ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

Corso di Laurea Magistrale in Informatica

Sviluppo di una pipeline C++ per l'analisi in tempo reale di dati per missioni spaziali con integrazione di Machine Learning su piattaforme Edge

Relatore:
Prof. Giuseppe Lisanti

Presentata da:
Luca Spadoni

Correlatori:
Dott. Luca Castaldini
Andrea Bulgarelli, PhD

Sessione Dicembre
Anno Accademico 2024/2025

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

Corso di Laurea Magistrale in Informatica

Sviluppo di una pipeline C++ per l'analisi in tempo reale di dati per missioni spaziali con integrazione di Machine Learning su piattaforme Edge

Relatore:
Prof. Giuseppe Lisanti

Presentata da:
Luca Spadoni

Correlatori:
Dott. Luca Castaldini
Andrea Bulgarelli, PhD

Sessione Dicembre
Anno Accademico 2024/2025

Indice

Abstract

Questa tesi presenta lo sviluppo, l'ottimizzazione e la validazione del framework di analisi in tempo reale "Real-Time Analysis DataProcessor (RTA-DP)", progettato presso INAF – OAS per il processamento distribuito di dati relativi a raggi gamma atmosferici. Il lavoro ha riguardato la revisione dell'architettura multi-thread e della gestione del ciclo di vita dei thread, l'introduzione di code concorrenti sicure, l'integrazione di ZeroMQ per lo streaming dei dati e l'ottimizzazione del codice per limitare il consumo di CPU e memoria, con l'obiettivo di garantire stabilità e bassa latenza in ambienti edge. La pipeline è stata estesa per il caso d'uso GammaSky, integrando un modello di machine learning per la ricostruzione delle waveform, quantizzato e ottimizzato per l'esecuzione su edge computer Jetson Orin Nano. I test di inferenza, gli stress test e il monitoraggio delle risorse mostrano latenze dell'ordine dei 400 µs, throughput superiori ai requisiti e consumo energetico minimo. Sono stati inoltre sviluppati test di integrazione end-to-end che verificano l'intero flusso di processamento dei dati. I risultati dimostrano che la pipeline è stabile, efficiente e adatta a scenari "resource constrained", costituendo una base solida per il deployment all'interno del setup di GammaSky presso l'Osservatorio del Monte Cimone e per futuri impieghi in missioni spaziali con nanosatelliti.

Introduzione

L'astronomia dei raggi gamma rappresenta uno dei campi più affascinanti e sfidanti della ricerca moderna. I raggi gamma sono fotoni ad altissima energia, la forma più energetica della radiazione elettromagnetica, e la loro osservazione consente di indagare fenomeni estremi dell'Universo. Un singolo fotone gamma può viaggiare per miliardi di anni prima di interagire con la materia, mentre gli eventi astrofisici che generano emissioni gamma presentano scale temporali molto variabili: i *Gamma-Ray Burst* (GRB), ad esempio, possono andare da pochi millisecondi a diverse ore. Esistono inoltre fenomeni gamma di origine atmosferica come i *Terrestrial Gamma-ray Flashes* (TGF), brevissimi ma estremamente energetici. La rivelazione di tali segnali richiede strumenti ad alta sensibilità e sistemi di acquisizione capaci di registrare eventi molto rapidi, insieme a tecniche avanzate di elaborazione dati in grado di distinguere le emissioni di interesse dal rumore di fondo e ottenere una ricostruzione accurata del fenomeno per comprenderne i meccanismi fisici sottostanti.

Negli ultimi anni, la crescente disponibilità di tecnologie avanzate di calcolo a basso consumo ha aperto nuove prospettive per l'analisi dei dati direttamente a bordo degli strumenti scientifici. I rivelatori di raggi gamma possono infatti essere installati su diverse piattaforme: dai grandi osservatori spaziali ai satelliti di piccole dimensioni, fino a postazioni a terra. In questo contesto, il paradigma dell'*edge computing* consente di eseguire algoritmi complessi su piattaforme embedded con risorse limitate, riducendo la quantità di dati da trasmettere a terra e migliorando l'autonomia operativa delle

missioni. Ciò risulta particolarmente rilevante per sistemi con capacità di comunicazione limitate, come i nanosatelliti *CubeSat*, dove l'elaborazione in loco permette di ottimizzare l'uso della banda disponibile e di selezionare e inviare solo i dati scientificamente più significativi.

All'interno di questo contesto si inserisce il progetto Real-Time Analysis Dataprocessor (RTA-DP), sviluppato presso l'Osservatorio di Astrofisica e Scienza dello Spazio (OAS) di Bologna, una sede dell'Istituto Nazionale di Astrofisica (INAF). RTA-DP è un framework per l'elaborazione dati distribuita, orientato all'implementazione di pipeline software di processamento dati in streaming e fornisce strumenti per il controllo e il monitoraggio del sistema. I dati che il framework tratta sono dati acquisiti da strumenti per la rivelazione di raggi gamma e processati in tempo reale dalle varie pipeline. Originariamente sviluppato in Python, è stato successivamente migrato in C++ per garantire migliori prestazioni, gestione più efficiente delle risorse e compatibilità con piattaforme a ridotta capacità computazionale.

