

Configurazione, installazione ed esecuzione di una CNN sulla DPU della Zynq Ultrascale+: Valutazione delle prestazioni rispetto ad una GPU in Cloud

L'avanzamento dell'intelligenza artificiale ha consentito il raggiungimento di risultati eccezionali in diversi ambiti, come come la classificazione di immagini/video, la segmentazione semantica, il rilevamento/tracciamento degli oggetti oltre ad altre applicazioni applicabili in un contesto embedded. Tuttavia, l'aumento delle dimensioni e della complessità dei modelli neurali richiesti per affrontare problemi di immagini di grandi dimensioni ha reso necessario eseguire tali reti neurali su dispositivi embedded con caratteristiche hardware specifiche al fine di ottenere elaborazioni in tempo reale con bassa latenza.

Nel presente elaborato, si esegue un'analisi dettagliata riguardante la configurazione, l'installazione e l'esecuzione di una rete neurale convoluzionale (CNN) sulla board Zynq Ultrascale+ (versione ZCU102). Questa piattaforma combina un sistema multiprocessore su chip (MPSoC) con un FPGA per offrire un'accelerazione hardware dedicata all'esecuzione efficiente delle reti neurali, riducendo al minimo il carico computazionale sul processore principale.

L'obiettivo principale del presente studio è valutare il possibile degrado delle prestazioni quando un modello CNN viene eseguito sulla Deep Learning Processing Unit (DPU) della ZCU102 rispetto all'esecuzione su un'infrastruttura Cloud. Nel caso in esame si è scelto di usare **Kaggle** come piattaforma Cloud. Attraverso un'analisi comparativa e qualitativa, saranno presi in considerazione diversi aspetti rilevanti, al fine di fornire una valutazione delle prestazioni sulla board target ZCU102.

Contents

Overview ZCU102 MPSoC

- Desrizione dell'architettura Hardware della ZCU102
- Deep Learning Processor Unit (DPU)
- Overview dell'applicazione: Traffic Sign Recognition System
- Il Dataset

PetaLinux

- Introduzione: importanza nel contesto di sviluppo su ZCU102
- Configurazione di PetaLinux per il progetto in esame
 - Creazione del progetto
 - Configurazione dell'HW: la DPU
 - Importazione della configurazione HW e build del progetto
 - Creazione dell'immagine di boot

Vitis AI

- Componenti fondamentali di Vitis AI
 - Vitis AI Runtime (VART)
 - AI Quantizer
 - AI Compiler

Configurazione dell'host

- Utilizzo di PetaLinux sull'host
 - Avvio di QEMU
- Utilizzo di Vitis AI sull'host
 - Configurazione Docker

Configurazione del target (ZCU102)

- Interconnessione dell'host e del target
 - UART
 - Ethernet
- Utilizzo di PetaLinux sul target (ZCU102)
 - Flash dell'immagine su SD card
 - Caricamento della DPU sulla parte PL del target (ZCU102)
- Utilizzo di Vitis AI sul target (ZCU102)
 - Installazione dell'ambiente e delle librerie di Vitis AI sulla board ZCU102

Applicazione: Signal Traffic Classification

- Desrizione dell'architettura della CNN utilizzata per la Signal Traffic Classification
 - LeNet
 - VGG16
- Implementazione dell'applicazione di Signal Traffic Classification su cloud Kaggle
 - Addestramento della CNN utilizzando PyTorch
 - Valutazione delle prestazioni del modello sul dataset di test su Kaggle
- Deploying della CNN sulla ZCU102
 - Descrizione del flusso di esecuzione di Vitis AI con PyTorch per la ZCU102
 - Quantizer
 - quantizer.py
 - common.py
 - Compiler
 - target.py
 - XIR Graph
 - Preparazione per l'esecuzione del modello quantizzato su DPU
 - Configurazione e caricamento del modello sulla ZCU102
 - Esecuzione dell'applicazione di Signal Traffic Classification sulla ZCU102
- Valutazione dei risultati
 - Analisi delle prestazioni dell'applicazione sulla ZCU102
 - DPU single core
 - DPU multi core
 - Confronto dei risultati ottenuti con quelli dell'implementazione su cloud Kaggle
- Conclusioni

Overview ZCU102 MPSoC

La ZCU102 è una piattaforma di sistema multiprocessore su chip (MPSoC) sviluppata da Xilinx. Si basa sull'architettura Zynq UltraScale+ e offre una combinazione tra il processore ARM e l'FPGA (Field-Programmable Gate Array).

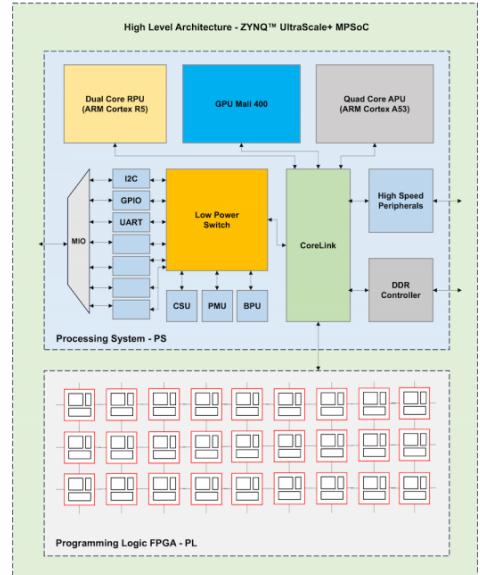
Descrizione dell'architettura Hardware della ZCU102

Dalla Figura, l'architettura della MPSoC ZCU102 è composta da diverse componenti e blocchi funzionali hardware che lavorano sinergicamente per offrire un sistema embedded ad alte prestazioni. La parte del sistema di elaborazione (**Processing System, PS**) comprende una combinazione di processori ARM, GPU e sottosistemi periferici, mentre la logica programmabile (**Programmable Logic, PL**) è implementata sull'FPGA.

La PS include un Dual Core Real-Time Processing Unit (**RPU**) basato su processori **ARM Cortex-R5**, che è dedicato all'esecuzione di task di controllo in tempo reale. Questa parte assicura prestazioni deterministiche e bassa latenza per applicazioni critiche.



ZCU102 e la sua architettura generica ad alto livello



La PS inoltre ospita un Quad Core Application Processing Unit (**APU**) basato su processori **ARM Cortex-A53** a 64 bit. Questo APU gestisce l'elaborazione generale ad alta velocità e offre capacità di calcolo potenti per applicazioni complesse. La ZCU102 è anche dotata di una **GPU Mali-400**, che fornisce accelerazione grafica per applicazioni che richiedono un'elaborazione visiva avanzata.

Nella parte PS ci sono ulteriori componenti utili per l'interazione tra le APU, RPU, GPU, PL e l'esterno come il:

- **Low-Power Switch:** Questo blocco funzionale gestisce la commutazione di potenza e il risparmio energetico all'interno del sistema. Coordinia la gestione dell'alimentazione per garantire un consumo efficiente e ridurre la dissipazione di energia. Nella modalità di funzionamento a basso consumo energetico, viene alimentato un sottoinsieme delle periferiche sulla parte PS, che include: il PMU, l'RPU, il CSU e l'IOU (MIO e le Low Performance Peripherals). In questa modalità, la capacità di modificare la frequenza di sistema consente di regolare la dissipazione di potenza. Tra i blocchi all'interno della modalità a basso consumo energetico, il PLL, il dual Cortex-R5F, le USB e i blocchi di RAM TCM e OCM offrono il power gating. Ad essi sono collegati la CSU (Configuration Security Unit), PMU (Platform Management Unit) e BPU (Battery Power Unit): questi blocchi funzionali gestiscono rispettivamente la sicurezza del sistema, la gestione della piattaforma Hardware e la gestione dell'alimentazione.
- **Low-Performance Peripherals:** Questi sono interfacce di comunicazione a bassa velocità come UART, SPI, I2C, GPIO e altre ancora. Consentono la connessione, pertanto, con periferiche a bassa velocità e supportano le comunicazioni di controllo.
- **Multiplexer Input Output:** Questo blocco funzionale si interfaccia con il mondo esterno e gestisce l'instradamento dei segnali di input e output tra il sistema e le periferiche esterne.

La ZCU102 supporta una vasta gamma di interfacce di comunicazione ad alta velocità, come Gigabit Ethernet, USB 3.0, DisplayPort, HDMI e PCIe. Queste interfacce consentono una connessione rapida e affidabile con altri dispositivi e sistemi, facilitando il trasferimento dei dati ad alta velocità e la comunicazione.

L'FPGA presente nella ZCU102 consente la programmazione e l'implementazione di circuiti logici personalizzati sulla logica programmabile (PL). L'FPGA offre una flessibilità senza pari nell'adattare il sistema alle esigenze specifiche dell'applicazione. Nella configurazione presa in considerazione, l'**FPGA sarà utilizzata per implementare una Deep Learning Processor Unit (DPU) personalizzata**, che fornirà accelerazione hardware dedicata alle reti neurali convoluzionali.

Il sistema include anche sottosistemi periferici dedicati, come controller di memoria, controller di interfaccia e controller di periferiche. Questi sottosistemi consentono una gestione efficiente delle risorse hardware e delle periferiche, nonché l'interazione tra il sistema di elaborazione (PS) e la logica programmabile (PL) all'interno dell'architettura della ZCU102.

Deep Learning Processor Unit (DPU)

La Deep Learning Processor Unit (DPU) rappresenta un'unità di elaborazione programmabile ottimizzata e personalizzata specificamente per l'implementazione di reti neurali deep learning. La DPU è configurabile utilizzando la piattaforma Vivado di Xilinx ed è stata progettata per offrire un'elevata accelerazione alle reti neurali convoluzionali.

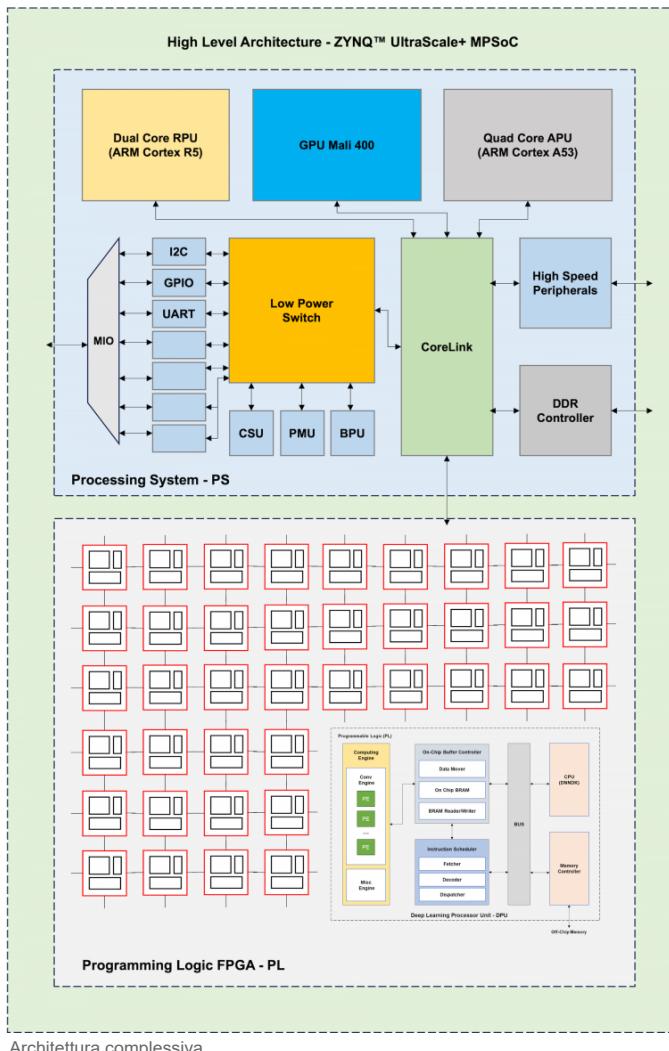
La DPU è **caratterizzata da un insieme di istruzioni altamente ottimizzate**, che consentono l'esecuzione accelerata delle **operazioni tipiche delle reti neurali convoluzionali**, come convoluzioni, pooling e normalizzazione. Questo set di istruzioni specializzate è stato progettato per massimizzare l'efficienza delle reti neurali, riducendo al minimo il consumo energetico e il tempo di elaborazione.

L'architettura hardware della DPU si basa su un **design pipelined**, che permette di ottenere un elevato **livello di parallelismo** e un **throughput superiore** rispetto alla CPU presente sulla board. Gli elementi di elaborazione (**Processing Elements, PE**) presenti all'interno della DPU sono stati progettati per sfruttare al meglio le caratteristiche specifiche dei dispositivi Xilinx, come moltiplicatori, sommatore e accumulatori, per eseguire in modo efficiente le operazioni di calcolo richieste dalle reti neurali.

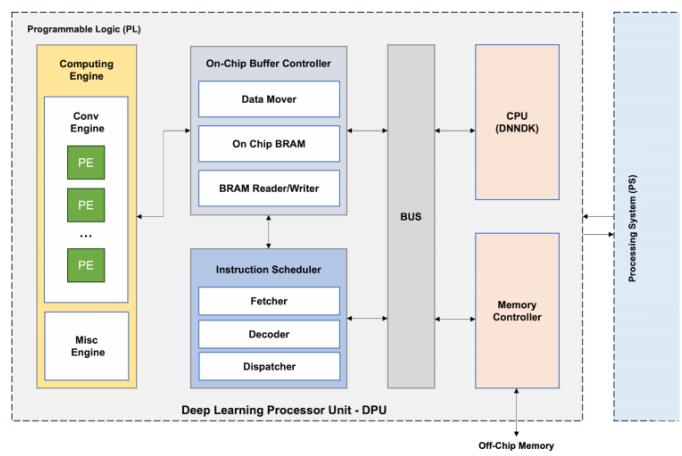
La DPU utilizza una **memoria on-chip (OCM)** per memorizzare temporaneamente i dati di input, intermedi e di output, riducendo così la necessità di accedere frequentemente alla memoria esterna e aumentando l'efficienza complessiva del sistema.

L'architettura della DPU è altamente configurabile e flessibile, in modo da poter essere adattata a diverse applicazioni. Supporta una vasta gamma di architetture grazie al framework **DNNDK (CPU)**, come *VGG*, *ResNet*, *GoogleNet*, *YOLO*, *SSD*, *MobileNet* e altre, garantendo la compatibilità e l'efficienza con le reti neurali ampiamente utilizzate nel campo della Computer Vision.

Successivamente si vedrà la **configurazione e installazione completa della DPU sulla board**.



Architettura complessiva



Architettura generica della DPU

Overview dell'applicazione: Traffic Sign Recognition System

Attraverso questa sperimentazione, si mira a dimostrare l'applicabilità della ZCU102 in un contesto reale, mostrando la sua capacità di supportare applicazioni di riconoscimento dei segnali stradali tramite un modello di rete neurale convoluzionale.

La sperimentazione condotta si focalizza sull'implementazione di un sistema in grado di riconoscere automaticamente segnali stradali come indicazioni di svolta a sinistra o a destra, limiti di velocità, divieto di sorpasso per veicoli pesanti, divieto di accesso, attraversamento pedonale per bambini e altri simboli correlati. L'obiettivo principale è garantire che i veicoli possano adottare le adeguate precauzioni in risposta a tali segnali.

Per raggiungere questo obiettivo, si svilupperà un modello di rete neurale convoluzionale utilizzando la libreria **PyTorch**.

Come già detto in precedenza, al fine di valutare possibili variazioni delle prestazioni, il modello sarà inizialmente eseguito sul cloud utilizzando l'infrastruttura fornita da *Kaggle*, sfruttando la potenza computazionale della NVIDIA TESLA GPU P100. Successivamente, il modello sarà implementato sulla board ZCU102. **Saranno eseguiti confronti e analisi per determinare eventuali differenze nelle prestazioni quando il modello viene eseguito sulla Deep Learning Processing Unit (DPU) della ZCU102 rispetto all'esecuzione sul Cloud.**

II Dataset

Il dataset utilizzato per questo progetto è il *Traffic Signs Classification*, un ampio database di segnali stradali suddivisi in 43 classi, di cui le immagini sono state opportunamente ritagliate. Questo dataset è stato acquisito dal *German Traffic Sign Recognition Benchmark* (<https://benchmark.ini.rub.de/>) e presenta le seguenti caratteristiche:

- Oltre 40 classi di segnali stradali
- Più di 50.000 immagini in totale (dimensione dell'immagine 32x32)
- Un database ampio e realistico

Il dataset è stato scelto per la sua ampiezza e per la sua somiglianza alla realtà, offrendo una rappresentazione significativa dei diversi segnali stradali che si possono incontrare nel contesto reale. Questo permette al modello di essere addestrato su un ampio spettro di situazioni e di acquisire una comprensione accurata dei segnali stradali, migliorando così la sua capacità di riconoscimento e classificazione.



Dataset

PetaLinux

PetaLinux è un ambiente di sviluppo open-source basato su Linux, progettato per facilitare la creazione di **sistemi operativi Linux personalizzati e su misura per l'esecuzione su dispositivi embedded**, come **FPGA** (Field-Programmable Gate Array) e **SoC** (System-on-Chip). È sviluppato da Xilinx, una società specializzata in tecnologie di semiconduttori programmabili.

Introduzione: importanza nel contesto di sviluppo su ZCU102

PetaLinux offre agli sviluppatori una serie di strumenti e librerie precompilate, nonché una vasta gamma di componenti software pronti all'uso per la configurazione del kernel Linux, la creazione del sistema radice e la generazione delle immagini di avvio. Utilizzando PetaLinux, gli sviluppatori possono personalizzare e configurare il sistema operativo Linux in base alle specifiche del proprio dispositivo embedded, includendo solo i componenti necessari per ridurre la dimensione del sistema e ottimizzarne le prestazioni.

L'ambiente di sviluppo PetaLinux si basa su Yocto Project, un framework open-source per la creazione di distribuzioni Linux embedded. Utilizzando Yocto Project come base, PetaLinux aggiunge una serie di strumenti, script e patch specifici di Xilinx per semplificare il processo di sviluppo per i dispositivi Xilinx.

PetaLinux è composto da diversi strumenti utilizzabili tramite interfaccia a linea di comando (**CLI**), in dettaglio, esso contiene i seguenti comandi:

- **petalinux-create**: Questo comando viene utilizzato per **creare un nuovo progetto PetaLinux**. Si specifica il nome del progetto e viene creata la struttura delle directory di base per il progetto, inclusi i file di configurazione iniziali;
- **petalinux-config**: Questo comando viene utilizzato per **configurare il progetto PetaLinux**. Consente di selezionare le opzioni di configurazione del kernel Linux, personalizzare i driver dei dispositivi, aggiungere o rimuovere pacchetti software, configurare le impostazioni del file system e molto altro ancora. Viene utilizzato un menu di configurazione testuale o una GUI a finestre per guidare l'utente attraverso le opzioni disponibili;
- **petalinux-build**: Questo comando viene utilizzato per **compilare il progetto PetaLinux**. Esegue la compilazione del kernel Linux, dei driver dei dispositivi, delle applicazioni utente e di altri componenti software definiti nel progetto. Vengono generate le *immagini di avvio, i file binari e altri artefatti del sistema operativo*;
- **petalinux-util**: Questo comando è un'utility che fornisce una serie di funzionalità ausiliarie per lo sviluppo di progetti PetaLinux. Può essere utilizzato per eseguire diverse operazioni, come l'installazione di componenti aggiuntivi, l'aggiornamento di PetaLinux, la pulizia del progetto e la generazione di script di configurazione;
- **petalinux-package**: Questo comando viene utilizzato per creare un pacchetto del sistema operativo PetaLinux. Consente di creare un file di pacchetto che può essere utilizzato per la distribuzione del sistema operativo su altri dispositivi embedded. Il pacchetto include il kernel Linux, il file system radice e altri componenti necessari per l'esecuzione del sistema;
- **petalinux-upgrade**: Questo comando viene utilizzato per aggiornare un progetto PetaLinux esistente alla versione successiva di PetaLinux. Consente di migrare il progetto alle ultime funzionalità e miglioramenti forniti da Xilinx;
- **petalinux-devtool**: Questo comando viene utilizzato per creare e gestire componenti software aggiuntivi all'interno di un progetto PetaLinux. Consente di aggiungere, rimuovere o modificare pacchetti software, nonché di generare strumenti di sviluppo personalizzati per il progetto;
- **petalinux-boot**: Questo comando viene utilizzato per avviare il sistema operativo PetaLinux sul dispositivo embedded di destinazione. Consente di **avviare il kernel Linux**, caricare il file system radice e avviare il sistema nel dispositivo target.

