In: (2015). Software available from tensorflow.org. URL: https://www.tensorflow.org/.

[5] Robert David et al. "TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems". In: *CoRR* abs/2010.08678 (2020). arXiv: 2010.08678. URL: https://arxiv.org/abs/2010.08678.

[7] *INA226 36V, 16-Bit, Ultra-Precise I2C Output Current, Voltage, and Power Monitor With Alert*. INA226. Texas Instruments. 2011. URL: https://www.ti.com/lit/gpn/ina226.

[9] Liangzhen Lai, Naveen Suda e Vikas Chandra. "CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs". In: *CoRR* abs/1801.06601 (2018). arXiv: 1801.06601. URL: http://arxiv.org/abs/1801.06601.

[11] Massimo Ruo Roch e Maurizio Martina. "VirtLAB: A Low-Cost Platform for Electronics Lab Experiments". In: *Sensors* 22.13 (2022). ISSN: 1424-8220. DOI: 10.3390/s22134840. URL: https://www.mdpi.com/1424-8220/22/13/4840.

[12] Flavius Opriţoiu e Mircea Vlăduţiu. "Systolic Array Architecture for Educational Use". In: (2023), pp. 18–23. DOI: 10.1109/ICSTCC59206.2023.10308496.

[13] Zhi-Gang Liu, Paul N. Whatmough e Matthew Mattina. "Systolic Tensor Array: An Efficient Structured-Sparse GEMM Accelerator for Mobile CNN Inference". In: (2020). arXiv: 2005.08098 [cs.DC]. URL: https://arxiv.org/abs/2005.08098.

[14] Yu-Hsin Chen, Joel S. Emer e Vivienne Sze. "Eyeriss v2: A Flexible and High-Performance Accelerator for Emerging Deep Neural Networks". In: *CoRR* abs/1807.07928 (2018). arXiv: 1807.07928. URL: http://arxiv.org/abs/1807.07928.

# Sitografia

[6] *Tensorflow Lite Micro*. https://github.com/tensorflow/tflite-micro/tree/main.

[8] *AI expansion pack for STM32CubeMX*. https://www.st.com/en/embedded-software/x-cube-ai.html.

[10] *coralmicro library*. https://github.com/google-coral/coralmicro/tree/main.