Durante il periodo di tirocinio svolto presso OAS, il lavoro si è concentrato sullo sviluppo, l'ottimizzazione e la validazione della versione C++ del framework. Sono state affrontate problematiche legate alla concorrenza, alla sincronizzazione tra thread e alla gestione della memoria, con il duplice obiettivo di ridurre la latenza e minimizzare il consumo di risorse. Nel corso di questo lavoro di tesi, inoltre, la pipeline RTA-DP è stata estesa per eseguire processing di eventi gamma tramite modelli di machine learning ottimizzati per ambienti edge, con lo scopo di ricostruire in tempo reale le proprietà fisiche degli eventi acquisiti. Tale approccio rientra nell'edge AI, ossia l'applicazione di algoritmi di intelligenza artificiale in prossimità della fonte dei dati. Tecniche di ottimizzazione come la quantizzazione sono state adottate per adattare i modelli alle capacità computazionali delle piattaforme edge impiegate. I test sono stati condotti su una scheda prodotta da NVIDIA chiamata Jetson Orin Nano, la quale rappresenta un buon compromesso tra prestazioni e consumo energetico e che è stata configurata per

ospitare tutta la pipeline di analisi e ricostruzione, dalla ricezione dei dati raw, passando per la fase di inferenza in tempo reale, fino alla scrittura dei parametri fisici ricostruiti su file in output.

Questa tesi presenta dunque lo sviluppo, l'ottimizzazione e l'estensione di una pipeline di analisi real-time orientata all'utilizzo in missioni spaziali con risorse limitate. Oltre a descrivere i fondamenti scientifici e tecnologici alla base del progetto, verranno illustrate le soluzioni implementative, le sfide affrontate e i risultati sperimentali ottenuti, con particolare attenzione al ruolo dell'edge AI e alla prospettiva, sempre più attuale, dell'elaborazione scientifica avanzata direttamente a bordo dello strumento.
La presente tesi è suddivisa in sei capitoli:

- **Capitolo 1** introduce il contesto scientifico e tecnologico del lavoro. Vengono presentati i Terrestrial Gamma-ray Flashes (TGF), le missioni spaziali che li hanno studiati, i rivelatori a scintillazione utilizzati per la rivelazione dei raggi gamma ed il ruolo crescente dell'edge computing nelle missioni spaziali.

- **Capitolo 2** descrive il progetto GammaSky e il relativo setup sperimentale, illustrando l'architettura hardware e software basata su detector a scintillazione SiPM, scheda Red Pitaya e piattaforma Jetson Orin Nano, motivando l'utilizzo di tecniche di machine learning nella ricostruzione degli eventi e descrivendo i formati di input e output del sistema.

- **Capitolo 3** fornisce una panoramica dell'architettura generale del framework RTA-DP, descrivendone le librerie utilizzate, i principali componenti e i meccanismi di configurazione, comunicazione e monitoraggio che abilitano la costruzione di pipeline di processamento in tempo reale.

- **Capitolo 4** presenta lo sviluppo e l'ottimizzazione della versione C++ del framework, discutendo le limitazioni iniziali dell'implementazione Python e approfondendo la gestione dei thread e del loro ciclo di vita, l'introduzione di code thread-

safe, l'integrazione di ZeroMQ per lo streaming dei dati, la gestione dei pacchetti binari provenienti dal modulo DAM e il sistema di logging configurabile, oltre alle principali ottimizzazioni finalizzate alla riduzione del consumo di risorse.

- **Capitolo 5** illustra l'estensione di RTA-DP al caso d'uso GammaSky. È descritto il modello di machine learning usato per la stima dell'area delle waveform, il processo di quantizzazione del modello per l'esecuzione su edge computer, l'integrazione del modello nella pipeline C++ e la generazione dei file `DL2` a partire dai pacchetti `R0`.

- **Capitolo 6** descrive la fase di testing e di analisi delle prestazioni della pipeline RTA-DP: vengono presentati i benchmark di inferenza del modello ML su Jetson Orin Nano, il monitoraggio dell'utilizzo delle risorse hardware in scenari di idle e di pieno carico, e i test di integrazione end-to-end che verificano il corretto funzionamento dell'intero flusso `R0` $\rightarrow$ inferenza ML $\rightarrow$ `DL2`.

Capitolo 1

Contesto scientifico e tecnologico

Questo capitolo raccoglie le basi utili per i capitoli applicativi: un richiamo su raggi gamma e Terrestrial Gamma-ray Flashes (TGF), una panoramica delle missioni che li hanno osservati, i principi dei rivelatori a scintillazione (PMT e SiPM) e, infine, il ruolo dell'edge computing a bordo (in particolare sui CubeSat) e il suo impatto sulle scelte architetturali e sulle ottimizzazioni software discusse nei capitoli successivi.

1.1 Astronomia dei raggi gamma

I raggi gamma sono fotoni altamente energetici, con energie superiori a 100 keV[1], e rappresentano la porzione più estrema dello spettro elettromagnetico. Quest'ultimo si estende infatti dalle onde radio, caratterizzate dalle energie più basse, passando per la luce visibile, fino ai raggi gamma, che possiedono le energie più elevate. In astronomia, lo studio dei raggi gamma si concentra su energie comprese tra alcune centinaia di keV e diversi PeV. La loro propagazione in linea retta nello spazio consente di determinare con buona precisione la sorgente di emissione. I raggi gamma possono originare da numerosi fenomeni astrofisici, come resti di supernova, pulsar, nuclei galattici attivi e Gamma-

[1] 1 eV (electronvolt) corrisponde all'energia cinetica acquisita da un elettrone accelerato da una differenza di potenziale di 1 volt, pari a circa $1,602 \cdot 10^{-19}$ J (joule).

ray Bursts (GRB). Questi rientrano nella più ampia categoria dei fenomeni transienti, ovvero eventi astrofisici o atmosferici di breve durata e difficilmente prevedibili, che comprendono anche lampi di raggi X e lampi radio veloci (FRB). Tuttavia, non si tratta di un'esclusiva cosmica: raggi gamma vengono prodotti anche sulla Terra, in particolare durante intensi temporali atmosferici. In questo contesto si inseriscono i Terrestrial Gamma-ray Flashes (TGF).