Configurazione di PetaLinux per il progetto in esame

Per l'installazione dei tools di PetaLinux sono richieste le seguenti specifiche HW/SW sulla macchina host:

- 8 GB RAM (requisito minimo richiesto dai tools Xilinx®)
- 2 GHz CPU clock o equivalente (requisito desiderato di 8 cores)
- 100 GB di spazio libero su HDD
- OS supportati:
 - Red Hat Enterprise Workstation/Server 7.4, 7.5, 7.6, 7.7, 7.8, 7.9, 8.1, 8.2 (64-bit)
 - CentOS Workstation/Server 7.4, 7.5, 7.6, 7.7, 7.8, 7.9, 8.1, 8.2 (64-bit)
 - Ubuntu Linux Workstation/Server 16.04.5, 16.04.6, 18.04.1, 18.04.2, 18.04.3, 18.04.4, 18.04.5, 20.04, 20.04.1 (64-bit)

Nel caso in esame, le specifiche dell'host **utilizzato** per eseguire PetaLinux sono:

- 16 GB RAM
- 2.8 GHz CPU base clock (4 cores - 8 threads)
- 300 GB di spazio libero su SSD
- OS: Ubuntu 20.04 eseguito su VM WSL2

Al fine di effettuare l'installazione di PetaLinux, occorre scaricare il file **installer** dell'ambiente dal sito ufficiale di Xilinx ([link per il download \(https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools/archive.html\)](https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools/archive.html)), questo file che permette l'installazione dell'ambiente di sviluppo di PetaLinux prende il nome in base alla versione scelta, in questo caso la 2021.1, per procedere all'installazione dell'ambiente di sviluppo si eseguono i seguenti comandi:

```
chmod 755 ./petalinux-v2021.1-final-installer.run (scaricato dal link precedente)
./petalinux-v2021.1-final-installer.run --dir <installation_dir>
```

Dopo aver installato PetaLinux, all'interno della directory `<installation_dir>`, nel caso in esame chiamata `petalinux`, saranno presenti le seguenti cartelle e file:

```
giuseppericcio@giuseppericcio:~$ ls
petalinux                               xilinx-zcu102-2021.1
petalinux-v2021.1-final-installer.run    xilinx-zcu102-v2021.1-final.bsp
petalinux-v2021.1-final-installer.run:Zone.Identifier xilinx-zcu102-v2021.1-final.bsp:Zone.Identifier
petalinux_installation_log
giuseppericcio@giuseppericcio:~$ cd petalinux/
giuseppericcio@giuseppericcio:~/petalinux$ ls
components doc etc settings.csh settings.sh tools
giuseppericcio@giuseppericcio:~/petalinux$ |
```

Installazione PetaLinux

Il passo successivo per l'installazione prevede di aggiungere il PATH di PetaLinux alle variabili di ambiente dell'OS dell'host tramite il comando:

```
source <installation_dir>/settings.sh
```

Se l'installazione è andata a buon fine, eseguendo il comando `echo $PETALINUX` dovrebbe comparire il percorso all'interno del quale sono stati installati i tools di PetaLinux, si riporta per il caso in esame il risultato ottenuto:

```
giuseppericcio@giuseppericcio:~$ echo $PETALINUX
/home/giuseppericcio/petalinux
```

PetaLinux

Creazione del progetto

Per la creazione del progetto in PetaLinux è necessario scaricare il file del Board Support Package (BSP) al medesimo link riportato in precedenza per scaricare l'installer. Questo file è strettamente legato alla scheda di sviluppo o piattaforma hardware per cui si intende creare il progetto PetaLinux, in questo caso la Zynq Ultrascale+ MPSoC (versione ZCU102), al suo interno questo file include tutti i pacchetti di progettazione e configurazione necessari per supportare il corretto funzionamento del sistema operativo Linux su quella specifica piattaforma hardware. Il BSP viene fornito dal produttore della scheda di sviluppo e comprende diversi elementi essenziali, tra cui:

- File di configurazione hardware:** Il BSP contiene informazioni dettagliate sulla configurazione hardware della scheda di sviluppo;
- Driver dei dispositivi:** Il BSP include i driver dei dispositivi necessari per il corretto funzionamento delle periferiche integrate sulla scheda di sviluppo, come porte GPIO, interfacce di comunicazione (UART, I2C, SPI), controller Ethernet, acceleratori hardware, ecc;
- Root file system:** Il BSP può includere un file system radice preconfigurato, che fornisce un ambiente di esecuzione di base per il sistema operativo Linux;
- Script e file di configurazione:** Il BSP può includere script e file di configurazione aggiuntivi specifici della piattaforma, ad esempio per il boot dell'ambiente di emulazione QEMU.

Una volta scaricato, il file relativo al BSP, si può procedere alla creazione del progetto tramite il seguente comando:

```
petalinux-create -t project -s ./xilinx-zcu102-v2021.1-final.bsp
```

Se il progetto è stato creato correttamente, all'interno della directory da cui si è dato il comando sarà presente una cartella con lo stesso nome del file BSP, a meno del suffisso `-final.bsp`, il cui contenuto è il seguente:

```
giuseppericcio@giuseppericcio:~$ cd xilinx-zcu102-2021.1/
giuseppericcio@giuseppericcio:~/xilinx-zcu102-2021.1$ ls
README README_hw build components config.project hardware images pre-built project-spec
giuseppericcio@giuseppericcio:~/xilinx-zcu102-2021.1$ |
```

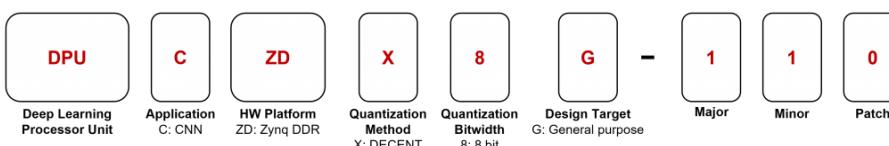
Creazione del progetto PetaLinux

Configurazione dell'HW: la DPU

Essendo che l'obiettivo di questo progetto è quello di eseguire una rete neurale sulla ZCU102, si è deciso, di utilizzare la DPU per efficientare le operazioni necessarie a tale scopo, quali convoluzioni, pooling e normalizzazione, come già detto in precedenza.

In particolare, si è partiti dall'implementazione fornita da Xilinx per la DPU della ZCU102, che è possibile individuare tramite l'opportuna sigla definita da Xilinx stessa con la seguente convenzione:

DPU Naming Scheme



DPU Naming Scheme

Una volta individuata la DPU corretta, si può procedere alla modifica del suo IP core sulla base delle esigenze specifiche del progetto, infatti, tra i file della DPU forniti da Xilinx è presente uno script **TCL** (Tool Command Language) che fornisce alcuni parametri configurabili dall'utente al fine di **ottimizzare l'utilizzo di risorse e personalizzare diverse funzioni**. Ad esempio, possono essere selezionate diverse configurazioni d'utilizzo per le slice DSP, LUT, block RAM(BRAM), e UltraRAM basate sulla quantità di risorse logiche programmabili disponibili. Ci sono anche opzioni per aggiungere funzioni, come channel augmentation, average pooling, depthwise convolution. Nel caso della DPU della ZCU102, nello script `trd_prj.tcl` fornito da Xilinx, possono essere configurati i seguenti parametri:

```

*****
# set param
*****
dict set dict_prj dict_param DPU_CLK_MHz {325}
dict set dict_prj dict_param DPU_NUM {2}
dict set dict_prj dict_param DPU_ARCH {4096}
dict set dict_prj dict_param DPU_RAM_USAGE {low}
dict set dict_prj dict_param DPU_CHN_AUG_ENA {1}
dict set dict_prj dict_param DPU_DWCV_ENA {1}
dict set dict_prj dict_param DPU_ELEW_MULT_ENA {0}
dict set dict_prj dict_param DPU_AVG_POOL_ENA {1}
dict set dict_prj dict_param DPU_CONV_RELU_TYPE {3}
dict set dict_prj dict_param DPU_SFm_NUM {1}
dict set dict_prj dict_param DPU_DSP48_USAGE {high}
dict set dict_prj dict_param DPU_URAM_PER_DPU {0}

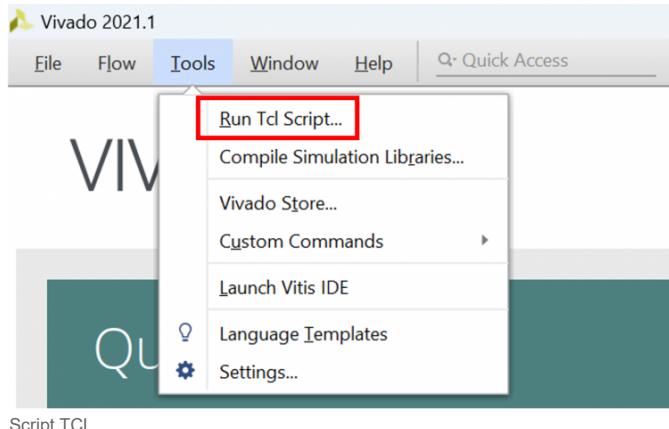
dict set dict_prj dict_param REG_CLK_MHz {100}
dict set dict_prj dict_param HP_CLK_MHz {334}
dict set dict_prj dict_param CN_STGY {flat}
dict set dict_prj dict_param DPU_IP_FOLDER {dpu}
dict set dict_prj dict_param DPU_SAXICLK_INDDPD {1}
dict set dict_prj dict_param DPU_CLK_GATING_ENA {1}
dict set dict_prj dict_param DPU_DSP48_MAX_CASC_LEN {4}
dict set dict_prj dict_param DPU_TIMESTAMP_ENA {1}

Configurazione DPU IP

```

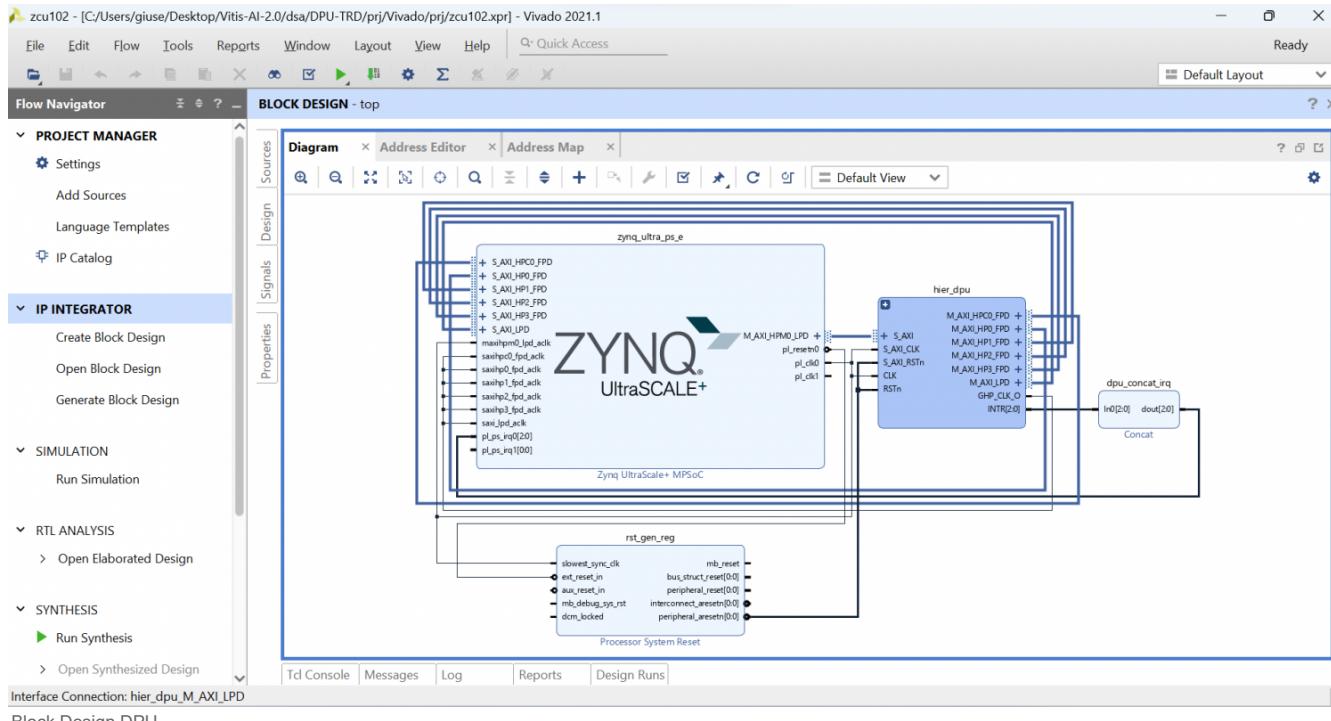
Tra i diversi parametri configurabili, per il progetto in questione si è deciso di provare diverse configurazione della DPU rispetto al numero di core utilizzati nella sua architettura, ovvero, si è agito sul parametro **DPU_NUM**, il quale di default è settato a 2 come è possibile notare dalla Figura, ma tale numero può essere cambiato fino ad un massimo di 4 cores. Volendo effettuare delle comparazioni sulle prestazioni della DPU in termini di throughput, si è scelto di selezionare una configurazione con 1 core ed una con 3 cores, si è evitato di settare il numero di core pari a 4 perchè la DPU utilizza in modo intensivo le LUTs e le RAMs. Quindi, usare 4 DPU cores potrebbe causare problemi rispetto alle risorse e ai tempi di risposta.

Una volta configurato opportunamente lo script TCL, attraverso **Vivado Design Suite** è possibile importare tale script che oltre a configurare la DPU permette tra l'altro di automatizzare e orchestrare le attività di progettazione all'interno dell'ambiente Vivado. Questo script permette di eseguire un insieme di comandi predefiniti in sequenza senza la necessità di interazione diretta con l'interfaccia utente grafica di Vivado, tra i comandi eseguibili si trovano quelli per la creazione del progetto, l'importazione di file di descrizione del circuito (in VHDL o Verilog), la sintesi e ottimizzazione, l'implementazione e routing, la simulazione e verifica ed infine la generazione di bitstream (in formato .xsa). Per eseguire lo script TCL all'interno di Vivado Design Suite è necessario selezionare dal menu *Tools>Run Tcl Script*, come mostrato in Figura, e selezionare il file salvato in precedenza con i parametri dell'IP core della DPU, successivamente occorre cliccare su Ok.



Script TCL

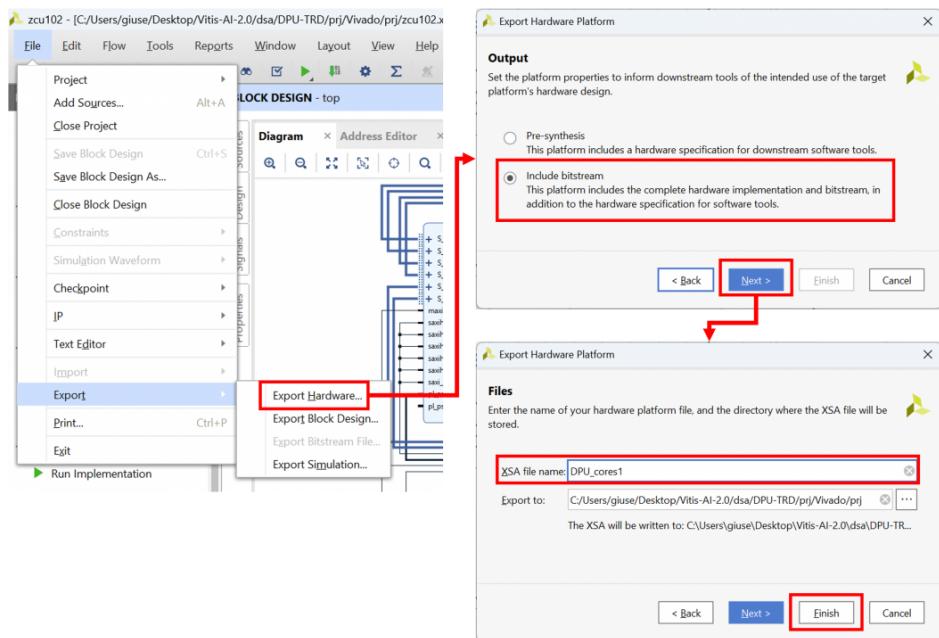
Una volta eseguito lo script, all'interno di Vivado si aprirà il Block Design che mostra l'interconnessione tra la parte PS della Zynq Ultrascale+ e la DPU (implementata sulla parte PLI), come mostrato nella Figura seguente.



Il passaggio successivo è quello di generare il **bitstream** relativo alla configurazione dell'**FPGA** (Field-Programmable Gate Array) su cui è stata implementata la DPU, che contiene le informazioni sulla configurazione dei componenti logici, la connettività tra di essi e le impostazioni di timing. Il bitstream viene generato attraverso il processo di sintesi e implementazione che a partire dal codice sorgente (ad esempio in VHDL o Verilog) viene tradotto in una rappresentazione hardware e quindi mappato e instradato all'interno del dispositivo target (la ZCU102). In Vivado, per generare il bitstream occorre cliccare sull'opzione *Generate Bitstream*.

Si noti: Per generare il bitstream di Vivado per la DPU in questione essendo presenti dei componenti proprietari di Xilinx è necessaria una licenza di tipo commerciale.

Al termine della generazione del bitstream si deve esportare la **piattaforma HW** che servirà nella **fase di build di PetaLinux** per integrare all'interno dell'immagine Linux del progetto tale architettura. In Vivado per esportare la configurazione HW si procede come nella seguente Figura.



Esportazione XSA della DPU

Dopo aver esportato la piattaforma HW, all'interno della cartella del progetto troveremo il file *DPU_cores1.xsa*, inoltre, all'interno della cartella *srcs/* generata da Vivado durante la fase di generazione del bitstream navigando tra le directory, in particolare, andando nella cartella *top/ip/top_DPU_CZDX8G_O* è possibile notare la presenza di un file *.json* (il file è denominato *arch.json*) che è **necessario al tool di Vitis AI per effettuare la compilazione del modello CNN rispetto alla specifica configurazione della DPU**.

Importazione della configurazione HW e build del progetto

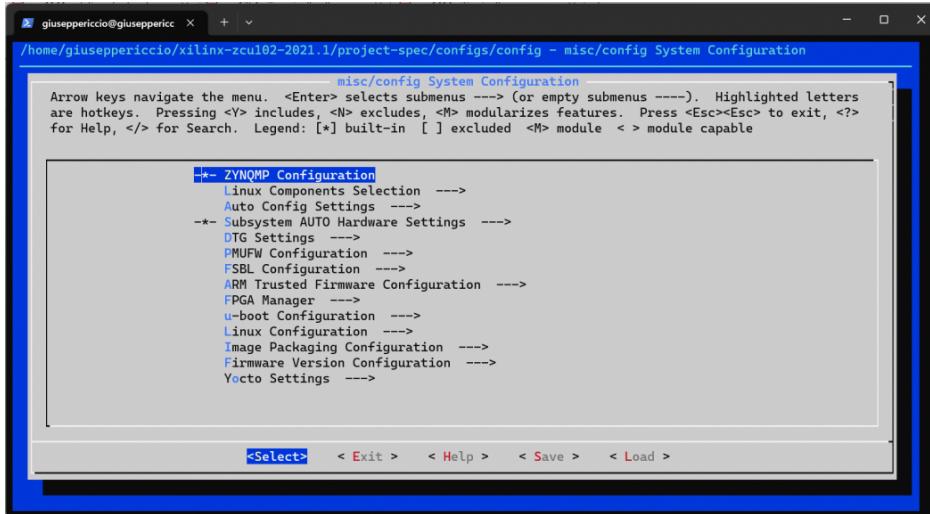
L'ultimo step prima della creazione dell'immagine di boot Linux per la ZCU102 è quello che prevede l'importazione della configurazione HW della DPU generata tramite Vivado Design Suite e la build del kernel Linux. All'interno della cartella relativa al progetto PetaLinux creato in precedenza per importare la configurazione HW bisogna eseguire il seguente comando:

```
petalinux-config --get-hw-description <PATH-TO-XSA Directory>/<PATH-TO-XSA>
```

Dove al posto di <PATH-TO-XSA Directory>/<PATH-TO-XSA> occorre inserire il percorso all'interno del quale si trova il file .xsa esportato nel paragrafo precedente tramite Vivado. Tuttavia, essendo che per il seguente progetto si è prevista l'implementazione della DPU in 2 diverse configurazioni, i file .xsa sono 2. Per questo motivo non si utilizza il comando precedente bensì il seguente:

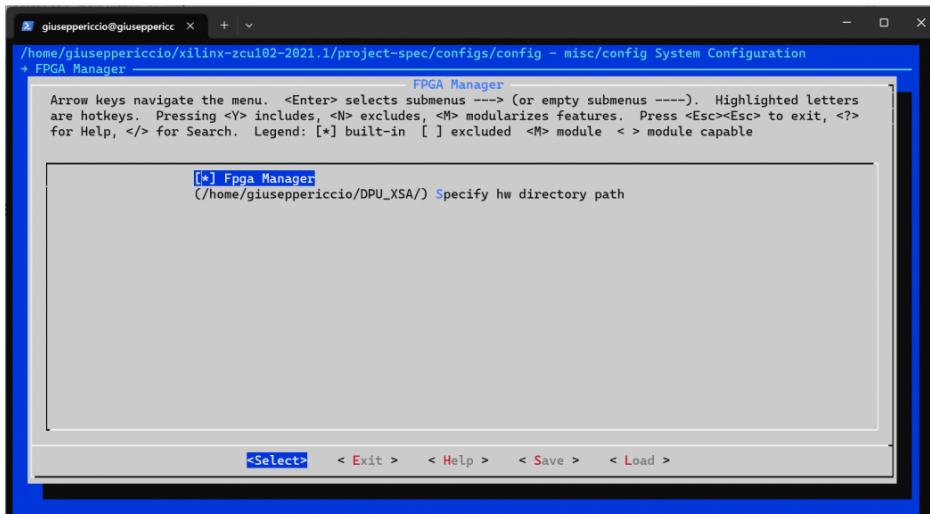
```
petalinux-config
```

Se il progetto è stato creato correttamente all'interno della bash Linux si aprirà la seguente schermata di configurazione del sistema:



Menu di System Configuration

All'interno del precedente menu si configurano, in particolare, due impostazioni necessarie per la corretta creazione dell'immagine di boot, la prima impostazione riguarda l'**FPGA Manager** che fornisce un'interfaccia verso Linux per la configurazione della parte PL (Programmable Logic) della Zynq Ultrascale+. Questo manager impacchetta i file .xsa all'interno della cartella `/lib/firmware/xilinx` del root file system di PetaLinux **generando** i corrispondenti file `.dtbo` e `.bin`. In particolare, occorre abilitare FPGA Manager ed indicare il path all'interno del quale sono presenti le diverse specifiche HW in formato .xsa, come mostrato di seguito.



FPGA Manager

La seconda impostazione riguarda la disabilitazione dell'uso di tftpboot, il quale genera diversi errori in fase di boot, non trovando il server **TFTP** (Trivial File Transfer Protocol) tramite cui effettuare il trasferimento dei file tra host e target sulla rete. Per disabilitarlo occorre deselezionare l'opzione *Copy final images to tftpboot* all'interno di **Image Packaging Configuration** come mostrato in Figura:

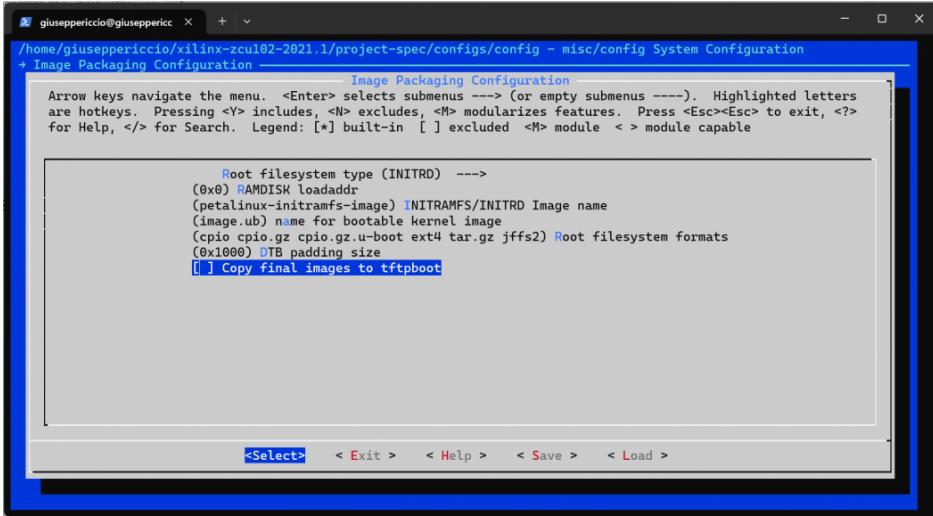


Image Packaging Configuration

Dopo aver configurato il sistema con gli opportuni parametri si passa alla configurazione dei parametri del kernel, in particolare tramite il seguente comando:

```
petalinux-config -c kernel
```

Verrà aperta una schermata di configurazione molto simile alla precedente, in cui sarà possibile abilitare le opportune **impostazioni del kernel Linux**, in dettaglio la configurazione finale sarà la seguente:

Device Drivers > FPGA Configuration Framework

- FPGA debug fs**
- Altera Partial Reconfiguration IP Core
- Altera FPGA Passive Serial over SPI
- Altera CvP FPGA Manager
- Xilinx Configuration over Slave Serial (SPI)
- Lattice ICE40 SPI
- Lattice MachXO SPI
- Xilinx AFI FPGA
- FPGa Bridge Framework
 - Altera FPGA Freeze Bridge
 - Xilinx LogiCORE PR Decoupler
 - FPGA Region
 - FPGA Region Device Tree Overlay Support
 - FPGA Device Feature List (IFL) support**
 - Xilinx ZynqMP FPGA**
 - Xilinx Versal FPMU

Memory Management Options

- Memory model (Sparse Memory) →**
 - Sparse memory virtual memmap
 - Allow for memory hot-add
 - Online the newly added memory blocks by default
 - Allow for memory hot remove
 - Allow for memory compaction
 - Free page reporting
 - Page migration
 - Enable KSM for page merging (32768) Low address space to protect from user allocation
 - Enable recovery from hardware memory errors
 - Transparent Hugepage Support**
 - Transparent Hugepage Support sets defaults (madvise) →
 - Enable cleancache driver to cache clean pages if tmem is present
 - Enable frontswap to cache swap pages if tmem is present**
 - Contiguous Memory Allocator**
 - DMA debug messages (DEVELOPMENT)
 - DMA debugfs interface
 - (7) Maximum count of the CMA areas
 - Common API for compressed memory storage
 - Low (Up to 2x) density storage for compressed pages
 - Memory allocator for compressed pages
 - Defers initialisation of struct pages to kthreads
 - Enable idle page tracking
 - Collect percpu memory statistics
 - Enable infrastructure for get_user_pages() and related calls benchmarking
 - Read-only TMR for filesystems (EXPERIMENTAL)

Device Drivers > Device Tree and Open Firmware support

- Device Tree and Open Firmware support**
 - Device Tree runtime unit tests**
 - Device Tree Overlay ConfigFS interface**

Library Routines

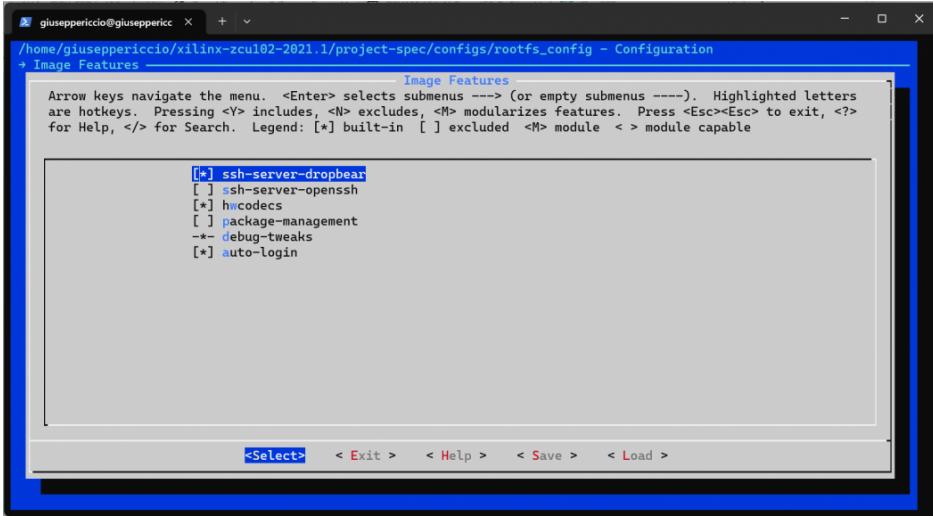
- Automatically choose fastest RNDNG PR functions**
- Generic bitfield packing and unpacking
- CRC32 algorithm
- Simple prime number generator for testing
- Access I/O in non-MMIO mode
- CRC-CCITT functions
- CRC16 functions
- CRC calculation for the T10 Data Integrity Field
- CRC32C/CRC32 functions
- CRC32 perform self test on init
- CRC32 implementation (Slice by 8 bytes) →
 - CRC64 functions
 - CRC4 functions
 - CRC7 functions
 - CRC32 (Castagnoli, et al) Cyclic Redundancy-Check**
 - CRC8 function
 - PRNG perform self test on init
- XZ decompression support
 - x86 BCJ filter decoder
 - PowerPC BCJ filter decoder
 - IA-64 BCJ filter decoder
 - ARM BCJ filter decoder
 - ARM-Thumb BCJ filter decoder
 - SPARC BCJ filter decoder
- zlib decompression support**
 - IMHA Contiguous Memory Allocator**
 - IMHA provides contiguous memory area for each NUMA Node
 - *** Default contiguous memory area size: ***
 - (256) Size in Mega Bytes
 - Selected region size (Use mega bytes value only) →
 - (8) Maximum PAGE_SIZE order of alignment for contiguous buffers
 - Enable debugging of DMA-API usage
 - glob self-test.on.init
 - IRQ polling library
 - Select compiled-in fonts
 - Test string functions

Configurazione del kernel Petalinux

Infine, si passa alla configurazione del root file system al fine di aggiungere alcuni software di utilità, come OpenCV, Python, ecc. Oltre ad abilitare la funzione di *auto-login* al fine di evitare che al boot di Linux occorra inserire la password dell'utente root. Per fare ciò occorre eseguire il comando:

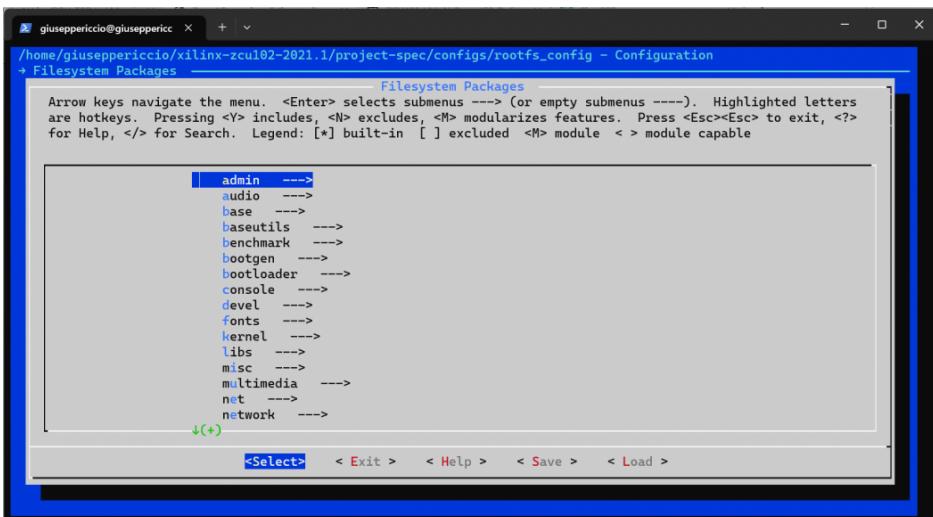
```
petalinux-config -c rootfs
```

Se il comando esegue correttamente, verrà aperto il menu di configurazione mostrato in Figura, all'interno della sezione *Image Features* viene abilitata l'opzione *auto-login*.



Configurazione del RootFS PetaLinux - Image Features

Per quanto riguarda l'inclusione delle librerie, attraverso la sezione *Filesystem Packages* è possibile abilitare nelle varie categorie gli strumenti da importare nel sistema Linux.



Configurazione del RootFS PetaLinux - Filesystem Packages

A questo punto è possibile generare l'FSBL, il firmware, il device tree (file DTB), l'immagine di U-Boot, l'eseguibile del kernel ed il root file system (il quale comprende i bitstream relativi alle implementazioni della DPU su FPGA). Per fare ciò si esegue il seguente comando:

```
petalinux-build
```

Al termine del build (questa fase impiega circa 2 ore), all'interno della cartella *images/linux/* è possibile osservare le **immagini di boot** generate.

Creazione dell'immagine di boot

L'immagine di boot generata dalla fase di build può essere **caricata** all'interno di una **SD card** e inserirla nella rispettiva board. In questo modo quando si accende il target (ZCU102), esso può avviare l'immagine di boot. Tuttavia, per fare ciò bisogna fare il packaging dell'immagine di boot in formato *.BIN*, in questo file vengono unite l'immagine di FSBL, i bitstream della DPU, il firmware e l'immagine di U-Boot. Il comando usato è:

```
petalinux-package --boot --u-boot --format BIN
```

Se il comando viene eseguito correttamente, all'interno della cartella *images/linux/* sarà adesso presente anche un file denominato *BOOT.BIN*.

Vitis AI

Immediatamente sopra al DPU Layer c'è il comparto software Vitis AI. Vitis AI è un ambiente di sviluppo integrato che accelera e supporta l'inferenza dell'intelligenza artificiale su piattaforme hardware Xilinx, tra cui proprio la ZCU102, utilizzando core IP ottimizzati, strumenti, librerie, modelli e progetti di esempio. La sua progettazione enfatizza l'alta efficienza e la facilità d'uso per sfruttare appieno il potenziale della DPU descritta in precedenza. Vitis AI semplifica lo sviluppo di applicazioni di inferenza deep learning **astraendo le complessità sottostanti della tecnologia FPGA**. Per garantire l'esecuzione corretta del modello, Vitis AI fornisce supporto per la quantizzazione e la calibrazione dei modelli. Inoltre, Xilinx offre un ottimizzatore AI opzionale in grado di ridurre le dimensioni di un modello fino al 90% con una tollerabile perdita di precisione. L'ambiente offre anche API ad alto livello in C++ e Python per una massima portabilità dall'Edge al Cloud e viceversa.

Componenti fondamentali di Vitis AI

Alla base c'è il kit di sviluppo necessario per il processo di supporto completo del modello di deep learning scelto (Vitis AI Development Kit), i modelli Vitis AI e i modelli compatibili con l'ambiente in caso di modelli personalizzati e la User Application.

Nel caso in esame, per la sperimentazione si è fatto uso di un **modello customizzato dagli autori** e non si è fatto uso del Model Zoo, ovvero un repository di modelli di deep learning predefiniti, ottimizzati per l'esecuzione su piattaforme hardware Xilinx. Il Model Zoo contiene modelli per vari casi d'uso, come riconoscimento di immagini ed altre applicazioni di Computer Vision.

Per quanto riguarda il Development Kit di Vitis AI, le componenti in breve sono:

- **VART (Vitis AI Runtime)**: consente alle applicazioni di utilizzare le API runtime per astrarre ad alto livello i dettagli delle tecnologie HW, come la DPU, rendendo le implementazioni Cloud-to-Edge efficienti e rapide.
- **AI Profiler**: è uno strumento di profiling che consente di analizzare le prestazioni dell'inferenza AI sul dispositivo hardware.
- **AI Library**: è una libreria di funzioni software predefinite per la creazione di applicazioni di inferenza AI.
- **AI Quantizer**: è uno strumento che supporta la quantizzazione dei modelli, ovvero la riduzione della precisione dei dati in ingresso per migliorare l'efficienza dell'inferenza AI sulla base della board target.
- **AI Compiler**: è uno strumento che consente di compilare il codice sorgente del modello di deep learning in un formato eseguibile compatibile con la piattaforma hardware Xilinx.
- **AI Optimizer**: è uno strumento di ottimizzazione che può ridurre le dimensioni del modello fino al 90% con una tollerabile perdita di precisione. L'AI Optimizer utilizza una combinazione di tecniche di pruning, quantizzazione e compressione per ridurre la dimensione dei modelli.

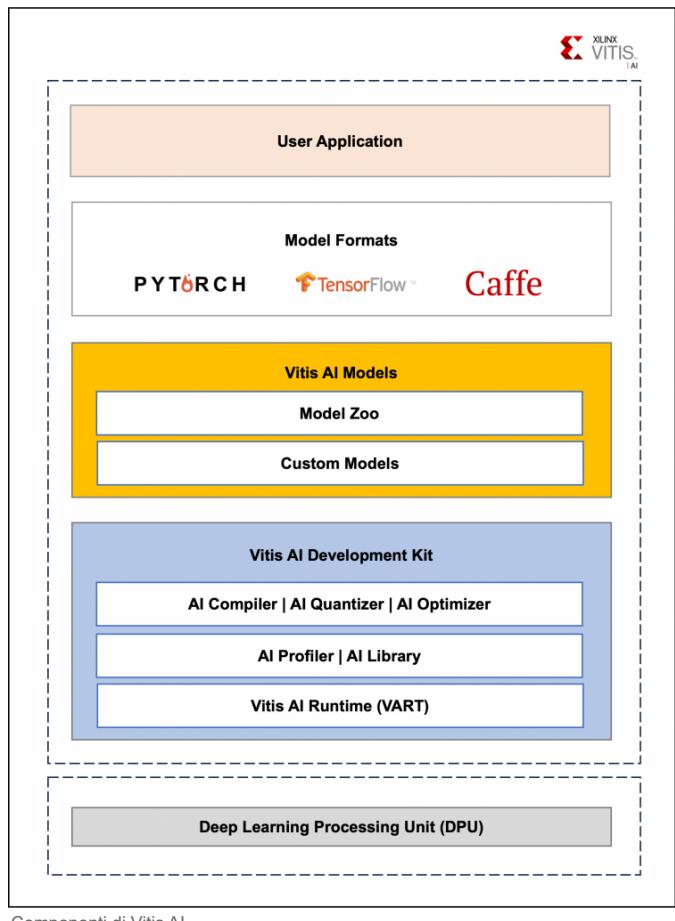
In particolare si è fatto uso dell'**AI Quantizer** e dell'**AI Compiler** per la sperimentazione in questione.

Vitis AI Runtime (VART)

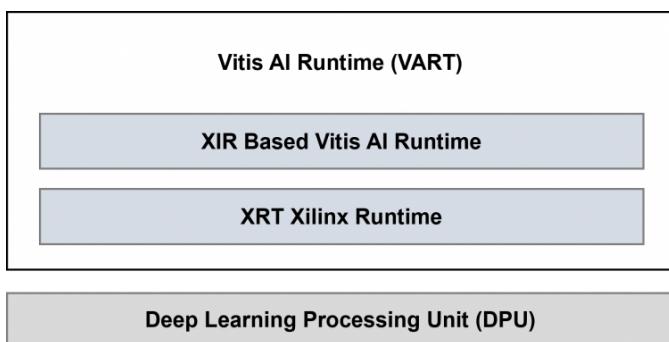
Vitis AI Runtime (VART) è un runtime di intelligenza artificiale sviluppato da Xilinx come parte della suite di strumenti Vitis AI per l'elaborazione AI su piattaforme hardware Xilinx. Il VART consente di eseguire l'elaborazione AI sulla DPU (Deep Learning Processing Unit) installata sulla board.

Il VART fornisce un'API di runtime ad alto livello semplificando la migrazione di applicazioni AI sulla DPU, consentendo agli sviluppatori di utilizzare strumenti familiari come C++ e Python. Inoltre, il VART presenta una serie di funzionalità e caratteristiche per **ottimizzare l'elaborazione AI sulla DPU**, tra cui la presentazione asincrona di lavori all'acceleratore, la raccolta asincrona di lavori dall'acceleratore e il supporto per l'esecuzione multithreading e multi-processo.

Il VART è basato su XRT (Xilinx Runtime), che fornisce un'infrastruttura di runtime comune per l'elaborazione di diversi carichi di lavoro su piattaforme hardware Xilinx, inclusa la DPU. Inoltre, il VART utilizza XIR (Xilinx Intermediate Representation), una rappresentazione intermedia di Xilinx che consente di ottimizzare il modello AI per l'esecuzione sulla DPU.