1.1.1 Terrestrial Gamma-ray Flashes

I TGF sono improvvisi lampi di raggi gamma prodotti tra l'alta troposfera e la stratosfera terrestre, generati da temporali intensi e intercettati da satelliti in orbita terrestre bassa. Si manifestano come emissioni estremamente brevi, in genere da poche decine fino a qualche centinaio di microsecondi, con rari casi che raggiungono circa un millisecondo[1], e sono tipicamente associate all'attività elettrica dei fulmini. Questa tipologia di fenomeni si distingue nettamente dai GRB per la scala temporale: mentre i TGF durano soltanto frazioni di millisecondo, i GRB presentano una grande varietà di durate, che possono andare da pochi millisecondi fino a diversi minuti e, in casi eccezionali (gli ultra-long GRB), anche a ore[2].

I TGF furono osservati per la prima volta nel 1994 dal Compton Gamma Ray Observatory[3], una missione NASA lanciata nel 1991 per studiare raggi gamma, e successivamente confermati da missioni come AGILE (Astrorivelatore Gamma a Immagini LEggero, ASI/INAF)[4] e dal Fermi Gamma-ray Space Telescope[5]. Si stima che, ogni giorno, circa 500 TGF vengano prodotti a livello globale (uno per ogni mille fulmini), ma la maggior parte non venga rilevata, lasciando questa stima come incerta. Le valutazioni più recenti indicano valori ancora più elevati, addirittura di circa 400.000 l'anno[6]. Dal punto di vista energetico, le prime osservazioni effettuate dal satellite RHESSI avevano mostrato che lo spettro dei TGF si estendeva fino a circa 20 MeV[7], mentre le misure più recenti del satellite AGILE hanno rivelato componenti ad energie ancora più elevate,

fino a 40 MeV e, in casi estremi, prossime a 100 MeV[8], dimostrando che le energie coinvolte nei TGF possono essere molto più elevate di quelle inizialmente osservate.

Alcuni studi hanno mostrato che l'origine dei raggi gamma nei TGF si trova a circa 10–25 km di altitudine, quindi all'interno delle nubi temporalesche nelle profondità dell'atmosfera[9]. La distribuzione geografica dei TGF presenta un picco nelle regioni tropicali, suggerendo che i temporali tipici di queste zone, caratterizzati da forti correnti convettive, siano in grado di generarli[10]. I temporali infatti, agiscono come veri e propri acceleratori naturali di particelle. Nello specifico, i Terrestrial Gamma-ray Flashes si originano all'interno di questi fenomeni quando i campi elettrici prodotti dalla tempesta raggiungono intensità tali da estendersi per diversi chilometri nell'atmosfera. In queste condizioni, gli elettroni liberi presenti nell'aria vengono accelerati a velocità prossime a quella della luce[11]. Durante la loro corsa, tali elettroni interagiscono con i nuclei degli atomi dell'atmosfera, emettendo fotoni gamma tramite bremsstrahlung[2], producendo l'impulso osservato in orbita. Questo processo genera quindi lampi di raggi gamma estremamente energetici e di brevissima durata, emessi dalla regione temporalesca. Alcune osservazioni hanno inoltre mostrato che, come conseguenza dei TGF, possono essere prodotti anche positroni, cioè le antiparticelle degli elettroni. Questi flussi di antimateria riescono a propagarsi nello spazio, seguendo le linee del campo magnetico terrestre in un fascio piuttosto stretto, dove possono essere rilevati dai satelliti che passano sopra la zona[12].

Il processo di formazione dei Terrestrial Gamma-ray Flashes viene mostrato in Figura 1.1.

[2]Il bremsstrahlung (dall'inglese "braking radiation") è la radiazione emessa da particelle cariche, come elettroni, quando vengono decelerate o deviate dall'interazione con campi elettrici di nuclei atomici. Questo fenomeno è responsabile della produzione di fotoni ad alta energia.