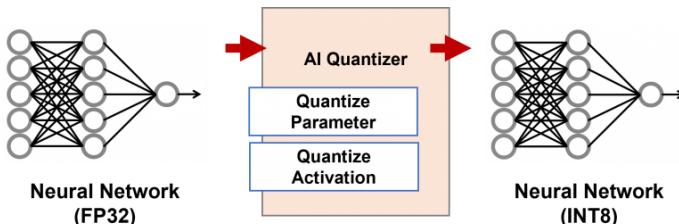
Componenti di Vitis AI



Vitis AI Runtime

AI Quantizer

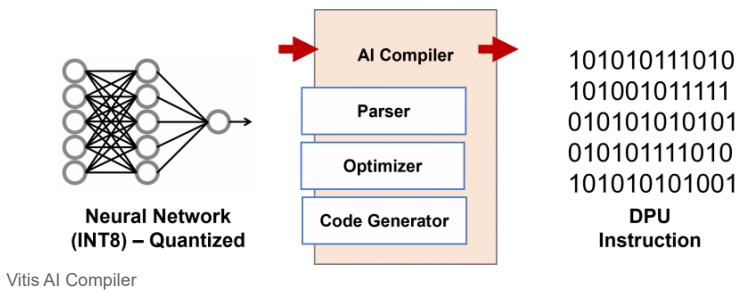
Il quantizzatore di Vitis AI consente di **ridurre la complessità di calcolo** senza compromettere la precisione della predizione, convertendo i pesi e le attivazioni in virgola mobile a 32 bit in formati a virgola fissa come INT8. Il modello di rete a virgola fissa richiede una quantità inferiore di banda di memoria, offrendo quindi velocità più elevate e maggiore efficienza energetica rispetto al modello a virgola mobile.



Vitis AI Quantizer

AI Compiler

Il compilatore di Vitis AI mappa il modello AI in istruzioni DPU e lo rende eseguibile sulla DPU.



Configurazione dell'host

Sull'host viene simulato l'ambiente PetaLinux su QEMU, al fine di **verificare se l'immagine è stata generata correttamente** prima di effettuarne il flash sulla SD card della Zynq UltraScale+ MPSoC. Inoltre, viene effettuata la **fase di quantizzazione e compilazione del modello CNN** da eseguire sulla DPU tramite i tools di **Vitis AI**.

In particolare, l'host è un PC con le seguenti caratteristiche HW:

- Intel i7-1165G7 con frequenza base del clock di 2.8 GHz
- 16 GB RAM DDR4
- 512 GB SSD NVMe
- GPU NVIDIA GeForce MX450

Mentre il SO è Ubuntu 20.04 LTS che gira sulla VM WSL2.

Utilizzo di PetaLinux sull'host

Il simulatore PetaLinux su **QEMU** (Quick EMULATOR) è un ambiente di sviluppo e di simulazione che consente di eseguire e testare il sistema operativo PetaLinux su un emulatore software, in questo caso la macchina host. Questo strumento è particolarmente utile per diversi scopi:

- **Sviluppo del software:** Il simulatore PetaLinux su QEMU consente agli sviluppatori di testare e validare il proprio software senza dover disporre di hardware fisico;
- **Debugging:** Il simulatore offre un ambiente controllato per il debugging del software. Gli sviluppatori possono eseguire il codice all'interno del simulatore, interrompere l'esecuzione per individuare errori e anomalie;
- **Test di sistema:** Il simulatore PetaLinux su QEMU consente di eseguire test di sistema per valutare il comportamento del software e identificare eventuali problemi o errori di interazione tra i diversi componenti del sistema operativo e dell'applicazione.

Avvio di QEMU

Per avviare QEMU sull'host è necessario eseguire all'interno della cartella del progetto di cui si è creata l'immagine boot, il seguente comando:

```
petalinux-boot --qemu --kernel
```

Se tutti i passaggi eseguiti precedentemente sono andati a buon fine, quindi la generazione dell'immagine di boot è stata effettuata con successo, dopo aver lanciato il comando verranno mostrate nel Terminale Linux le seguenti informazioni, mostrate in Figura.

```
Model: ZynqMP ZCU102 Rev1.0
Board: Xilinx ZynqMP
DRAM: 4 GiB
PMUFW: v1.1
EL Level: EL2
Chip ID: unknown
NAND: 0 MiB
MMC: mmc@ff170000: 0
In: serial
Out: serial
Err: serial
Bootmode: JTAG_MODE
Reset reason:
Net:
ZYNQ GEM: ff0e0000, mdio bus ff0e0000, phyaddr 12, interface rgmii-id
```

QEMU - Info sulla board mostrate al boot

Nella Figura precedente, si può notare che la board per cui è stata buildata l'immagine di boot è la Zynq UltraScale+ MPSoC, in particolare, il modello è la ZCU102. Successivamente viene avviato il kernel di Linux con i messaggi restituiti dal processo di boot.

```
Starting kernel ...
[    0.000000] Booting Linux on physical CPU 0x410fd034]
[    0.000000] Linux version 5.10.0-xilinx-v2021.1 (oe-user@oe-host) (arch64-xilinx-linux-gcc (GCC) 10.2.0, GNU ld (GNU Binutils) 2.
35.1) #1 SMP Fri Jun 4 15:57:16 UTC 2021
[    0.000000] Machine model: ZynqMP ZCU102 Rev1.0
[    0.000000] Memory policy: ECC disabled, Data caging disabled, Normal memory
[    0.000000] printk: bootconsole [cdns5] enabled
[    0.000000] efi: UEFI not found.
[    0.000000] cma: Reserved 256 MiB at 0x00000000d0000000
[    0.000000] Zone ranges:
[    0.000000]   DMA32  [mem 0x0000000000000000-0x00000000ffffffffff]
[    0.000000]   Normal  [mem 0x0000000100000000-0x000000087fffffffff]
[    0.000000]   Movable zone start for each node
[    0.000000] Early memory node ranges:
[    0.000000]   Node 0: [mem 0x0000000000000000-0x000000007fffffff]
[    0.000000]   node 0: [mem 0x0000000800000000-0x000000087fffffff]
[    0.000000] Zeroed struct page in unavailable ranges: 256 pages
[    0.000000] Initrd setup node 0 [mem 0x0000000000000000-0x000000087fffffff]
```

QEMU - Avvio del kernel

Al termine del boot, viene mostrata la shell di Linux con cui è possibile interagire tramite i classici comandi, e quindi, a questo punto è **possibile effettuare le opportune sperimentazioni tramite il simulatore senza aver bisogno dell'HW fisico**.

```
Starting tcf-agent: [ 18.922852] random: crng init done
[ 18.924204] random: 7 urandom warning(s) missed due to ratelimiting
OK
```

```
root@xilinx-zcu102-2021_1:~# |
```

QEMU - Shell Linux

È possibile altresì collegarsi a QEMU anche tramite *ssh* specificando nel comando precedente di boot la seguente opzione:

```
--qemu-args "-net nic -net nic -net nic -net nic,netdev=eth0 -netdev user,id=eth0,hostfwd=tcp::1114-:22"
```

Se la rete è configurata correttamente, con il comando *ssh root@127.0.0.1 -p 1114*, ci verrà richiesta la password del sistema Linux, e successivamente sarà possibile interagire con la shell di PetaLinux allo stesso modo visto prima, sfruttando in questo caso anche la possibilità di trasferire dei file a QEMU tramite il comando *scp*.

```
PS C:\Users\giuse> ssh root@127.0.0.1 -p 1114
root@127.0.0.1's password:
root@xilinx-zcu102-2021_1:~# |
```

QEMU - Collegamento tramite SSH

Utilizzo di Vitis AI sull'host

Per utilizzare Vitis AI, in questa sperimentazione, si fa uso di un **Docker Container** messo a disposizione da Xilinx/Vitis-AI. Prima di passare all'installazione del Docker, si fa un git clone del repository di Vitis AI.

```
git clone --recurse-submodules https://github.com/Xilinx/Vitis-AI
cd Vitis-AI
```

A questo punto sarà possibile eseguire l'installazione del container Docker.

Si noti: si necessita dell'installazione del Docker Engine e dell'avvio dello stesso per eseguire correttamente i successivi comandi. E' possibile installare Docker Desktop dal [sito ufficiale](#) (<https://www.docker.com/products/docker-desktop/>) del prodotto.

Configurazione Docker

Si eseguono i seguenti comandi sulla shell WSL2 nella cartella Vitis-AI (**vers. 2.0**) precedentemente clonata da GitHub:

```
docker pull xilinx/vitis-ai:2.0.0
./docker_run.sh xilinx/vitis-ai
```

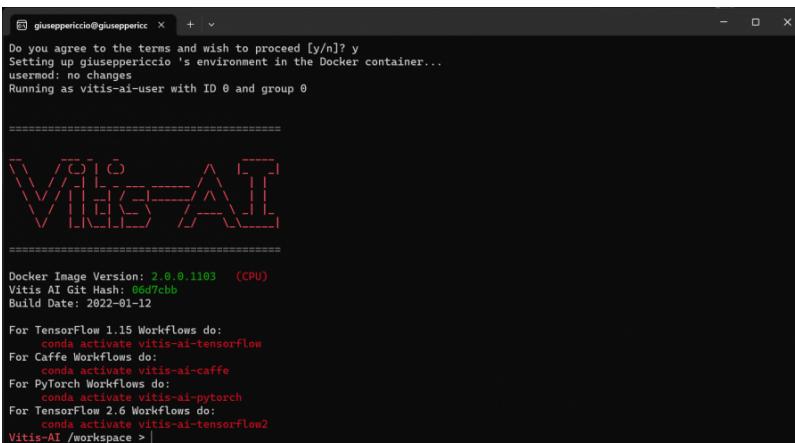
Per verificare la corretta installazione dell'immagine di Vitis-AI sul Docker, si esegue il comando:

```
docker images
```

```
giuseppericcio@giuseppericcio:/mnt/c/Users/giuse/Desktop/Vitis-AI-2.0$ docker images
REPOSITORY          TAG      IMAGE ID   CREATED    SIZE
xilinx/vitis-ai-cpu 2.0      187ef9604662  18 months ago  18.1GB
xilinx/vitis-ai     2.0      187ef9604662  18 months ago  18.1GB
```

Immagini Vitis AI scaricate su Docker

./docker_run.sh è un file bash presente nella cartella Vitis-AI, che permette di avviare il Docker con l'immagine di Vitis-AI precedentemente scaricata.



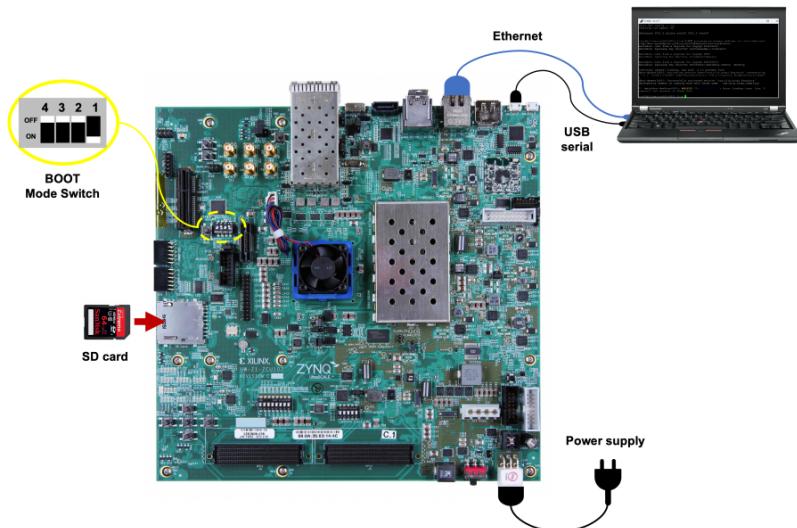
Vitis AI - Avvio sul Docker

L'ambiente è pronto per eseguire il flow di Vitis-AI.

Configurazione del target (ZCU102)

Interconnessione dell'host e del target

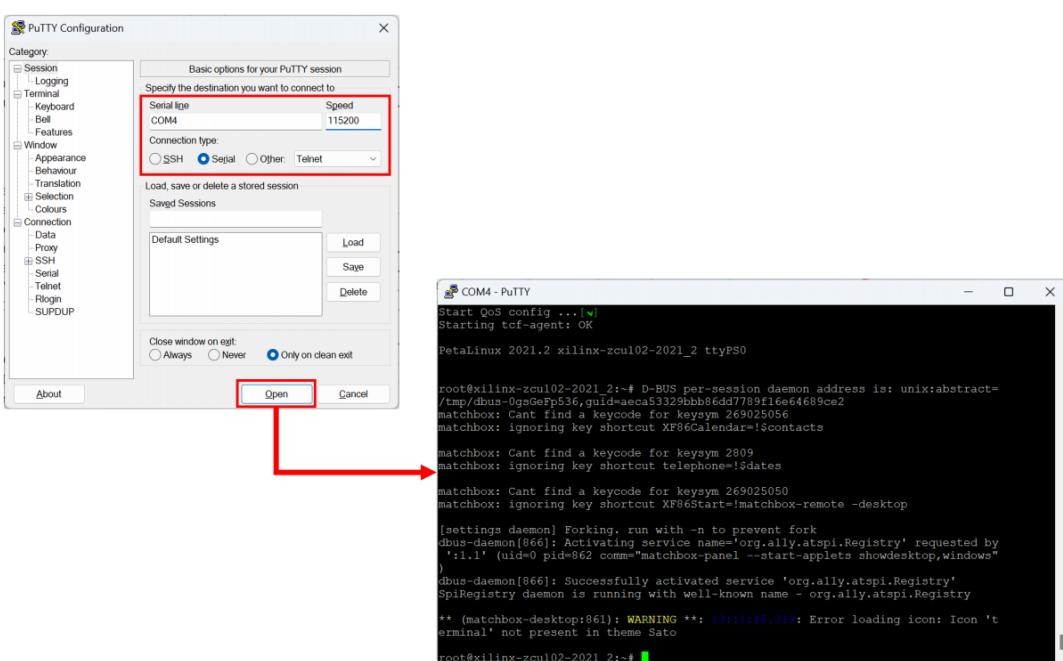
L'interconnessione tra Host e la ZCU102 può avvenire sia via USB UART e sia via Ethernet.



Interconnessione tra ZCU102 e Host

UART

Nel caso della sperimentazione si è usata la USB UART durante il processo di Flash dell'immagine su SD Card. Questa opzione consente di utilizzare un terminale seriale (es. PuTTY) per comunicare con la scheda ZCU e accedere alla console di sistema. Per collegarsi alla board ZCU102 tramite UART, si fa uso del programma PuTTY, in particolare per configurare opportunamente il collegamento serve impostare la linea seriale a COM4 ed il baud rate a 115200, come mostrato di seguito.



Collegamento tramite UART - PuTTY

Se la board è collegata correttamente all'host, all'interno di PuTTY verrà aperto il terminale che permette di interagire con il target dando le istruzioni da linea di comando. Tuttavia, per il progetto in esame si è deciso di usare il collegamento tramite Ethernet in quanto quello tramite UART **non permette lo scambio di file** rendendo le fasi successive più lunghe.

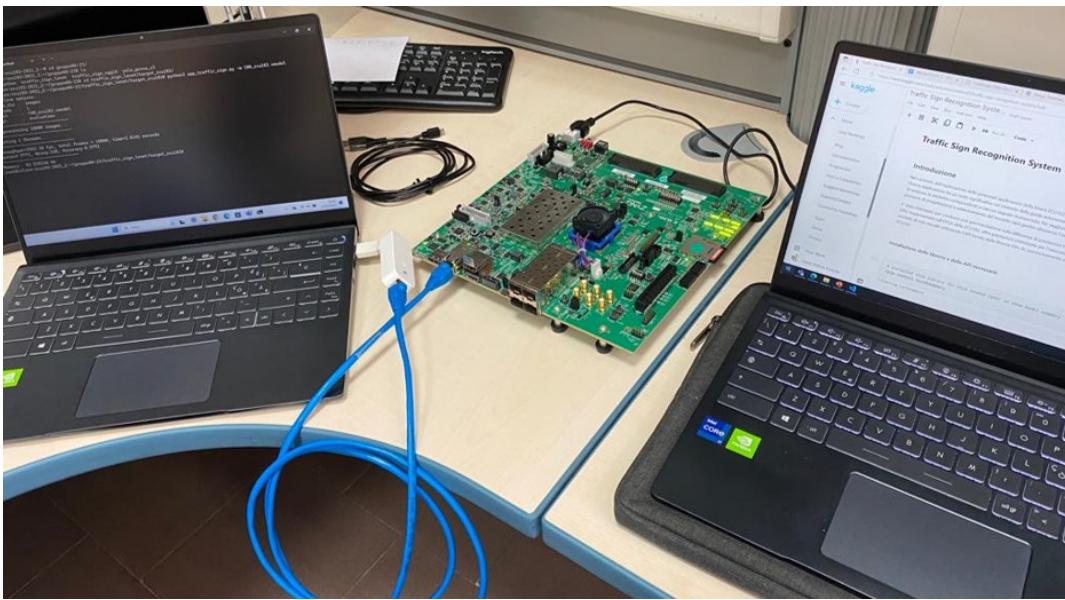
Ethernet

La connessione Ethernet tra la scheda ZCU e l'host consente di accedere alla scheda tramite un indirizzo IP assegnato sulla rete locale, utilizzando protocolli come SSH o Telnet. Questa connessione consente di trasferire dati tra l'host e la scheda attraverso il comando `scp`, inclusi l'eseguibile dell'applicazione, il modello contenente la rete neurale, il runtime di Vitis-AI, le immagini da elaborare, i file contenenti le etichette di classificazione e altri dati necessari. La connessione tramite SSH, avviene eseguendo il comando `ssh root@192.168.1.1` dalla PowerShell della macchina host (si noti che l'indirizzo IP del target potrebbe essere diverso in base alle impostazioni specifiche dell'ambiente). Il risultato in caso di successo sarà il seguente.

```
PS C:\Users\giuse> ssh root@192.168.1.1
root@192.168.1.1's password:
root@xilinx-zcu102-2021_2:~# |
```

Collegamento tramite Ethernet - SSH

Interconnessione Host - ZCU102 in laboratorio per la sperimentazione:



Interconnessione Host - ZCU102 svolto in laboratorio per la sperimentazione

Utilizzo di PetaLinux sul target (ZCU102)

Flash dell'immagine su SD card

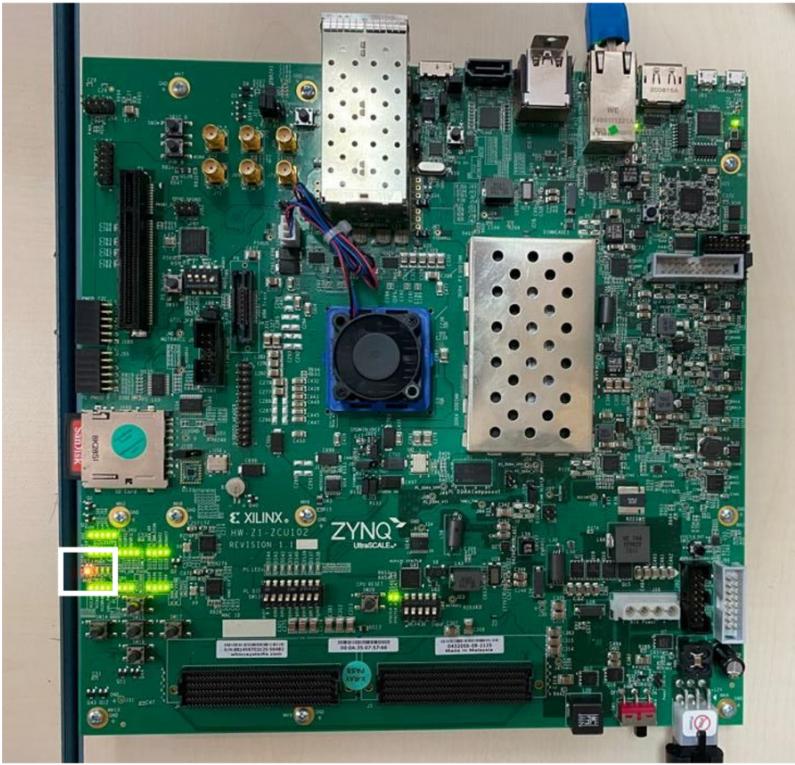
A valle del processo di installazione di Petalinux precedente sarà possibile **montare** la stessa sulla board mediante l'utilizzo di una **SD card**.

Pertanto si elencano i passi necessari per il processo della configurazione target:

- Si copiano i seguenti file da:
 - <plnx-proj-root>/pre-built/linux/images/ nella directory principale della prima partizione della scheda SD. (**Si noti:** La scheda SD deve essere partizionata in maniera opportuna: 4 MB non allocati all'inizio, poi una partizione FAT32 da 1 GB etichettata come BOOT, in cui saranno caricati *BOOT.BIN*, *boot.scr* ed *image.ub*, ed il resto dello spazio formattato in ext4 etichettato come *rootfs*, in cui estrarre il file *rootfs.tar.gz*)
 - I file da caricare sulla SD card sono:
 - *BOOT.BIN*
 - *image.ub*
 - *boot.scr*
 - Estrarre la cartella *rootfs.tar.gz* nella partizione ext4 della scheda SD.
 - Si setta la board in boot mode:
 - **Rev 1.0: SW6[4:1] - OFF,OFF,OFF,ON** (Figura Interconnessione tra ZCU102 e Host)
 - Inserire la SD card e accendere la board.
 - Collegare la porta seriale della scheda (USB) all'Host e aprire una console per la comunicazione seriale come PuTTY nel caso in esame (impostare baud rate a 115200).
 - Verrà mostrato un messaggio di boot sulla console.