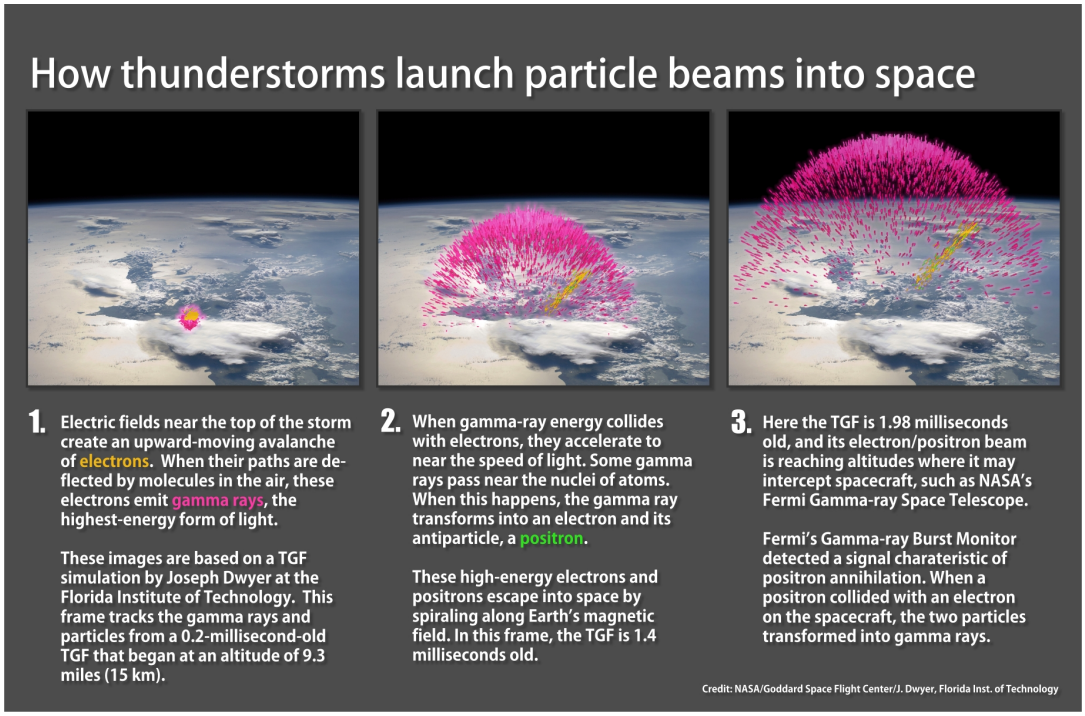


Figura 1.1: Sequenza illustrativa del processo di formazione dei Terrestrial Gamma-ray Flashes (TGF) durante un temporale (vedi fonte in figura).

1.2 Missioni spaziali per lo studio dei TGF

Dalla loro scoperta nel 1994, i Terrestrial Gamma-ray Flashes sono stati oggetto di numerose osservazioni condotte da diverse missioni spaziali. Ciascun satellite ha contribuito con strumenti e capacità uniche, permettendo di migliorare progressivamente la comprensione del fenomeno: dalla semplice identificazione degli eventi iniziali, alla determinazione della loro distribuzione globale e origine. Nei paragrafi seguenti vengono ripercorse le principali tappe di questo percorso, attraverso le missioni più rilevanti.

Compton Gamma Ray Observatory (CGRO)

La scoperta dei Terrestrial Gamma-ray Flashes risale al 1994, grazie al Burst and Transient Source Experiment (BATSE) a bordo del Compton Gamma Ray Observatory

(CGRO) della NASA[3]. BATSE era stato concepito per lo studio dei Gamma-ray Burst cosmici, fenomeni di durata molto più lunga, ma si rivelò in grado di identificare anche lampi gamma terrestri estremamente rapidi. Nel corso dei suoi nove anni di attività, lo strumento rilevò 76 eventi di TGF[13], fornendo la prima conferma sperimentale dell'esistenza di questo fenomeno. Sebbene il numero di rilevamenti fosse limitato, questi risultati aprirono la strada a un nuovo campo di ricerca, mostrando che l'atmosfera terrestre, in condizioni particolari, può comportarsi come una sorgente di radiazione gamma di altissima energia.

RHESSI

Un passo cruciale nello studio dei TGF avvenne con il lancio del satellite *Reuven Ramaty High Energy Solar Spectroscopic Imager* (RHESSI)[7] nel 2002. Progettato principalmente per lo studio dei brillamenti solari, RHESSI permise di raccogliere un numero molto maggiore di eventi rispetto a CGRO e, soprattutto, di analizzarne lo spettro energetico con maggiore precisione. Le osservazioni mostrarono che i TGF si estendono fino a circa 20 MeV, fornendo così le prime misure dirette dell'energia dei fotoni emessi.

Fermi Gamma-ray Space Telescope

Il lancio del *Fermi Gamma-ray Space Telescope* nel 2008 segnò un ulteriore salto di qualità. Lo strumento *Gamma-ray Burst Monitor* (GBM)[14], progettato per la rivelazione di Gamma-ray Bursts, si è dimostrato particolarmente adatto anche allo studio dei TGF, grazie al suo ampio campo di vista e alla sensibilità agli impulsi di durata molto breve. In pochi anni, Fermi ha raccolto migliaia di eventi, portando la stima del tasso globale a centinaia di TGF al giorno[5].

AGILE

Un contributo decisivo è arrivato anche dal satellite italiano *AGILE*[15], lanciato nel 2007. Il suo strumento *Mini-Calorimeter* (MCAL)[16] si è dimostrato particolarmente efficace nello studio dei TGF grazie a una sensibilità spettrale estesa e a un'elevata risoluzione temporale (nell'ordine del microsecondo). Le osservazioni di AGILE hanno permesso non solo di stabilire che lo spettro dei TGF si estende fino a oltre 40 MeV, con eventi eccezionali prossimi ai 100 MeV, ma anche di ottenere le prime localizzazioni spaziali dirette e di evidenziare correlazioni temporali con i fulmini[8]. Questi risultati hanno rappresentato una svolta nello studio del fenomeno, indicando che i TGF sono più frequenti ed energetici di quanto ipotizzato inizialmente.

Atmosphere-Space Interactions Monitor (ASIM)

L'*Atmosphere-Space Interactions Monitor* (ASIM)[17] è un esperimento installato nel 2018 sul modulo Columbus della Stazione Spaziale Internazionale (ISS)[3] e rappresenta la prima missione dedicata principalmente allo studio dei TGF. Il suo payload scientifico comprende il *Modular X- and Gamma-ray Sensor* (MXGS), sensibile ai fotoni fino a circa 20 MeV, e il *Modular Multi-Imaging Assembly* (MMIA), in grado di registrare l'emissione ottica associata ai fenomeni temporaleschi. ASIM ha permesso di effettuare le prime osservazioni simultanee di TGF e scariche elettriche nei temporali, mostrando in modo diretto la connessione tra i lampi gamma e l'attività dei fulmini. Inoltre, grazie alla combinazione di misure nei raggi gamma e nel visibile, è stato possibile caratterizzare con maggiore dettaglio lo spettro e la dinamica temporale dei TGF[18]. L'orbita bassa della ISS, che sorvola frequentemente le regioni equatoriali e tropicali, ha garantito ad ASIM una posizione privilegiata per lo studio di questi fenomeni, che si manifestano con

[3]La Stazione Spaziale Internazionale è una piattaforma orbitante abitata permanentemente dal 2000, frutto della cooperazione tra diverse agenzie spaziali, che orbita a circa 400 km di quota e funge principalmente da laboratorio in assenza di gravità.

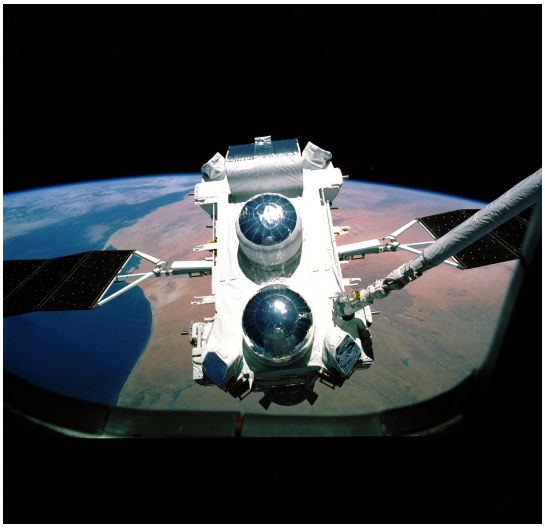
maggiore frequenza proprio in tali aree. Ad oggi, ASIM costituisce lo strumento più avanzato per l'osservazione dei TGF.

LIGHT-1 CubeSat

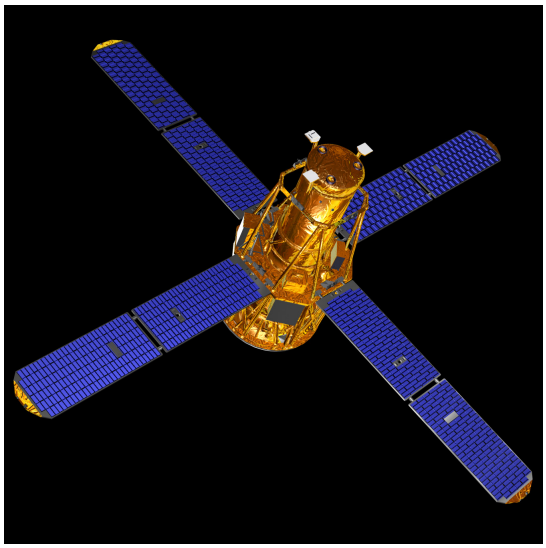
Un esempio rilevante per questo lavoro di tesi è rappresentato dal CubeSat[4] *LIGHT-1* (noto anche come RAADSat), una missione congiunta del Bahrein e degli Emirati Arabi Uniti. Il satellite, di tipo 3U, è stato lanciato nel dicembre 2021 e trasportato sulla Stazione Spaziale Internazionale, dove è stato rilasciato in orbita il 3 febbraio 2022 e ha fatto rientro nell'atmosfera nel gennaio 2023. A bordo, LIGHT-1 ospitava il payload RAAD (Rapid Acquisition Atmospheric Detector), progettato per la rivelazione dei TGF con alta risoluzione temporale (fino a 500 ns). RAAD utilizzava cristalli scintillatori ad alta risoluzione accoppiati a fotomoltiplicatori convenzionali (PMT) e fotomoltiplicatori al silicio (SiPM), per confrontare diverse configurazioni di rivelazione in orbita[19] (per una descrizione dettagliata dei due tipi di rivelatore si veda la Sezione 1.3). L'elaborazione dei dati avveniva direttamente a bordo del satellite grazie a un sistema elettronico dedicato: i segnali prodotti dai rivelatori venivano convertiti in forma digitale e confrontati con soglie prestabilite per identificare i fotoni di interesse. Quando venivano rilevati più fotoni in un intervallo di tempo molto breve, il sistema li riconosceva come possibile evento di tipo TGF e ne registrava con precisione sia l'energia sia il tempo di arrivo. I dati così raccolti venivano poi trasmessi a Terra, dove venivano analizzati in dettaglio e messi in relazione con altre osservazioni, come i cataloghi di fulmini e i dati di altre missioni[19]. Questa missione dimostra che è possibile integrare sistemi per la rilevazione di TGF su CubeSat dalle dimensioni compatte, confermando la fattibilità dell'approccio studiato nella presente tesi.

[4]I CubeSat sono nanosatelliti standardizzati, introdotti all'inizio degli anni 2000, caratterizzati da un formato modulare basato su unità (U) di $10 \times 10 \times 10$ cm. Le configurazioni più comuni sono 1U, 3U e 6U. Grazie al basso costo di realizzazione e lancio, vengono ampiamente utilizzati per missioni scientifiche, tecnologiche ed educative.

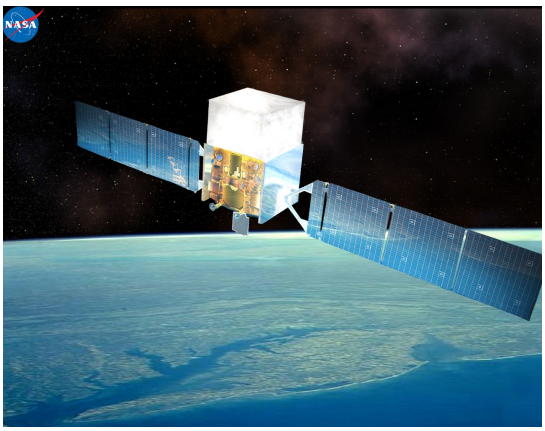
Le principali missioni spaziali dedicate allo studio dei TGF sono illustrate in Figura 1.2.