Caricamento della DPU sulla parte PL del target (ZCU102)

All'avvio la board Zynq UltraScale+ MPSoC (versione ZCU102) non presenta nessun IP core caricato nella sezione FPGA (PL - Programmable Logic), tale situazione è facilmente denotabile osservando la board stessa. Infatti, ci sono due file di led verdi con al centro un ulteriore led che quando è rosso indica che nella parte PL non ci sono IP core caricati.

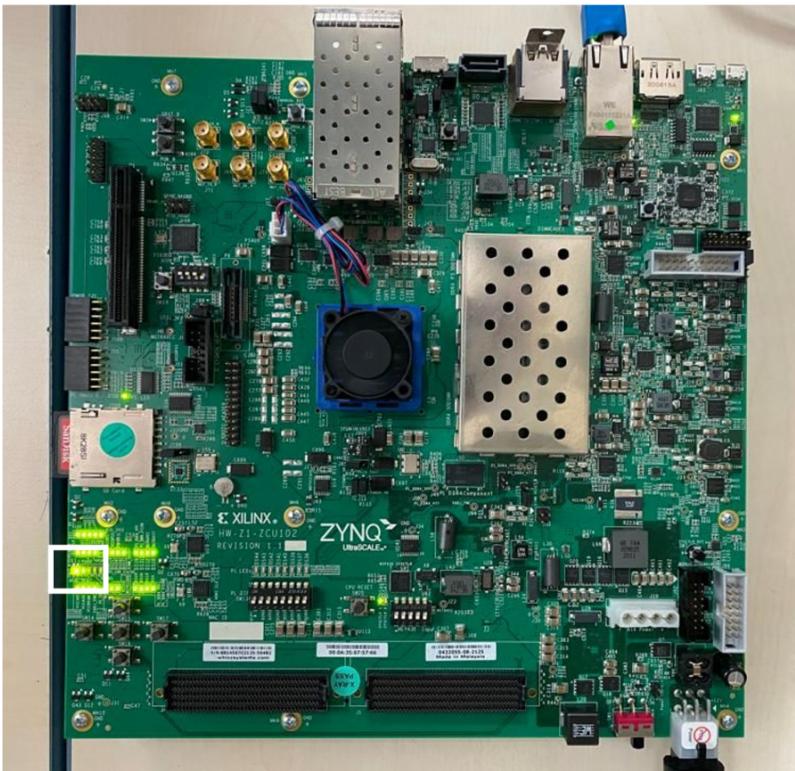


ZCU102 senza DPU caricata sulla PL

Essendo che il nostro progetto prevede l'uso della DPU, in 2 diverse configurazioni, una con 1 core ed un'altra con 3 cores dobbiamo caricare i file generati a partire dall'xsa che risiedono nella cartella `lib/firmware/xilinx/`. In particolare, nel caso in esame all'interno della precedente cartella avremo altre 2 cartelle, rispettivamente `DPU_cores1` e `DPU_cores3`, al loro interno queste cartelle presentano 2 file, il primo con formato `.dtbo` rappresenta il file di descrizione dell'HW da aggiungere al Device Tree di base in maniera dinamica durante l'avvio del sistema. In pratica, i file `.dtbo` contengono una rappresentazione binaria dei dati di configurazione specifici dei dispositivi aggiuntivi o personalizzati. Il secondo file ha estensione `.bin` e contiene i file binari che implementano la DPU su FPGA. Il comando che permette di caricare la DPU sulla ZCU102, prevede l'uso del tool `fpgautl` fornito dall'ambiente di sviluppo PetaLinux. Il comando completo è il seguente:

```
fpgautl -o DPU_cores1.dtbo -b DPU_cores1.bit.bin
```

Se la DPU viene caricata correttamente sulla parte PL, il led che in precedenza era **rosso** dovrebbe diventare di colore **verde**, come mostrato in Figura.



ZCU102 con DPU caricata sulla PL

Utilizzo di Vitis AI sul target (ZCU102)

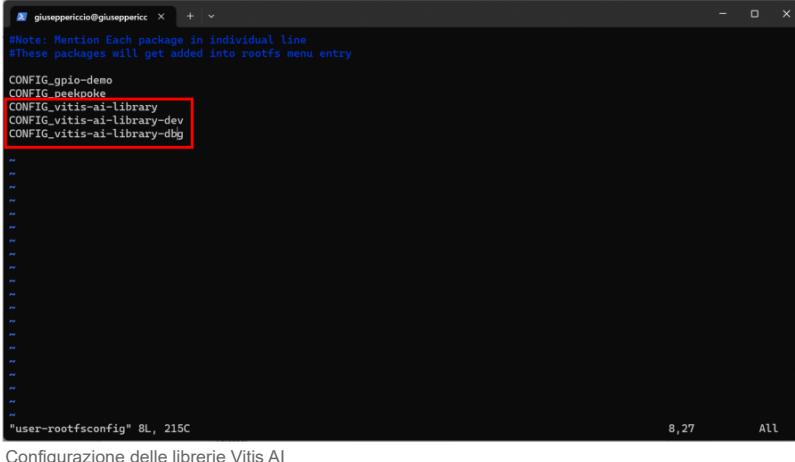
Al fine di eseguire il modello quantizzato e compilato con Vitis AI occorre installare tale ambiente, e tutte le librerie ad esso associate, anche sulla piattaforma target (Zynq UltraScale+ MPSoC ZCU102). Per fare ciò ci sono 2 modalità, la prima prevede il rebuild dell'ambiente PetaLinux dopo l'aggiunta delle librerie, l'altra strada prevede l'installazione delle librerie online direttamente sulla board. Per il seguente elaborato si segue la prima strada, con i passaggi spiegati di seguito.

Installazione dell'ambiente e delle librerie di Vitis AI sulla board ZCU102

In manzitutto, se si vuole installare Vitis AI 2.0 nel root file system quando viene generata l'immagine da PetaLinux, occorre importare le recipes Vitis AI 2.0. La cartella delle recipes di Vitis AI 2.0 è localizzata in *Vitis-AI/tools/Vitis-AI-Recipes/recipes-vitis-ai* (Questa cartella si trova nel [repository ufficiale di GitHub](https://github.com/Xilinx/Vitis-AI/tree/2.0) (<https://github.com/Xilinx/Vitis-AI/tree/2.0>) di Vitis-AI). Successivamente, occorre copiare questa cartella nella cartella del progetto di PetaLinux, il comando per fare ciò è il seguente:

```
cp -r Vitis-AI/tools/Vitis-AI-Recipes/recipes-vitis-ai <petalinux project>/project-spec/metauser/
```

Dopo aver copiato la cartella delle recipes di Vitis-AI, vanno aggiunte queste librerie da configurare nel root file system, aggiungendo all'interno del file localizzato nella cartella *<petalinux project>/project-spec/meta-user/conf/user-rootfsconfig*, le seguenti linee:

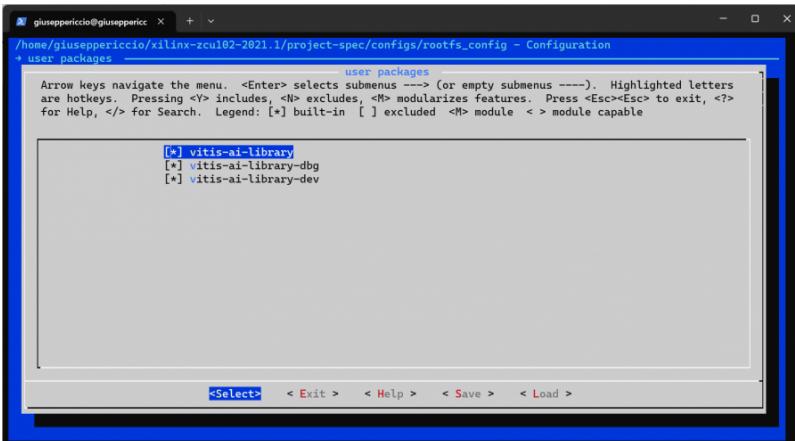


Configurazione delle librerie Vitis AI

A questo punto è possibile abilitare l'importazione vera e propria delle librerie di Vitis-AI all'interno dell'immagine generata da PetaLinux, eseguendo il seguente comando:

```
petalinux-config -c rootfs
```

Dopo aver dato il comando precedente, si apre il menu mostrato in Figura, in cui vanno abilitate nella sezione *User Packages*, tutte le librerie di Vitis-AI.



Aggiunta librerie Vitis AI al rootfs

Dopo aver salvato la configurazione del rootfs appena modificato, si procede al rebuild dell'immagine tramite il comando *petalinux-build*, a cui segue il flash dell'immagine sulla scheda SD allo stesso modo visto in precedenza. Infine, si importano sulla ZCU102, le librerie di Vitis-AI Runtime, scaricando l'archivio dei pacchetti dal link ufficiale fornito da Xilinx ([link per il download](https://www.xilinx.com/bin/public/openDownload?filename=vitis-ai-runtime-2.0.0.tar.gz) (<https://www.xilinx.com/bin/public/openDownload?filename=vitis-ai-runtime-2.0.0.tar.gz>)). Dopo aver scaricato l'archivio, si estra il contenuto, e si importa la cartella *vitis-ai-runtime-2.0.0/aarch64/centos* sulla ZCU102 tramite il seguente comando:

```
scp -r vitis-ai-runtime-2.0.x/aarch64/centos root@192.168.1.1:~/
```

Per installare i pacchetti di Vitis-AI runtime sulla board, si eseguono le seguenti operazioni tramite la shell (collegandosi tramite ssh):

```
cd ~/centos  
bash setup.sh
```

Dopo che l'installazione delle librerie Vitis-AI runtime è stata completata, queste librerie saranno installate nella cartella */usr/lib*. Un estratto del contenuto di questa cartella è mostrato di seguito.

```

libvitis_ai_library-3Dsegmentation.so
libvitis_ai_library-3Dsegmentation.so.2
libvitis_ai_library-3Dsegmentation.so.2.0.0
libvitis_ai_library-RGBDsegmentation.so
libvitis_ai_library-RGBDsegmentation.so.2
libvitis_ai_library-RGBDsegmentation.so.2.0.0
libvitis_ai_library-bcc.so
libvitis_ai_library-bcc.so.2
libvitis_ai_library-bcc.so.2.0.0
libvitis_ai_library-benchmark.so
libvitis_ai_library-benchmark.so.2
libvitis_ai_library-benchmark.so.2.0.0
libvitis_ai_library-c2d2_lite.so
libvitis_ai_library-c2d2_lite.so.2
libvitis_ai_library-c2d2_lite.so.2.0.0
libvitis_ai_library-carplaterecog.so
libvitis_ai_library-carplaterecog.so.2
libvitis_ai_library-carplaterecog.so.2.0.0
libvitis_ai_library-centerpoint.so
libvitis_ai_library-centerpoint.so.2
libvitis_ai_library-centerpoint.so.2.0.0
libvitis_ai_library-classification.so
libvitis_ai_library-classification.so.2
libvitis_ai_library-classification.so.2.0.0
libvitis_ai_library-clocs.so
libvitis_ai_library-clocs.so.2
libvitis_ai_library-clocs.so.2.0.0
libvitis_ai_library-covid19segmentation.so

```

Librerie Vitis AI sulla board

La configurazione sulla board è pronta.

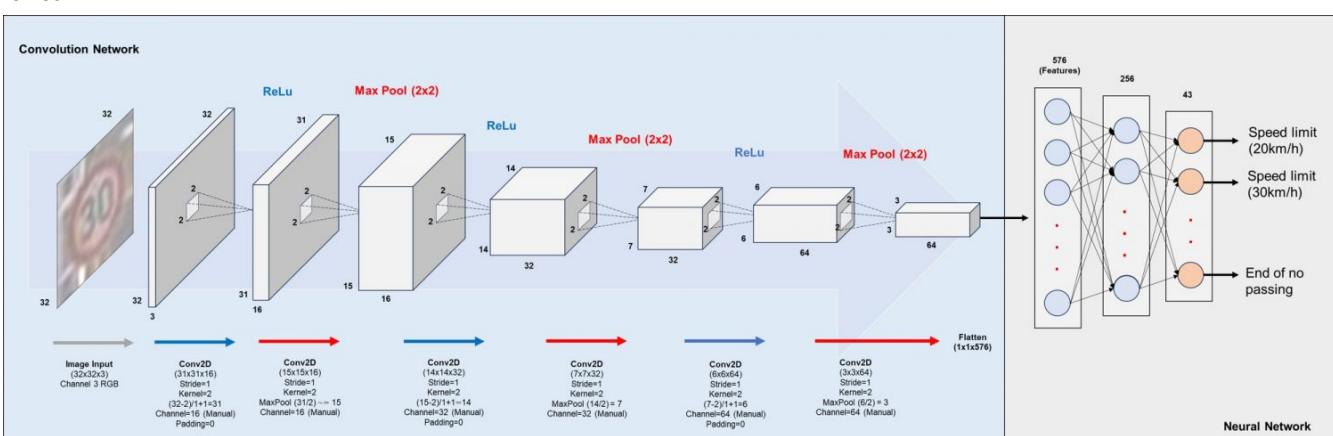
Applicazione: Signal Traffic Classification

Si descrive l'applicazione che verrà sperimentata sulla board ZCU102.

Descrizione dell'architettura della CNN utilizzata per la Signal Traffic Classification

Per sperimentare le prestazioni dell'applicazione sulla board, si costruiscono due reti convoluzionali per il problema della recognition dei segnali stradali, applicazione utile ad esempio nell'ambito di *autonomous driving technology*. Per risolvere tale problema si propongono il modello LeNet e il modello VGG16.

LeNet



Architettura del modello LeNet

Il primo blocco prende in input immagini RGB di dimensioni 32x32 e produce 16 feature map, che vengono successivamente ridotte tramite pooling. Il secondo blocco aumenta la complessità del modello aumentando il numero di feature map a 32. Infine, il terzo blocco produce 64 feature map. Dopo l'ultimo strato di pooling, i dati vengono appiattiti e passati attraverso due strati lineari, con la funzione di attivazione ReLU tra di essi. L'ultimo strato lineare produce l'output finale, che corrisponde alle etichette dei 43 segnali stradali possibili. Il modello è stato implementato utilizzando il framework PyTorch e addestrato su una GPU NVIDIA TESLA P100 (con 16 GB di memoria dedicata) messo a disposizione dalla piattaforma cloud di Kaggle per accelerare il processo di addestramento. Di seguito un estratto di codice del modello:

```

model = nn.Sequential(
    # 1st convolutional network Layers
    nn.Conv2d(3, 16, kernel_size=(2, 2), stride=(1, 1), padding=get_same_padding((2, 2))), # Convolution
    nn.BatchNorm2d(16), # Normalization
    nn.ReLU(inplace=True), # Activation
    nn.MaxPool2d(kernel_size=(2, 2)), # Pooling

    # 2nd convolutional network Layers
    nn.Conv2d(16, 32, kernel_size=(2, 2), stride=(1, 1), padding=get_same_padding((2, 2))), # Convolution
    nn.BatchNorm2d(32), # Normalization
    nn.ReLU(inplace=True), # Activation
    nn.MaxPool2d(kernel_size=(2, 2)), # Pooling

    # 3rd convolutional network Layers
    nn.Conv2d(32, 64, kernel_size=(2, 2), stride=(1, 1), padding=get_same_padding((2, 2))), # Convolution
    nn.BatchNorm2d(64), # Normalization
    nn.ReLU(inplace=True), # Activation
    nn.MaxPool2d(kernel_size=(2, 2)), # Pooling

    # Flatten Data
    nn.Flatten(), # Flatten
)

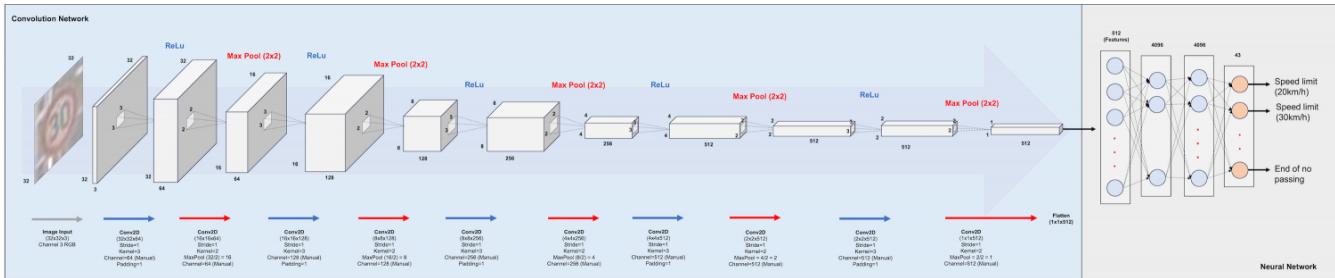
```

```

    # feed forward Layers
    nn.Linear(576, 256), # Linear
    nn.ReLU(inplace=True), # Activation
    nn.Linear(256, 43) # Linear
)

```

VGG16



Architettura del modello VGG16

È costituito da cinque blocchi di convoluzioni, ciascuno seguito da attivazione ReLU e max pooling. I primi quattro blocchi aumentano progressivamente il numero di feature map, partendo da 64 e arrivando a 512. Ogni blocco sfrutta convoluzioni 3x3 con padding per mantenere le dimensioni degli input. Dopo l'ultimo strato di pooling, i dati vengono appiattiti e passati attraverso tre strati lineari con attivazione ReLU. Gli strati lineari riducono gradualmente le dimensioni dell'output, partendo da 512 e terminando con 43, corrispondenti alle etichette dei segnali stradali da riconoscere. Anche in questo caso, il modello è stato implementato utilizzando il framework *PyTorch* e addestrato sulla *GPU NVIDIA TESLA P100* (con 16 GB di memoria dedicata) messo a disposizione dalla piattaforma cloud di *Kaggle* per accelerare il processo di addestramento. Di seguito un estratto di codice del modello:

```

model = nn.Sequential(
    # 1st convolutional network Layers
    nn.Conv2d(3, 64, kernel_size=(3,3), stride=(1,1), padding=(1,1)), # Convolution
    nn.ReLU(inplace=True), # Activation
    nn.MaxPool2d(kernel_size=(2,2)), # Pooling

    # 2nd convolutional network Layers
    nn.Conv2d(64, 128, kernel_size=(3,3), stride=(1,1), padding=(1,1)), # Convolution
    nn.ReLU(inplace=True), # Activation
    nn.MaxPool2d(kernel_size=(2,2)), # Pooling

    # 3rd convolutional network Layers
    nn.Conv2d(128, 256, kernel_size=(3,3), stride=(1,1), padding=(1,1)), # Convolution
    nn.ReLU(inplace=True), # Activation
    nn.MaxPool2d(kernel_size=(2,2)), # Pooling

    # 4th convolutional network Layers
    nn.Conv2d(256, 512, kernel_size=(3,3), stride=(1,1), padding=(1,1)), # Convolution
    nn.ReLU(inplace=True), # Activation
    nn.MaxPool2d(kernel_size=(2,2)), # Pooling

    # 5th convolutional network Layers
    nn.Conv2d(512, 512, kernel_size=(3,3), stride=(1,1), padding=(1,1)), # Convolution
    nn.ReLU(inplace=True), # Activation
    nn.MaxPool2d(kernel_size=(2,2)), # Pooling

    # Flatten Data
    nn.Flatten(), # Flatten

    # feed forward Layers
    nn.Linear(512, 4096), # Linear
    nn.ReLU(inplace=True), # Activation

    nn.Linear(4096, 4096), # Linear
    nn.ReLU(inplace=True), # Activation

    nn.Linear(4096, 43) # Linear
)

```

Entrambi i modelli verranno salvati in **formato.pth** per poi essere opportunamente quantizzati e compilati per l'esecuzione sulla board ZCU102. È importante salvare il modello in questo formato in quanto sarà il punto di partenza per l'utilizzo di Vitis AI.

Entrambi modelli sono eseguiti in primo luogo sulla GPU fornita da Kaggle: NVIDIA TESLA P100 GPUs. Si è scelta questa GPU in quanto presenta un limite di dimensioni disponibile per l'esecuzione di 15.5 GB circa, sufficienti per l'addestramento del modello completo.

Implementazione dell'applicazione di Signal Traffic Classification su cloud Kaggle

Si lascia il [repository GitHub](https://github.com/LaErreq/Zynq_Ultrascaling_Vitis_AI/blob/main/traffic_sign_recognition_lenet_vgg16.ipynb) (https://github.com/LaErreq/Zynq_Ultrascaling_Vitis_AI/blob/main/traffic_sign_recognition_lenet_vgg16.ipynb) relativa all'implementazione completa del modello su cloud *Kaggle*.

Addestramento della CNN utilizzando PyTorch

```

def train_model(model=model,
               optimizer=torch.optim.Adam,
               epochs=5,
               batch_size=100,
               steps_per_epochs=100,
               l2_reg=0,
               max_lr=0.01,
               grad_clip=0.5):

    # Preparazione del dataset di training per il Data Loader
    train_ds = [(x,y) for x,y in zip(xtrain,ytrain)]

    # Data Loader usato per il train model
    training_dl = torch.utils.data.DataLoader(train_ds,batch_size=batch_size)

    # Data Loader per epoch e evaluation sul train data
    train_dl = torch.utils.data.DataLoader(train_ds,batch_size=batch_size * steps_per_epochs)

```

```

# Initialized the Optimizer to update weights and bias of model parameters
optimizer = optimizer(model.parameters(), weight_decay=l2_reg) LeNet
optimizer = optimizer(model.parameters(), weight_decay=l2_reg, lr=max_lr) VGG16

# Initialized the Scheduler to update learning rate as per one cycle policy
sched = torch.optim.lr_scheduler.OneCycleLR(optimizer, max_lr, epochs=epochs, steps_per_epoch=int(steps_per_epochs * 1.01))

# Training Started
for i in range(epochs):

    # Carica i batch del training data loader
    # Load Batches of training data loader
    for j, (xb, yb) in enumerate(training_dl):

        # ogni training batch spostato nella cuda memory per velocizzare il processing
        xb = xb.to(torch.device('cuda'), non_blocking=True)
        yb = yb.to(torch.device('cuda'), non_blocking=True)

        # calcolo della Loss e gradienti
        yhat = model(xb.float())
        loss = nn.functional.cross_entropy(yhat, yb)
        loss.backward()

        # clip degli outlier like gradients
        nn.utils.clip_grad_value_(model.parameters(), grad_clip)

        # aggiornamento dei weights and bias
        optimizer.step()
        optimizer.zero_grad()

        # aggiornamento della Learning Rate
        sched.step()

        # break the loop quando arriva al numero fissato
        if steps_per_epochs == j:
            break

    # Epochs end evaluation

    device = torch.device('cuda') # initialized cuda to device

    # move train data to cuda
    xtrain_cuda = xtrainb.to(device, non_blocking=True)
    ytrain_cuda = ytrainb.to(device, non_blocking=True)

    # calcolo del train loss e accuracy
    yhat = model(xtrain_cuda.float())
    ypred = yhat.argmax(axis=1)
    train_loss = float(nn.functional.cross_entropy(yhat, ytrain_cuda))
    train_acc = float((ypred == ytrain_cuda).sum() / len(ytrain_cuda))

    # move test data to cuda
    xtest_cuda = xtest.to(device, non_blocking=True)
    ytest_cuda = ytest.to(device, non_blocking=True)

    # Calculate test loss and accuracy
    yhat = model(xtest_cuda.float())
    ypred = yhat.argmax(axis=1)
    val_loss = float(nn.functional.cross_entropy(yhat, ytest_cuda))
    val_acc = float((ypred == ytest_cuda).sum() / len(ytest_cuda))

# Initialized all the evaluation history of all epochs to a dict
history = {'Train Loss':hist[0], 'Val Loss':hist[1], 'Train Accuracy':hist[2], 'Val Accuracy':hist[3]}

# return the history as pandas dataframe
return pd.DataFrame(history)

```

Questa funzione consente di addestrare il modello utilizzando un set di dati di addestramento suddiviso in batch. Durante il processo di addestramento, vengono calcolate la **loss function** e i gradienti per aggiornare i pesi e i bias del modello utilizzando un optimizer specificato, come ad esempio, nel caso in esame **Adam** (per LeNet) e **SGD** (Statistic Gradient Descent)(per VGG16) sulla base della rete. Durante ogni epoca di addestramento, vengono valutate la loss function e l'accuratezza del modello sia sul set di dati di addestramento che sul set di dati di validazione. Queste metriche vengono quindi registrate in un elenco *history* per monitorare le prestazioni del modello nel corso delle epoche.

Il modello e i dati vengono spostati sulla GPU nel caso in Cloud per accelerare il processo di addestramento. Vengono utilizzate anche tecniche di gestione della memoria, come la liberazione dei tensori utilizzati e lo svuotamento della cache della GPU per i limiti imposti su *Kaggle*.

Al termine dell'addestramento, viene restituito un DataFrame contenente lo storico delle metriche di addestramento e validazione per la valutazione delle prestazioni che verranno confrontate successivamente con quelle della board.

Valutazione delle prestazioni del modello sul dataset di test su Kaggle

Per addestrare i due modelli sono stati usati i seguenti iperparametri, scelti in **maniera sperimentale**, con il quale si sono ottenute le prestazioni migliori in termini di **accuracy**.

- **LeNet**

```
history = train_model(model, optimizer=torch.optim.Adam, epochs=25, steps_per_epochs=100, l2_reg=0, max_lr=0.015, grad_clip=0.5)
```

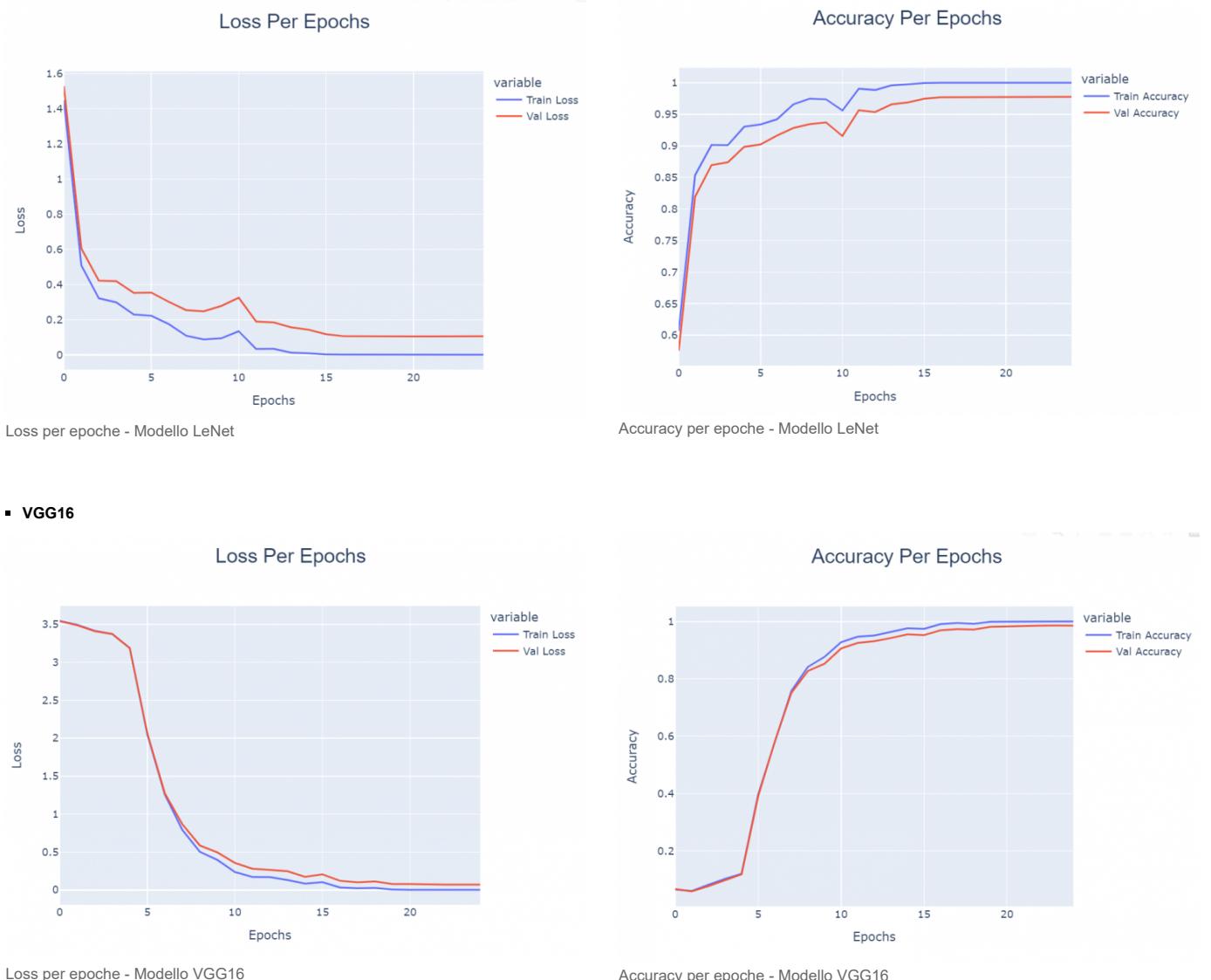
- **VGG16**

```
history = train_model(model, optimizer=torch.optim.SGD, epochs=25, steps_per_epochs=100, l2_reg=0, max_lr=0.1, grad_clip=0.5)
```

Si addestra il modello utilizzando l'optimizer *Adam* per LeNet e *SGD* per VGG16 per 25 epoche. Durante l'addestramento, vengono eseguiti 100 steps per epoca. Viene calcolata la loss function e vengono aggiornati i pesi del modello utilizzando i gradienti calcolati. Vengono, inoltre, impostati il learning rate massimo a 0.015 per LeNet, mentre il learning rate massimo a 0.1 per VGG16 e il gradiente massimo a 0.5 per limitare i valori dei gradienti.

I risultati:

- **LeNet**



Deploying della CNN sulla ZCU102

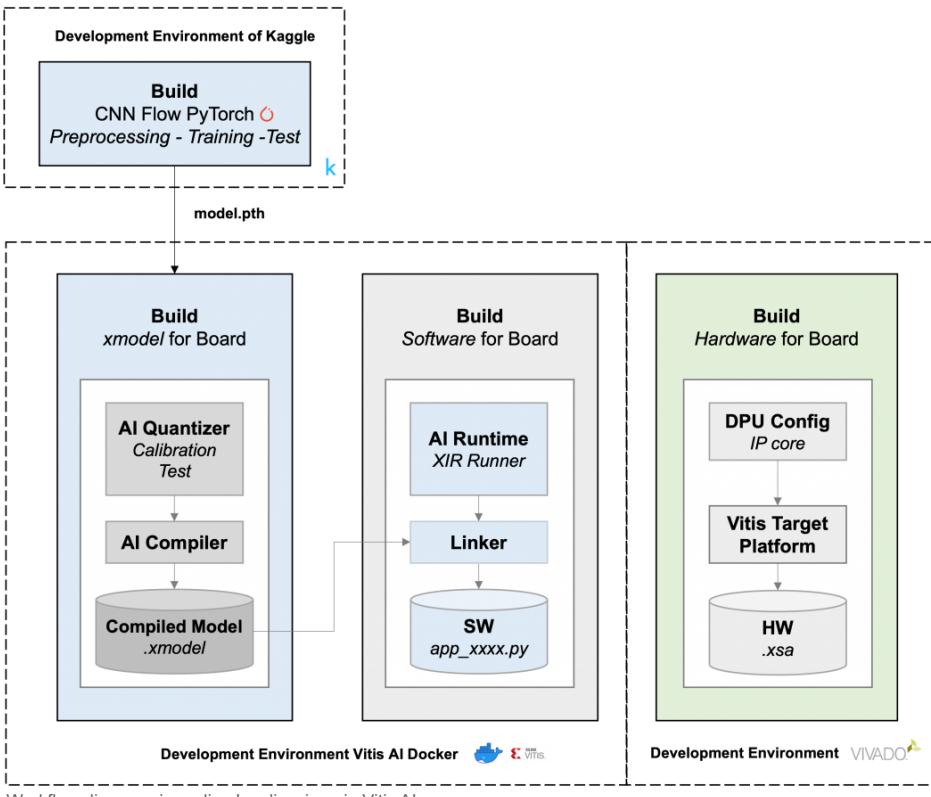
Come accennato in precedenza, il processo di **AI Quantizer** converte il modello in virgola mobile in un modello in virgola fissa a 8 bit. Successivamente, l'**AI Compiler** compila il modello quantizzato per generare un insieme di istruzioni eseguibili sulla DPU. La compilazione riguarda solo il modello CNN, mentre l'applicazione nel caso specifico in questione, essendo stata implementata utilizzando Python, non richiede la compilazione del codice sorgente. Questo perché Python è un linguaggio interpretato che consente di eseguire direttamente il codice senza dover passare attraverso la fase di compilazione, come richiesto ad esempio da C++. In dettaglio l'esecuzione di un'applicazione Python passa prima da una fase di pre-compilazione in bytecode eseguita a runtime, che viene quasi sempre riutilizzata dopo la prima esecuzione del programma, evitando così di reinterpretare ogni volta il sorgente e migliorando le prestazioni.

Descrizione del flusso di esecuzione di Vitis AI con PyTorch per la ZCU102

Sarà dunque possibile generare un modello eseguibile sulla DPU partendo dall'architettura di una rete neurale definita con PyTorch (modello .pth), tramite il framework Vitis AI. Per ogni principale framework di Machine Learning supportato da Vitis AI, è disponibile un flusso dedicato per la generazione di un file .xmodel. Il flusso di esecuzione seguito sono le seguenti fasi:

- Addestramento di una rete neurale floating point (come visto in precedenza);
- Quantizzazione (**AI Quantizer**) del modello floating-point per generare un modello fixed-point ad 8 bit;
- Compilazione (**AI Compiler**) del modello quantizzato per generare un *xmodel* eseguibile dalla DPU della ZCU102;
- Trasferimento del modello compilato e dell'applicazione sul dispositivo target ZCU102.

Il flusso è eseguito totalmente all'interno del contenitore Docker di Vitis AI. Si mostra uno schema dell'ipotetico flusso di esecuzione:



Workflow di esecuzione di un'applicazione in Vitis AI

Quantizer

La quantizzazione consiste, come già detto, nell'approssimare i parametri del modello da rappresentazioni floating-point a rappresentazioni fixed-point a precisione ridotta, come ad esempio a 8 bit. Questo processo consente di **ridurre significativamente la dimensione del modello** e accelerare l'esecuzione su dispositivi a risorse limitate proprio come la DPU. La quantizzazione viene eseguita in due fasi: calibrazione e test. Si riportano i dettagli delle fasi di seguito:

- Durante la fase di **calibrazione**, il modello viene esposto a un sottoinsieme rappresentativo del dataset di input. L'obiettivo è determinare l'intervallo di valori che il modello potrebbe incontrare durante l'esecuzione reale. Questo è importante perché i parametri del modello saranno successivamente arrotondati alla precisione desiderata, e conoscere l'**intervallo di valori aiuta a determinare la scala di quantizzazione ottimale** per ciascun livello del modello.
- Successivamente, nella fase di **test**, il modello quantizzato viene valutato per determinare la sua accuratezza rispetto al modello originale a rappresentazione floating-point. Questo consente di valutare l'effetto della quantizzazione sulla performance del modello, identificando eventuali perdite di precisione e valutando se il modello quantizzato è ancora adatto al compito a cui è destinato.

Per eseguire la quantizzazione, Vitis AI fornisce API specifiche che svolgono queste operazioni, in questo contesto si è riadattato al caso specifico il codice presente al [repository GitHub](#) (<https://github.com/Xilinx/Vitis-AI-Tutorials/tree/1.4>) di Vitis AI. In particolare, i file principali sono quantizer.py e common.py.

quantizer.py

Nel quantizer.py si estrae la parte di codice di interesse, in particolare si fa riferimento all'utilizzo della seguente classe:

```
class torch_quantizer():
    def __init__(self,
                 quant_mode: str, # ['calib', 'test']
                 module: torch.nn.Module,
                 input_args: Union[torch.Tensor, Sequence[Any]] = None,
                 state_dict_file: Optional[str] = None,
                 output_dir: str = "quantize_result",
                 bitwidth: int = 8,
                 device: torch.device = torch.device("cuda"),
                 quant_config_file: Optional[str] = None),
                 target: Optional[str]=None)
```

dove:

- **quant_mode**: permette di settare la modalità calibrazione o la modalità test ('calib' o 'test');
- **module**: modello da quantizzare (torch.nn.Module), ovvero il modulo PyTorch che rappresenta il modello da quantizzare;
- **input_args**: tensore che ha la stessa dimensione dei vettori in ingresso. È necessario affinché il quantizzatore possa inferire la dimensione dei tensori in ingresso alla rete; ragion per cui, il tensore può essere inizializzato con valori casuali;
- **output_dir**: cartella in cui salvare il modello quantizzato;
- **bitwidth**: numero di bit per la quantizzazione, di default è 8 bit (dipende anche dal tipo di DPU su cui si intende eseguire il modello);
- **device**: il dispositivo su cui verrà eseguita la quantizzazione (default è 'cuda');
- **quant_config_file**: il percorso del file di configurazione JSON per la quantizzazione (default è "quantization_config.json")

Pertanto si mostra l'estratto di codice dove viene usata la classe:

```
rand_in = torch.randn([batchsize, 3, 32, 32])
quantizer = torch_quantizer(quant_mode, model, (rand_in), output_dir=quant_model)
quantized_model = quantizer.quant_model

# evaluate
test(quantized_model, device, dset_dir)

# export config
if quant_mode == 'calib':
    quantizer.export_quant_config()
if quant_mode == 'test':
    quantizer.export_xmodel(deploy_check=False, output_dir=quant_model)
```

Se la modalità di quantizzazione è impostata su "calib", esporta le informazioni di quantizzazione chiamando il metodo `export_quant_config()` dell'oggetto **quantizer**.

Se la modalità di quantizzazione è impostata su "test", esporta il modello quantizzato in formato `.xmodel` chiamando il metodo `export_xmodel()` dell'oggetto **quantizer**, specificando la directory di output per il modello quantizzato (`output_dir`).