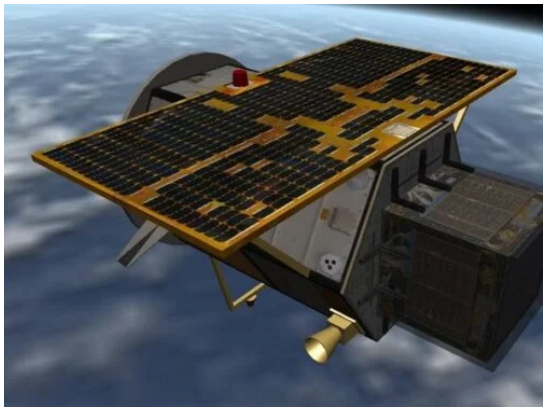
(a) Compton Gamma Ray Observatory (CGRO) – 1991.



(b) Reuven Ramaty High Energy Solar Spectroscopic Imager (RHESSI) – 2002.



(c) Fermi Gamma-ray Space Telescope – 2008.



(d) Astrorivelatore Gamma a Immagini LEggero (AGILE) – 2007.

(e) Atmosphere–Space Interactions Monitor (ASIM, ISS) – 2018.

(f) LIGHT-1 3U CubeSat – 2021 (Crediti: Bahrain NSSA).

Figura 1.2: Principali missioni spaziali che hanno contribuito allo studio dei Terrestrial Gamma-ray Flashes (TGF).

1.3 Rivelatori a scintillazione per raggi gamma

Lo studio dei Terrestrial Gamma-ray Flashes (TGF) richiede rivelatori capaci di misurare eventi di brevissima durata e di alta energia con efficienza e precisione. Una delle tecnologie più diffuse è quella basata sui materiali (cristalli) scintillatori, che emettono luce visibile quando attraversati da radiazione ionizzante. L'intensità della luce prodotta è proporzionale all'energia depositata nel materiale, rendendo possibile ricostruire informazioni sia temporali che energetiche sull'evento di interesse. Per rendere utilizzabile questa luce, agli scintillatori vengono accoppiati dispositivi di conversione ottico-elettronica chiamati fotomoltiplicatori (PM). Questi trasformano linearmente i fotoni in elettroni e ne amplificano la carica di un fattore 10^5–10^9, producendo un segnale elettrico misurabile. Storicamente, i più usati sono i tubi fotomoltiplicatori (PMT), ma negli ultimi anni i Silicon Photomultiplier (SiPM) hanno guadagnato un ruolo centrale

grazie ai loro vantaggi in termini di compattezza, robustezza e consumi.

1.3.1 PMT

I *Photomultiplier Tubes* (PMT) sono fotomoltiplicatori a tubo e rappresentano la tecnologia tradizionale per la rivelazione della luce di scintillazione. Essi sono costituiti da un fotocatodo, un sistema di elettrodi noti come dinodi e un anodo. Questi componenti sono racchiusi in un tubo a vuoto. I PMT sfruttano l'effetto fotoelettrico: i fotoni incidenti colpiscono il fotocatodo, generando elettroni che vengono moltiplicati attraverso la catena di dinodi, producendo un segnale elettrico amplificato e facilmente misurabile[20] (vedi Figura 1.3). I PMT offrono un'elevata sensibilità, bassa rumorosità e ottima risoluzione temporale, caratteristiche che li hanno resi lo standard per decenni.

Tuttavia, presentano alcuni limiti come dimensioni e peso elevati, necessità di alte tensioni di alimentazione (centinaia o migliaia di volt) e fragilità meccanica, in quanto si tratta di dispositivi a vuoto. Per queste ragioni, sebbene ancora utilizzati in esperimenti ground-based di grande scala, i PMT risultano poco adatti a missioni spaziali di piccole dimensioni o a scenari con sistemi compatti a basso consumo tipo CubeSat[21].

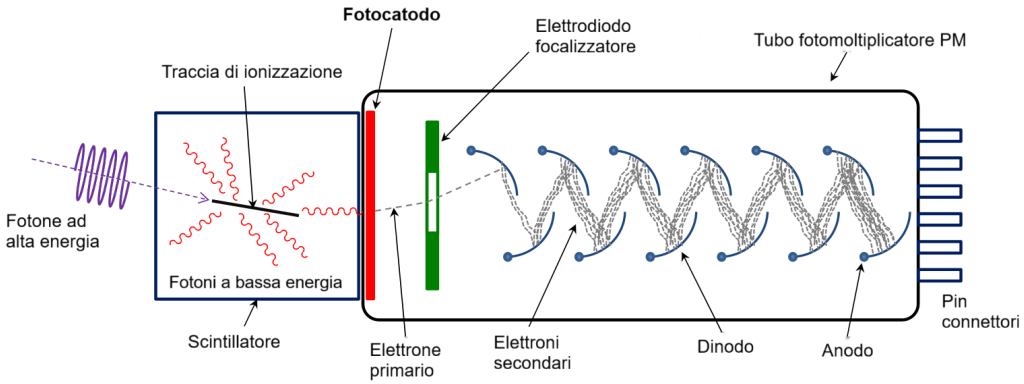


Figura 1.3: Schema di un fotomoltiplicatore accoppiato ad uno scintillatore (PMT) per la rivelazione dei raggi gamma[22].