Su Vitis AI, per eseguire opportunamente la quantizzazione si useranno i seguenti comandi:

■ calibration

```
python -u quantize.py -d ${BUILD_DIR} --quant_mode calib 2>&1 | tee ${LOG_DIR}/quant_calib.log
```

Per **LeNet** si ottiene:

```
[VAIQ_NOTE]: Quantization calibration process start up...
[VAIQ_NOTE]: =>Quant Module is in 'cpu'.
[VAIQ_NOTE]: =>Parsing CNN...
[VAIQ_NOTE]: Start to trace model...
[VAIQ_NOTE]: Finish tracing.
[VAIQ_NOTE]: Processing ops...
| 1/17 [00:00<00:00, 1895.30it/s, OpInfo: name = CNN/Sequential[model]
| 2/17 [00:00<00:00, 1624.44it/s, OpInfo: name = CNN/Sequential[model]
| 3/17 [00:00<00:00, 1687.40it/s, OpInfo: name = CNN/Sequential[model]
| 4/17 [00:00<00:00, 1883.18it/s, OpInfo: name = CNN/Sequential[model]
| 5/17 [00:00<00:00, 1940.01it/s, OpInfo: name = CNN/Sequential[model]
| 6/17 [00:00<00:00, 2050.84it/s, OpInfo: name = CNN/Sequential[model]
| 7/17 [00:00<00:00, 2171.45it/s, OpInfo: name = CNN/Sequential[model]
| 8/17 [00:00<00:00, 2364.32it/s, OpInfo: name = CNN/Sequential[model]
| 9/17 [00:00<00:00, 2490.19it/s, OpInfo: name = CNN/Sequential[model]
| 10/17 [00:00<00:00, 2529.58it/s, OpInfo: name = CNN/Sequential[model]
| 11/17 [00:00<00:00, 2601.19it/s, OpInfo: name = CNN/Sequential[model]
| 12/17 [00:00<00:00, 2736.90it/s, OpInfo: name = CNN/Sequential[model]
| 13/17 [00:00<00:00, 2831.78it/s, OpInfo: name = CNN/Sequential[model]
| 14/17 [00:00<00:00, 2733.97it/s, OpInfo: name = CNN/Sequential[model]
| 15/17 [00:00<00:00, 2672.55it/s, OpInfo: name = CNN/Sequential[model]
| 16/17 [00:00<00:00, 2767.95it/s, OpInfo: name = CNN/Sequential[model
| 17/17 [00:00<00:00, 2889.87it/s, OpInfo: name = CNN/Sequential[model
] /Linear[15]/611, type = addmm
[VAIQ_NOTE]: =>Doing weights equalization...
[VAIQ_NOTE]: =>Quantizable module is generated./build/quant_model/CNN.py
[VAIQ_NOTE]: =>Get module with quantization.
Numero di immagini valutate: 10000
Test set: Accuracy: 9757/10000 (97.57%)
[VAIQ_NOTE]: =>Exporting quant config./build/quant_model/quant_info.json
```

Al Quantizer di Vitis AI - fase di calibrazione - modello LeNet

Per **VGG16** si ottiene:

```
[VAIQ_NOTE]: Quantization calibration process start up...
[VAIQ_NOTE]: =>Quant Module is in 'cpu'.
[VAIQ_NOTE]: =>Parsing CNN...
[VAIQ_NOTE]: Start to trace model...
[VAIQ_NOTE]: Finish tracing.
[VAIQ_NOTE]: Processing ops...
| 1/22 [00:00<00:00, 3551.49it/s, OpInfo: name = CNN/Sequential[model]
| 2/22 [00:00<00:00, 1463.47it/s, OpInfo: name = CNN/Sequential[model]
| 3/22 [00:00<00:00, 1734.62it/s, OpInfo: name = CNN/Sequential[model]
| 4/22 [00:00<00:00, 1662.26it/s, OpInfo: name = CNN/Sequential[model]
| 5/22 [00:00<00:00, 1623.56it/s, OpInfo: name = CNN/Sequential[model]
| 6/22 [00:00<00:00, 1824.27it/s, OpInfo: name = CNN/Sequential[model]
| 7/22 [00:00<00:00, 1790.14it/s, OpInfo: name = CNN/Sequential[model]
| 8/22 [00:00<00:00, 1876.75it/s, OpInfo: name = CNN/Sequential[model]
| 9/22 [00:00<00:00, 2022.22it/s, OpInfo: name = CNN/Sequential[model]
| 10/22 [00:00<00:00, 1889.42it/s, OpInfo: name = CNN/Sequential[model
| 11/22 [00:00<00:00, 2034.36it/s, OpInfo: name = CNN/Sequential[model
| 12/22 [00:00<00:00, 2139.50it/s, OpInfo: name = CNN/Sequential[model
| 13/22 [00:00<00:00, 2141.80it/s, OpInfo: name = CNN/Sequential[model
| 14/22 [00:00<00:00, 2176.36it/s, OpInfo: name = CNN/Sequential[model
| 15/22 [00:00<00:00, 2272.68it/s, OpInfo: name = CNN/Sequential[model
| 16/22 [00:00<00:00, 2339.27it/s, OpInfo: name = CNN/Sequential[model
| 17/22 [00:00<00:00, 2281.70it/s, OpInfo: name = CNN/Sequential[model
| 18/22 [00:00<00:00, 2255.34it/s, OpInfo: name = CNN/Sequential[model
| 19/22 [00:00<00:00, 2289.20it/s, OpInfo: name = CNN/Sequential[model
| 20/22 [00:00<00:00, 2291.03it/s, OpInfo: name = CNN/Sequential[model
| 21/22 [00:00<00:00, 2342.19it/s, OpInfo: name = CNN/Sequential[model
| 22/22 [00:00<00:00, 2373.75it/s, OpInfo: name = CNN/Sequential[model
] /Linear[20]/821, type = addmm
[VAIQ_NOTE]: =>Doing weights equalization...
[VAIQ_NOTE]: =>Quantizable module is generated./build/quant_model/CNN.py
[VAIQ_NOTE]: =>Get module with quantization.
Numero di immagini valutate: 10000
Test set: Accuracy: 9827/10000 (98.27%)
[VAIQ_NOTE]: =>Exporting quant config./build/quant_model/quant_info.json
```

Al Quantizer di Vitis AI - fase di calibrazione - modello VGG16

■ test

```
python -u quantize.py -d ${BUILD_DIR} --quant_mode test 2>&1 | tee ${LOG_DIR}/quant_test.log
```

Per **LeNet** si ottiene:

```
[VAIQ_NOTE]: Quantization test process start up...
[VAIQ_NOTE]: =>Quant Module is in 'cpu'.
[VAIQ_NOTE]: =>Parsing CNN...
[VAIQ_NOTE]: Start to trace model...
[VAIQ_NOTE]: Finish tracing.
[VAIQ_NOTE]: Processing ops...
| 1/17 [00:00<00:00, 1779.51it/s, OpInfo: name = CNN/Sequential[model]
| 2/17 [00:00<00:00, 1454.84it/s, OpInfo: name = CNN/Sequential[model]
| 3/17 [00:00<00:00, 1557.10it/s, OpInfo: name = CNN/Sequential[model]
| 4/17 [00:00<00:00, 1751.82it/s, OpInfo: name = CNN/Sequential[model]
| 5/17 [00:00<00:00, 1832.21it/s, OpInfo: name = CNN/Sequential[model]
| 6/17 [00:00<00:00, 1896.16it/s, OpInfo: name = CNN/Sequential[model]
| 7/17 [00:00<00:00, 2026.37it/s, OpInfo: name = CNN/Sequential[model]
| 8/17 [00:00<00:00, 2202.89it/s, OpInfo: name = CNN/Sequential[model]
| 9/17 [00:00<00:00, 2326.58it/s, OpInfo: name = CNN/Sequential[model]
| 10/17 [00:00<00:00, 2340.83it/s, OpInfo: name = CNN/Sequential[model]
| 11/17 [00:00<00:00, 2284.25it/s, OpInfo: name = CNN/Sequential[model]
| 12/17 [00:00<00:00, 2390.71it/s, OpInfo: name = CNN/Sequential[model]
| 13/17 [00:00<00:00, 2459.67it/s, OpInfo: name = CNN/Sequential[model]
| 14/17 [00:00<00:00, 772.11it/s, OpInfo: name = CNN/Sequential[model]
| 15/17 [00:00<00:00, 791.96it/s, OpInfo: name = CNN/Sequential[model]
| 16/17 [00:00<00:00, 835.00it/s, OpInfo: name = CNN/Sequential[model]
| 17/17 [00:00<00:00, 875.26it/s, OpInfo: name = CNN/Sequential[model]

/Linear[15]/611, type = addmm]
[VAIQ_NOTE]: =>Doing weights equalization...
[VAIQ_NOTE]: =>Quantizable module is generated.(./build/quant_model/CNN.py)

[VAIQ_NOTE]: =>Get module with quantization.
Numero di immagini valutate: 10000
Test set: Accuracy: 9739/10000 (97.39%)

[VAIQ_NOTE]: =>Converting to xmodel ...
[VAIQ_NOTE]: =>Successfully convert 'CNN' to xmodel.(./build/quant_model/CNN_int.xmodel)
```

Al Quantizer di Vitis AI - fase di test - modello LeNet

Per VGG16 si ottiene:

```
[VAIQ_NOTE]: Quantization test process start up...
[VAIQ_NOTE]: =>Quant Module is in 'cpu'.
[VAIQ_NOTE]: =>Parsing CNN...
[VAIQ_NOTE]: Start to trace model...
[VAIQ_NOTE]: Finish tracing.
[VAIQ_NOTE]: Processing ops...
| 1/22 [00:00<00:00, 3130.08it/s, OpInfo: name = CNN/Sequential[model]
| 2/22 [00:00<00:00, 1883.81it/s, OpInfo: name = CNN/Sequential[model]
| 3/22 [00:00<00:00, 1575.82it/s, OpInfo: name = CNN/Sequential[model]
| 4/22 [00:00<00:00, 1542.45it/s, OpInfo: name = CNN/Sequential[model]
| 5/22 [00:00<00:00, 1629.24it/s, OpInfo: name = CNN/Sequential[model]
| 6/22 [00:00<00:00, 1823.61it/s, OpInfo: name = CNN/Sequential[model]
| 7/22 [00:00<00:00, 1989.84it/s, OpInfo: name = CNN/Sequential[model]
| 8/22 [00:00<00:00, 2092.44it/s, OpInfo: name = CNN/Sequential[model]
| 9/22 [00:00<00:00, 2247.08it/s, OpInfo: name = CNN/Sequential[model]
| 10/22 [00:00<00:00, 2364.45it/s, OpInfo: name = CNN/Sequential[model]
| 11/22 [00:00<00:00, 2031.14it/s, OpInfo: name = CNN/Sequential[model]
| 12/22 [00:00<00:00, 2186.10it/s, OpInfo: name = CNN/Sequential[model]
| 13/22 [00:00<00:00, 2001.54it/s, OpInfo: name = CNN/Sequential[model]
| 14/22 [00:00<00:00, 1983.06it/s, OpInfo: name = CNN/Sequential[model]
| 15/22 [00:00<00:00, 2055.76it/s, OpInfo: name = CNN/Sequential[model]
| 16/22 [00:00<00:00, 2131.05it/s, OpInfo: name = CNN/Sequential[model]
| 17/22 [00:00<00:00, 2046.18it/s, OpInfo: name = CNN/Sequential[model]
| 18/22 [00:00<00:00, 1997.50it/s, OpInfo: name = CNN/Sequential[model]
| 19/22 [00:00<00:00, 2532.63it/s, OpInfo: name = CNN/Sequential[model]
| 20/22 [00:00<00:00, 2583.73it/s, OpInfo: name = CNN/Sequential[model]
| 21/22 [00:00<00:00, 2654.70it/s, OpInfo: name = CNN/Sequential[model]
| 22/22 [00:00<00:00, 2705.84it/s, OpInfo: name = CNN/Sequential[model]

]/Linear[20]/821, type = addmm]
[VAIQ_NOTE]: =>Doing weights equalization...
[VAIQ_NOTE]: =>Quantizable module is generated.(./build/quant_model/CNN.py)

[VAIQ_NOTE]: =>Get module with quantization.
Numero di immagini valutate: 10000
Test set: Accuracy: 9848/10000 (98.48%)

[VAIQ_NOTE]: =>Converting to xmodel ...
[VAIQ_NOTE]: =>Successfully convert 'CNN' to xmodel.(./build/quant_model/CNN_int.xmodel)
```

Al Quantizer di Vitis AI - fase di test - modello VGG16

common.py

Il modulo common.py è utilizzato come modulo di supporto, contenente delle funzioni di utilità usate da quantizer.py e target.py, per cui è necessario importare e utilizzare le funzioni e le classi definite in esso tramite la direttiva `from common import *function*`. In particolare, all'interno di common.py le funzioni principali sono:

- **Classe CNN:** definisce il modello di rete neurale convoluzionale (CNN) con gli strati convoluzionali, normalizzazione, attivazione e pooling utilizzati per la costruzione del modello floating point;
- **Funzione test:** Valuta le performance del modello utilizzando un set di dati di test. Carica le immagini da una cartella specificata, effettua le predizioni del modello su di esse e calcola l'accuratezza del modello.

Compiler

La fase successiva alla quantizzazione prevede la compilazione del modello in formato .xmodel.

target.py

La compilazione si occupa di tradurre le istruzioni in un formato compatibile con la DPU utilizzata sulla ZCU102, in particolare, per fare ciò essa necessita del file arch.json generato da Vivado durante l'implementazione dell'IP core della DPU. Quindi, la compilazione avviene tramite il seguente comando:

```

compile() {
    vai_c_xir \
    --xmodel      ${BUILD}/quant_model/CNN_int.xmodel \
    --arch        $ARCH \
    --net_name    CNN_${TARGET} \
    --output_dir   ${BUILD}/compiled_model
}

```

Il risultato della compilazione (il file *.xmodel*), viene salvato nella cartella *compiled_model*, come mostrato dalla Figura seguente (la Figura è riferita al modello LeNet, per il modello VGG16 il risultato è il medesimo).

```

COMPILING MODEL FOR ZCU102...
[UNILOG][INFO] Compile mode: dpu
[UNILOG][INFO] Debug mode: function
[UNILOG][INFO] Target architecture: DPUCZDX8G_ISA0_B4096_MAX_BG2
[UNILOG][INFO] Graph name: CNN, with op num: 44
[UNILOG][INFO] Begin to compile...
[UNILOG][INFO] Total device subgraph number 3, DPU subgraph number 1
[UNILOG][INFO] Compile done.
[UNILOG][INFO] The meta json is saved to "/workspace/.build/compiled_model/meta.json"
[UNILOG][INFO] The compiled xmodel is saved to "/workspace/.build/compiled_model/CNN_zcu102.xmodel"
[UNILOG][INFO] The compiled xmodel's md5sum is 4f56a344a2cd4215c407ee7d95c5b9e8, and has been saved to "/workspace/.build/compiled_model/md5sum.txt"
*****
* VITIS_AI Compilation - Xilinx Inc.
*****
MODEL COMPILED

```

AI Compiler di Vitis AI - modello LeNet

Il modulo target invece serve semplicemente ad **impacchettare** opportunamente i file in una cartella, in modo da poter passare il contenuto al target ZCU102 per eseguire il modello. All'interno della cartella *target_zcu102* saranno presenti le immagini da valutare, l'*xmodel* e l'applicazione che usa il modello e calcola l'accuracy ed il throughput del modello stesso.

```

3.6.15 | packaged by conda-forge | (default, Dec 3 2021, 18:49:41)
[GCC 9.4.0]

Command line options:
--build_dir : ./build
--target     : zcu102
--num_images : 10000
--app_dir    : application

Copying application code from application ...
Copying compiled model from ./build/compiled_model/CNN_zcu102.xmodel ...

```

Creazione della cartella per il deploy su ZCU102 - modello LeNet e VGG16

XIR Graph

Vitis AI utilizza la struttura degli **XIR Graph** per rappresentare le reti neurali e le operazioni di calcolo. Questi XIR Graph sono grafi diretti aciclici (*DAG*) in cui i nodi rappresentano le operazioni, come la convoluzione, il ridimensionamento o il fully connected layer. Ogni nodo riceve un tensore in ingresso e produce uno o più tensori in uscita.

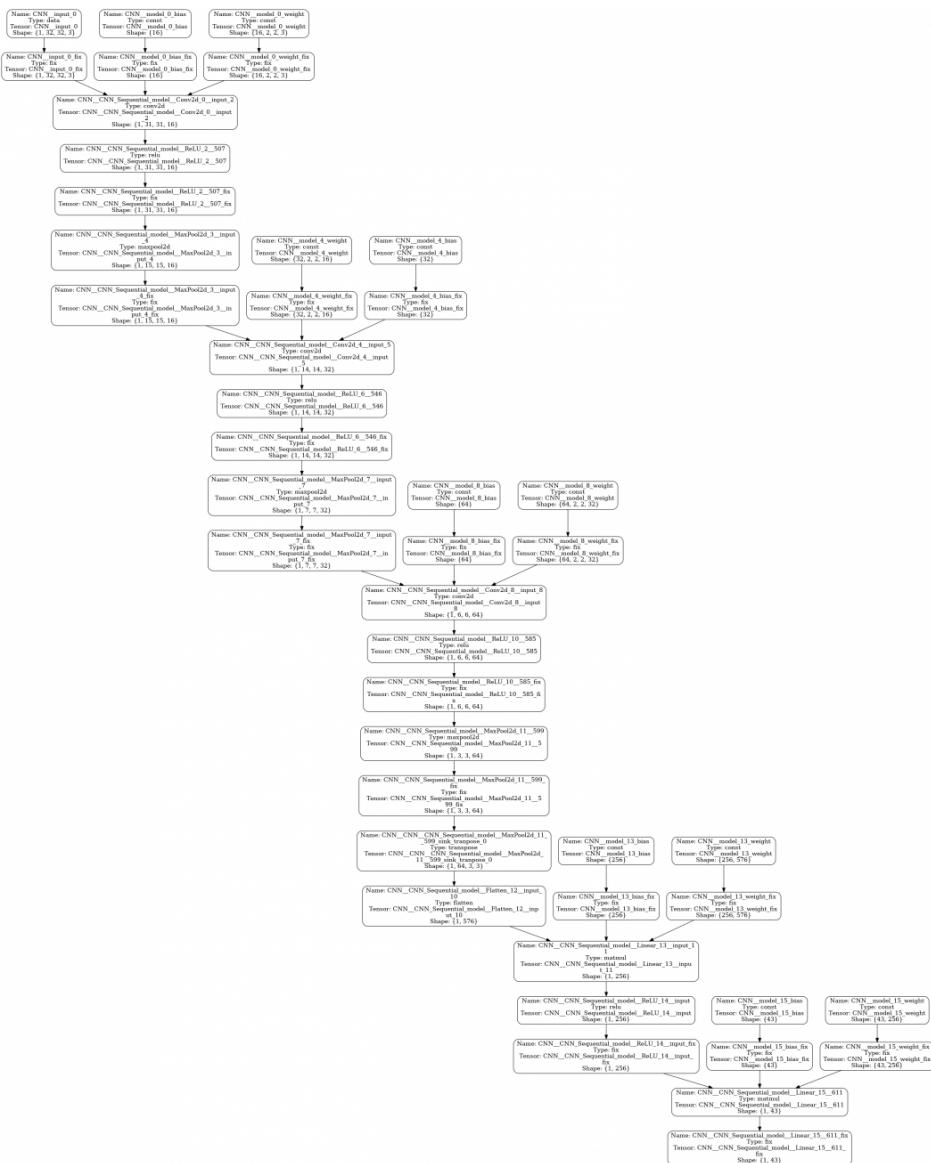
Il concetto di "tensore" è fondamentale in Vitis AI. Si ricorda che il tensore è una struttura multidimensionale che rappresenta i dati di input o output delle operazioni, simile a un array multidimensionale per trasferire e manipolare i dati nel modello.

Vitis AI permette di suddividere il grafo in sottografi distinti, in base al tipo di operazione da eseguire e all'unità di elaborazione disponibile, come la CPU o la DPU. Ogni sottografo viene eseguito su un'unità di elaborazione specifica.

Per visualizzare il grafo XIR associato a un file *.xmodel*, Vitis AI fornisce l'utilità *xdputil*, con cui è possibile generare un *.png* del grafo tramite il seguente comando:

```
xdputil xmodel -p IMAGE_NAME.png XMODEL
```

Questa utility può essere eseguita da linea di comando all'interno del contenitore Docker. Questa utility permette di generare una rappresentazione grafica del grafo XIR, consentendo una migliore comprensione della struttura e dell'organizzazione del modello. Nel caso in esame lo XIR Graph della LeNet sarà quello mostrato nella Figura seguente, in particolare si può notare che il grafo presenta un solo sottografo motivo per il cui l'intero modello può essere eseguito direttamente sulla DPU, senza necessità di utilizzare la CPU ed effettuare successivamente il join delle operazioni.



XIR Graph del modello LeNet

Anche per lo XIR Graph della VGG16, mostrato nella Figura seguente, è presente un solo sottografo per cui il modello esegue interamente sulla DPU. Si noti come il grafo della VGG16 risulti più profondo e complesso rispetto a quello della LeNet, coerentemente con quanto visto in precedenza, ciò risulterà in un minor throughput di questo modello essendo la DPU messa più sotto sforzo.

XIR Graph del modello VGG16

Preparazione per l'esecuzione del modello quantizzato su DPU

Per avviare l'esecuzione di un grafo su DPU, si fa anche in questo caso il reverse engineering del file **app.py**. In primo luogo si importa il file .xmodel utilizzando le API fornite da Vitis AI attraverso:

l'oggetto `xir.Graph` rappresenta il grafo del modello che è stato generato utilizzando Vitis AI, mentre con `deserialize` significa ricostruire un oggetto `xir.Graph` a partire dai dati

Per ogni grafo è possibile creare un runner, ovvero un oggetto che rappresenta il motore di inferenza per l'esecuzione del modello quantizzato su una specifica DPU. Il motore di

Questo oggetto *Runner* rappresenta il motore di inferenza per l'esecuzione del modello quantizzato sulla **DPU** (Deep Learning Processor Unit). La funzione `run` permette di eseguire la logica del modello sulle unità disponibili della scheda.

Ogni runner mette a disposizione un insieme di informazioni sui tensori di ingresso e di uscita, che consentono di eseguire l'elaborazione dei dati prima e dopo l'esecuzione su

```
inputTensor = runner.get_input_tensors()[0]
outputTensors = runner.get_input_tensors()
inputFractionalBits = runner.get_attr("fix_point")
inputName = inputTensors.name
inputDim = inputTensors.dims
# ...
```

Il nome e la dimensione dei tensori di output tornano utili per discriminare i tensori generati dalla DPU dopo l'esecuzione del grafo. Infine, il runner può essere eseguito all'interno di un thread.

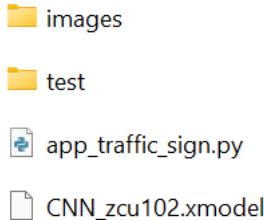
```
job_id = runner.execute_async(inputData,outputData)
runner.wait(job_id)
```

È importante notare che ogni runner viene associato a un core specifico della DPU e che l'assegnazione dei thread ai core avviene automaticamente. Se si desidera forzare l'assegnazione di un thread a un core specifico, è necessario istanziare thread fittizi corrispondenti ai core desiderati.

Arrivati a questo punto si è pronti per caricare il modello sulla ZCU102.

Configurazione e caricamento del modello sulla ZCU102

A valle del processo di compilazione del modello e creazione dell'applicazione da eseguire sul target, la cartella *target_zcu102* sarà composta dai file da caricare sulla board, come mostrato in Figura.



Contenuto della cartella *target_zcu102*

Sarà pertanto necessario spostare il contenuto di questa cartella dall'host alla board attraverso il comando *scp*, che trasferisce i file tramite Ethernet. Se la board di valutazione ZCU102 è connessa alla stessa rete del computer host attraverso l'Ethernet, la cartella *target_zcu102* può essere copiata sulla ZCU102 utilizzando il seguente comando:

```
scp -r ./build/target_zcu102 root@192.168.1.1:~/gruppo06-23/traffic_sign_lenet
```

assumendo che l'indirizzo IP della scheda di destinazione sia 192.168.1.1 e poi inserire, se richiesta, la password (nel caso in esame "root").

Ecco il contenuto della cartella *target_zcu102* sulla board:

```
root@xilinx-zcu102-2021_2:~# cd gruppo06-23/
root@xilinx-zcu102-2021_2:~/gruppo06-23# ls
resnet50_python traffic_sign_lenet traffic_sign_vgg16 yolo_prova_v3
root@xilinx-zcu102-2021_2:~/gruppo06-23# cd traffic_sign_lenet/
root@xilinx-zcu102-2021_2:~/gruppo06-23/traffic_sign_lenet# ls
target_zcu102
root@xilinx-zcu102-2021_2:~/gruppo06-23/traffic_sign_lenet# cd target_zcu102/
root@xilinx-zcu102-2021_2:~/gruppo06-23/traffic_sign_lenet/target_zcu102# ls
CNN_zcu102.xmodel app_traffic_sign.py images test
root@xilinx-zcu102-2021_2:~/gruppo06-23/traffic_sign_lenet/target_zcu102# |
```

Trasferimento di *target_zcu102* sulla board

Esecuzione dell'applicazione di Signal Traffic Classification sulla ZCU102

L'applicazione può essere avviata nella cartella *target_zcu102* sulla board eseguendo il seguente comando:

```
python3 app_traffic_sign.py -m CNN_zcu102.xmodel --threads (N)
```

L'applicazione si avvierà e dopo alcuni secondi mostrerà la velocità di elaborazione (**throughput**) in frame al secondo e l'**accuracy** come segue:

```
root@xilinx-zcu102-2021_2:~/gruppo06-23/traffic_sign_lenet/target_zcu102# python3 app_traffic_sign.py -m CNN_zcu102.xmodel
Command line options:
--image_dir : images
--threads   : 1
--model     : CNN_zcu102.xmodel
--eval      : evaluation
-----
Pre-processing 10000 images...
Starting 1 threads...
Throughput=3593.36 fps, total frames = 10000, time=2.7829 seconds
Correct:9772, Wrong:228, Accuracy:0.9772
Memory: 83.765625 kb
```

Esecuzione del modello sulla ZCU102

Oltre al throughput in frame/sec e l'accuracy, viene mostrata anche l'occupazione di memoria in kB. Con *--threads* si può selezionare il numero di threads per l'esecuzione parallela del modello con l'obiettivo di migliorare il throughput.

Valutazione dei risultati

In primo luogo, si prova a valutare se il modello sulla board sia in grado di predire i segnali stradali di test. I 5 segnali stradali usati per testare i due modelli sono:



Immagini di test

Ecco il risultato:

```

root@xilinx-zcu102-2021_2:~/gruppo06-23/traffic_sign_lenet/target_zcu102# python3 app_traffic_sign.py -m CNN_zcu102.xmodel --threads
4 --image_dir test/ --eval prediction
Command line options:
--image_dir : test/
--threads : 4
--model : CNN_zcu102.xmodel
--eval : prediction
-----
Pre-processing 5 images...
-----
Starting 4 threads...
-----
Throughput=1488.48 fps, total frames = 5, time=0.0034 seconds
Correct:5, Wrong:0, Accuracy:1.0000
[Traffic Sign Recognized: Speed limit (30km/h)
Traffic Sign Recognized: No passing
Traffic Sign Recognized: Bicycles crossing
Traffic Sign Recognized: End of no passing by vehicles over 3.5 metric tons
Traffic Sign Recognized: Yield]
Memory: 31.81500 kb

```

Predizioni ottenute - modello LeNet

```

root@xilinx-zcu102-2021_2:~/gruppo06-23/traffic_sign_vgg16/target_zcu102# python3 app_traffic_sign.py -m CNN_zcu102.xmodel --threads
4 --image_dir test/ --eval prediction
Command line options:
--image_dir : test/
--threads : 4
--model : CNN_zcu102.xmodel
--eval : prediction
-----
Pre-processing 5 images...
-----
Starting 4 threads...
-----
Throughput=485.99 fps, total frames = 5, time=0.0123 seconds
Correct:5, Wrong:0, Accuracy:1.0000
[Traffic Sign Recognized: Speed limit (30km/h)
Traffic Sign Recognized: No passing
Traffic Sign Recognized: Bicycles crossing
Traffic Sign Recognized: End of no passing by vehicles over 3.5 metric tons
Traffic Sign Recognized: Yield]
Memory: 117.14200 kb

```

Predizioni ottenute - modello VGG16

Si può notare che entrambi i modelli predicono in **maniera corretta** i 5 segnali stradali, fornendo inoltre, anche il tipo di cartello riconosciuto.

Analisi delle prestazioni dell'applicazione sulla ZCU102

Si è pertanto pronti ad eseguire il modello sulla DPU per valutarne le differenze. Si è scelto di eseguire inizialmente sulla board con la configurazione single core e poi multi core variando il numero di threads.

DPU single core

Durante l'esecuzione con 1 thread, si osserva che la rete LeNet ha un'accuratezza del 97% e un throughput di circa 3593 frame al secondo. Aumentando il numero di thread a 4, non si registra un incremento nell'accuratezza, ma il throughput aumenta con una media di circa 5529 frame al secondo. Tuttavia, nel caso della rete VGG16, si ottiene un miglioramento meno significativo in termini di throughput, pur avendo un'accuratezza leggermente migliore del 98%.

La differenza nel throughput tra la rete LeNet e la rete VGG16 può essere attribuita alla loro complessità e alla quantità di calcoli richiesti. La rete VGG16 è più profonda e ha un numero maggiore di parametri rispetto alla LeNet, il che richiede un maggiore tempo di elaborazione. Di conseguenza, il throughput della VGG16 è leggermente inferiore rispetto alla LeNet. Tuttavia, la VGG16 compensa questa diminuzione del throughput con un'accuratezza leggermente migliore.

```

root@xilinx-zcu102-2021_2:~/gruppo06-23/traffic_sign_lenet/target_zcu102# python3 app_traffic_sign.py -m CNN_zcu102.xmodel
Command line options:
--image_dir : images
--threads : 1
--model : CNN_zcu102.xmodel
--eval : evaluation
-----
Pre-processing 10000 images...
-----
Starting 1 threads...
-----
Throughput=3593.36 fps, total frames = 10000, time=2.7829 seconds
Correct:9772, Wrong:228, Accuracy:0.9772
Memory: 83.765625 kb

```

DPU single core - 1 thread - modello LeNet

```

root@xilinx-zcu102-2021_2:~/gruppo06-23/traffic_sign_lenet/target_zcu102# python3 app_traffic_sign.py -m CNN_zcu102.xmodel --threads
4
Command line options:
--image_dir : images
--threads : 4
--model : CNN_zcu102.xmodel
--eval : evaluation
-----
Pre-processing 10000 images...
-----
Starting 4 threads...
-----
Throughput=5529.12 fps, total frames = 10000, time=1.8086 seconds
Correct:9772, Wrong:228, Accuracy:0.9772
Memory: 84.003906 kb

```

DPU single core - 4 threads - modello LeNet

```

root@xilinx-zcu102-2021_2:~/gruppo06-23/traffic_sign_vgg16/target_zcu102# python3 app_traffic_sign.py -m CNN_zcu102.xmodel
Command line options:
--image_dir : images
--threads : 1
--model : CNN_zcu102.xmodel
--eval : evaluation
-----
Pre-processing 10000 images...
-----
Starting 1 threads...
-----
Throughput=333.46 fps, total frames = 10000, time=29.9886 seconds
Correct:9839, Wrong:161, Accuracy:0.9839
Memory: 117.839844 kb

```

DPU single core - 1 thread - modello VGG16

```

root@xilinx-zcu102-2021_2:~/gruppo06-23/traffic_sign_vgg16/target_zcu102# python3 app_traffic_sign.py -m CNN_zcu102.xmodel --threads
4
Command line options:
--image_dir : images
--threads   : 4
--model     : CNN_zcu102.xmodel
--eval      : evaluation
-----
Pre-processing 10000 images...
Starting 4 threads...
Throughput=353.58 fps, total frames = 10000, time=28.2823 seconds
Correct:9839, Wrong:161, Accuracy:0.9839
Memory: 118.093750 kb

```

DPU single core - 4 threads - modello VGG16

DPU multi core

Eseguendo lo stesso modello con la DPU multi core, si aspettava di avere ulteriori miglioramenti, ma anche in questo caso i miglioramenti sono poco apprezzabili.

Un possibile motivo può essere che il modello in questione non richiede un carico di lavoro molto intenso o computazionalmente complesso e pertanto l'utilizzo di una DPU multi core potrebbe non portare a miglioramenti significativi. In altre parole, la capacità di calcolo aggiuntiva offerta dalla DPU multi core potrebbe non essere completamente sfruttata dal modello.

```

root@xilinx-zcu102-2021_2:~/gruppo06-23/traffic_sign_lenet/target_zcu102# python3 app_traffic_sign.py -m CNN_zcu102.xmodel
Command line options:
--image_dir : images
--threads   : 1
--model     : CNN_zcu102.xmodel
--eval      : evaluation
-----
Pre-processing 10000 images...
Starting 1 threads...
Throughput=3559.38 fps, total frames = 10000, time=2.8095 seconds
Correct:9772, Wrong:228, Accuracy:0.9772
Memory: 83.855469 kb

```

DPU multi core - 1 thread - modello LeNet

```

root@xilinx-zcu102-2021_2:~/gruppo06-23/traffic_sign_lenet/target_zcu102# python3 app_traffic_sign.py -m CNN_zcu102.xmodel --threads
4
Command line options:
--image_dir : images
--threads   : 4
--model     : CNN_zcu102.xmodel
--eval      : evaluation
-----
Pre-processing 10000 images...
Starting 4 threads...
Throughput=5631.97 fps, total frames = 10000, time=1.7756 seconds
Correct:9772, Wrong:228, Accuracy:0.9772
Memory: 84.003906 kb

```

DPU multi core - 4 threads - modello LeNet

```

root@xilinx-zcu102-2021_2:~/gruppo06-23/traffic_sign_vgg16/target_zcu102# python3 app_traffic_sign.py -m CNN_zcu102.xmodel
Command line options:
--image_dir : images
--threads   : 1
--model     : CNN_zcu102.xmodel
--eval      : evaluation
-----
Pre-processing 10000 images...
Starting 1 threads...
Throughput=331.46 fps, total frames = 10000, time=30.1694 seconds
Correct:9839, Wrong:161, Accuracy:0.9839
Memory: 118.046875 kb

```

DPU multi core - 1 thread - modello VGG16

```

root@xilinx-zcu102-2021_2:~/gruppo06-23/traffic_sign_vgg16/target_zcu102# python3 app_traffic_sign.py -m CNN_zcu102.xmodel --threads
4
Command line options:
--image_dir : images
--threads   : 4
--model     : CNN_zcu102.xmodel
--eval      : evaluation
-----
Pre-processing 10000 images...
Starting 4 threads...
Throughput=486.66 fps, total frames = 10000, time=20.5482 seconds
Correct:9839, Wrong:161, Accuracy:0.9839
Memory: 118.089844 kb

```

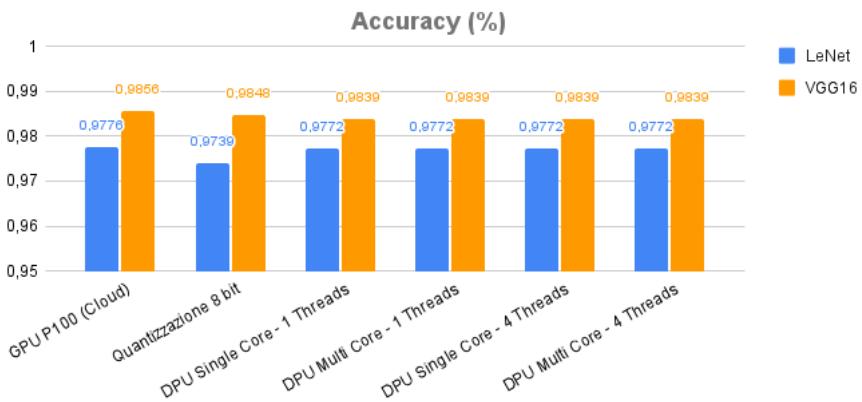
DPU multi core - 4 threads - modello VGG16

Confronto dei risultati ottenuti con quelli dell'implementazione su cloud Kaggle

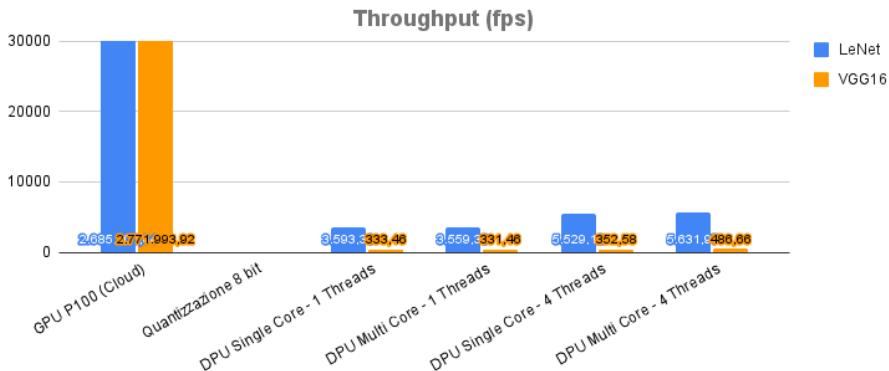
Si mostrano i dati complessivi delle performance delle rispettive configurazioni ed esecuzioni:

Piattaforma HW		Throughput (fps)		Accuracy (%)	
		LeNet	VGG16	LeNet	VGG16
Cloud	GPU P100 (Cloud)	2,685,901.64	2,771,993.92	0.9776	0.9856
Host	Quantizzazione 8 bit			0.9739	0.9848
ZCU102	DPU Single Core - 1 Threads	3,593.36	333.46	0.9772	0.9839
	DPU Multi Core - 1 Threads	3,559.38	331.46	0.9772	0.9839
	DPU Single Core - 4 Threads	5,529.12	352.58	0.9772	0.9839
	DPU Multi Core - 4 Threads	5,631.97	486.66	0.9772	0.9839

Da cui i grafici:



Comparazione Accuracy (%) - LeNet vs VGG16

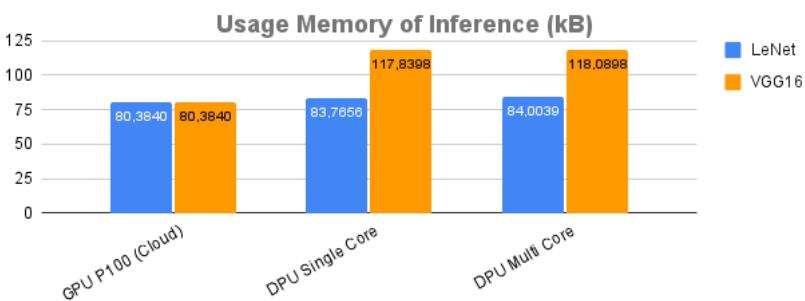


Comparazione Throughput (fps) - LeNet vs VGG16

E la tabella relativa alla memoria usata dai 2 modelli per eseguire le inferenze rispetto al target:

Piattaforma HW		Usage Memory of Inference (kB)	
		LeNet	VGG16
Cloud	GPU P100 (Cloud)	80.3840	80.3840
	DPU Single Core	83.7656	117.8398
ZCU102	DPU Multi Core	84.0039	118.0898

Da cui il grafico:



Comparazione Memoria usata per inferenza (kB) - LeNet vs VGG16

Conclusioni

Sulla base dei risultati e delle osservazioni fatte durante il progetto, è possibile trarre alcune conclusioni sull'utilizzo della Zynq UltraScale+ MPSoC (versione ZCU102) nell'ambito di applicazioni di Edge AI. In particolare, si è riscontrato che l'**accuracy** ottenuta dal modello sulla DPU è all'incirca uguale a quella ottenuta dal modello eseguito in Cloud, tuttavia, il throughput (calcolato come frame/sec) risulta notevolmente più basso rispetto alla soluzione Cloud.

Questo conduce alla scelta di un trade off, infatti, essendo questa applicazione rivolta all'uso a bordo di una vettura a guida autonoma è fondamentale avere sia un'accuracy alta che un throughput adeguato come ci risulta nel modello LeNet nel caso in esame. Infatti, nel caso della DPU multi core con 4 Thread il sistema riesce ad effettuare la recognition del segnale stradale in 0,17756 ms senza considerare ulteriori vincoli, quali la distanza a cui il modello riesce ad eseguire la recognition del segnale stradale ed altro.

In generale, quindi, nonostante la quantizzazione effettuata durante il workflow di Vitis-AI, il modello mantiene comunque un buon livello di accuratezza nelle predizioni effettuate, questo è un notevole vantaggio nel momento in cui l'applicazione di Edge AI ha come obiettivo principale quello di essere il più preciso possibile senza grossi vincoli temporali.

Il limite maggiore di questo approccio, invece, risiede nei vincoli tecnologici imposti dalla Zynq UltraScale+; infatti, pur avendo integrato una DPU che effettua un'accelerazione delle operazioni di convoluzione per le reti neurali, il throughput non risulta ancora comparabile con quello di una soluzione in Cloud.

Costi di utilizzo

Nel caso in esame, il costo complessivo di realizzazione è prevalentemente dovuto dal costo oneroso della evaluation board ZCU102 che si aggira intorno ai migliaia di euro. Si ricorda che la valutazione dei costi e delle possibili soluzioni dipenderà dalle specifiche esigenze del progetto e dalle risorse finanziarie disponibili. È importante condurre un'analisi approfondita per identificare le migliori opzioni in termini di prestazioni, scalabilità e costi che la board offre.

Possibili sviluppi futuri

Una possibile direzione per lo sviluppo futuro è l'esplorazione dell'utilizzo di modelli neurali diversi e applicazioni specifiche che possano beneficiare dell'accelerazione hardware offerta dalla board ZCU102. Inoltre, si potrebbero esplorare opportunità di integrazione con altre tecnologie, come l'IoT (Internet of Things) o l'Edge Computing, al fine di creare soluzioni più complesse e scalabili. Ad esempio, la board ZCU102 potrebbe essere utilizzata come parte di un sistema di intelligenza artificiale distribuita per il controllo di dispositivi IoT o per l'elaborazione di dati sul campo, nonché per la fase di detection del segnale stradale oltre a quella di recognition affrontata in questo elaborato.

Bibliografia

1. Vitis™ AI User Guide (<https://docs.xilinx.com/r/2.0-English/ug1414-vitis-ai/Vitis-AI-Overview>)
2. Documentation DPU for Zynq UltraScale+ MPSoC (https://docs.xilinx.com/r/en-US/pg338-dpu?tocId=3xsG16y_QFTWvAJKHbisEw)
3. Vitis™ AI Library User Guide (<https://docs.xilinx.com/r/2.0-English/ug1354-xilinx-ai-sdk/Introduction>)
4. PetaLinux Tools Documentation Reference Guide (<https://docs.xilinx.com/r/2021.1-English/ug1144-petalinux-tools-reference-guide/Overview>)
5. Repository Completa Zynq Ultrascale+ degli Autori: Antonio Romano - Giuseppe Riccio (https://github.com/LaErre9/Zynq_Ultrascale_Vitis_AI)

Retrieved from "http://www.naplespu.com/es/index.php?title=Configurazione,_installazione_ed_esecuzione_di_una_CNN_sulla_DPU_della_Zynq_Ultrascale%2B:_Valutazione_delle_prestazioni_rispetto_ad_una_GPU_in_Cloud&oldid=75"

This page was last edited on 24 July 2023, at 19:40.