1.3.2 SiPM

I *Silicon Photomultipliers* (SiPM) sono fotomoltiplicatori a stato solido costituiti da un semiconduttore al silicio (vedi Figura 1.4). Essi sono rivelatori costituiti da matrici di fotodiodi a valanga (*Avalanche Photodiode*, APD), impiantate direttamente sul semiconduttore e operanti in modalità Geiger[5]. Ciascuna microcella funziona come un contatore binario: quando viene colpita da un fotone genera un segnale "acceso/spento". L'uscita complessiva del SiPM è quindi proporzionale al numero di celle attivate nello stesso momento, permettendo una misura diretta dell'intensità del segnale luminoso[23]. Rispetto ai tradizionali fotomoltiplicatori (PMT), i SiPM offrono diversi vantaggi: sono molto più compatti e leggeri e richiedono tensioni di alimentazione molto più basse (tipicamente tra 20 e 70 V). A questo si aggiungono una notevole robustezza meccanica, che li rende resistenti a vibrazioni e urti, e una buona tolleranza a un ampio intervallo di temperature operative. D'altra parte, i SiPM presentano alcune limitazioni rispetto ai PMT: sono più sensibili al rumore termico (dark count rate) e tendono a saturarsi più rapidamente in presenza di eventi molto intensi, a causa del numero finito di microcelle disponibili. Nonostante ciò, i continui progressi tecnologici hanno reso questi limiti sempre meno significativi, confermando i SiPM come una delle soluzioni più promettenti per applicazioni di rivelazione di fotoni in astrofisica e in contesti spaziali[21].

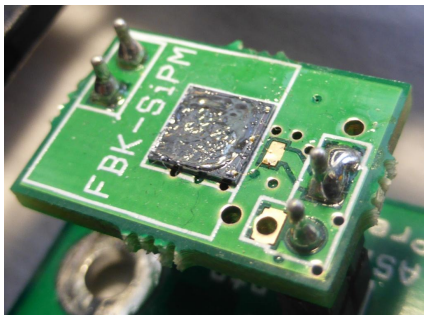


Figura 1.4: Fotomoltiplicatore al silicio (SiPM) realizzato da FBK, montato su circuito stampato (PCB) (Da: PhysicsOpenLab).

[5]Per modalità Geiger si intende un regime di funzionamento in cui basta l'arrivo di un singolo fotone per accendere una microcella e produrre un impulso elettrico sempre uguale.

1.4 Edge computing nei CubeSat e nelle missioni spaziali

L'edge computing in ambito spaziale si riferisce all'elaborazione dei dati direttamente a bordo del satellite, invece di inviarli grezzi a Terra per il processamento. Ciò è particolarmente vantaggioso per CubeSat e piccoli satelliti, che spesso hanno finestre di comunicazione molto brevi e banda limitata verso le stazioni di terra. Eseguendo analisi e processing dei dati in tempo reale sul payload orbitale, si riduce la latenza tra l'osservazione e la disponibilità a Terra dell'informazione[24]. Questo è un aspetto critico in applicazioni in cui serve una risposta tempestiva, come ad esempio nel monitoraggio di disastri naturali quali incendi ed eruzioni vulcaniche oppure nel rilevamento di fenomeni astrofisici di brevissima durata, come i TGF. Inoltre, data la ridotta capacità di downlink, è spesso impraticabile trasmettere tutte le osservazioni grezze: conviene selezionare a bordo e inviare solo i dati effettivamente rilevanti al task, massimizzando l'efficacia dell'utilizzo del canale radio.

Oltre allo scenario orbitale, concetti simili possono trovare impiego nell'esplorazione planetaria. Ad esempio, un rover lunare equipaggiato con un rivelatore gamma intelligente potrebbe rilevare fenomeni transienti nel cielo lunare (come GRB lontani) senza dover inviare continuamente dati grezzi alla Terra. In situazioni in cui la comunicazione è limitata (si pensi a un rover sul lato nascosto della Luna o su Marte), dotare lo strumento di capacità decisionali tramite edge AI significa permettergli di reagire localmente ad eventi di interesse: il rover potrebbe autonomamente identificare un aumento anomalo di radiazione e attivare misure di raccolta dati approfondite, oppure scegliere di inviare un segnale di allerta a Terra con priorità. Tali funzionalità aumentano il valore scientifico della missione, massimizzando i dati utili raccolti pur entro i ristretti canali di comunicazione disponibili.

Già nei primi anni 2000 si è intravisto il potenziale dell'elaborazione autonoma a bordo

con la missione NASA EO-1 (Earth Observing-1), la quale sperimentò dei primi algoritmi di intelligenza artificiale on-board per individuare eventi scientifici rilevanti osservati sulla superficie e reagire autonomamente senza attendere i comandi da Terra[25]. Oggi, con l'avvento di costellazioni di decine o centinaia di nanosatelliti, l'approccio tradizionale con satelliti che si limitano a ritrasmettere i dati a terra rischia di non essere scalabile. Studi recenti mostrano che sfruttare l'elaborazione in orbita può assicurare una migliore scalabilità delle infrastrutture spaziali, riducendo la necessità di molte stazioni di terra pur mantenendo la stessa capacità di raccolta dati[26]. In altri termini, una costellazione di piccoli satelliti cooperanti può fungere da sistema distribuito in orbita che filtra ed elabora localmente i dati, inviando a Terra soltanto risultati sintetici o segnalazioni di eventi di interesse.

L'abilitazione dell'edge AI sui CubeSat è dovuta anche ai progressi dell'elettronica embedded. Tradizionalmente i computer di bordo privilegiano affidabilità e tolleranza alle radiazioni rispetto alle prestazioni, risultando lenti e poco adatti a carichi intensivi come quelli di machine learning. Negli ultimi anni si sta invece sperimentando l'uso di componenti commerciali (Commercial Off-the-Shelf Components, COTS)[6] anche nello spazio, accettando una durata missione limitata in cambio di capacità computazionale molto superiore[24]. Numerose missioni hanno verificato l'impiego di processori avanzati low-power in orbita per l'esecuzione di algoritmi di intelligenza artificiale: il CubeSat ESA Φ-Sat-1 ha utilizzato un acceleratore AI Intel Movidius Myriad 2 a bordo[27]; la piattaforma OPS-SAT integra invece FPGA (Field Programmable Gate Arrays) e un SoC (System-on-a-Chip) con CPU dual-core per eseguire algoritmi in tempo reale[28]. Test preliminari indicano che alcuni dispositivi COTS possono operare nello spazio per periodi brevi senza subire danni da radiazioni, aprendo la strada a missioni sperimentali con capacità di calcolo elevate[24].

[6] *Commercial Off-the-Shelf Components*, ovvero componenti elettronici standard reperibili in commercio e non progettati specificamente per l'ambiente spaziale. L'adozione di COTS consente di ridurre costi e tempi di sviluppo, ma introduce maggiori rischi di guasti dovuti a radiazioni e condizioni estreme, rendendo la durata della missione potenzialmente più breve.

Figura 1.5: Il CubeSat ESA Φ-Sat-2 integrato e pronto per il lancio. Si tratta di un satellite di tipo 6U, con dimensioni 22 × 10 × 33 cm e massa al lancio di circa 9 kg (Crediti: Open Cosmos).

Tra le missioni europee che hanno sperimentato con successo l'edge computing e l'edge AI on-board possiamo citare alcune pietre miliari:

Φ-Sat-1 (ESA, 2020)

Primo dimostratore europeo di deep learning a bordo di un CubeSat. In questa missione un chip Intel Myriad 2 (VPU) ha eseguito una rete neurale convoluzionale (CNN) per il rilevamento di nubi (denominata CloudScout), con l'obiettivo di scartare automaticamente le immagini coperte da nuvole, quindi inutilizzabili, prima di trasmetterle a Terra[29]. L'esperimento ha così dimostrato che l'elaborazione a bordo può ridurre significativamente il volume di dati da trasmettere a terra senza sacrificare informazioni utili[30].

OPS-SAT (ESA, 2019–2024)

Un CubeSat 3U progettato come "laboratorio orbitale" per testare software e tecnologie di controllo avanzate. OPS-SAT disponeva di un computer di bordo sperimentale circa dieci volte più potente dei normali computer spaziali, il che ha consentito di eseguire il deployment di space software sotto forma di applicazioni

AI direttamente a bordo del nanosatellite. Ciò è stato reso possibile grazie ad un'architettura che, da remoto, permetteva di installare, aggiornare e gestire app, chiamata NanoSat MO Framework (NMF)[7][32]. Questa piattaforma ha dimostrato la capacità di far girare modelli di machine learning in tempo reale, eseguendo inferenza usando CNN quantizzate post allenamento.

Particolarmente significativi sono i traguardi tecnologici raggiunti da OPS-SAT nell'ambito dell'edge AI: il satellite ha ospitato il primo deployment in orbita di una rete neurale per applicazioni di intelligenza artificiale, ha effettuato per la prima volta l'addestramento a bordo di modelli supervisionati e non supervisionati e ha realizzato il primo ri-addestramento in volo di un modello AI utilizzando dati raccolti dal satellite stesso. Inoltre, OPS-SAT è stato il primo esperimento a riutilizzare reti neurali pre-addestrate sviluppate per applicazioni terrestri, dimostrando la possibilità di trasferire modelli esistenti in un contesto spaziale[33].

L'ESA ha inoltre organizzato una competizione chiamata "OPS-SAT Case" per coinvolgere sviluppatori nel progettare modelli di image classification ottimizzati per l'esecuzione sul satellite[28]. I vincitori di questa challenge hanno caricato le loro app AI a bordo di OPS-SAT, verificando con successo in orbita la classificazione autonoma di scene terrestri (come riconoscimento di suolo agricolo, acque, neve, vegetazione ecc.). Questo caso dimostra l'interesse crescente verso modelli ML custom, pre-allenati e ottimizzati per essere eseguiti su nanosatelliti a limitate capacità prestazionali.

HYPSO-1 (NTNU, 2022)

Un CubeSat 6U per l'osservazione oceanografica che adotta algoritmi di ML a bordo. In particolare, HYPSO-1 sfrutta cosiddette mappe auto-organizzanti (Self-

[7]Il *NanoSat MO Framework* è un software sviluppato dall'ESA per nanosatelliti di tipo CubeSat. Implementa un'architettura modulare basata sullo standard CCSDS Mission Operations (MO) e consente di gestire applicazioni a bordo, eseguire aggiornamenti remoti e monitorare i processi in esecuzione in maniera standardizzata.[31]

Organizing Maps) per rilevare in tempo reale la presenza di fioriture algali potenzialmente dannose nelle acque oceaniche[34]. Elaborando localmente i dati iperspettrali del colore dell'oceano, il satellite è in grado di identificare queste anomalie ecologiche e segnalarle tempestivamente, ottimizzando l'efficacia scientifica della missione senza dover inviare a Terra l'intero flusso di immagini grezze[35].

Φ-Sat-2 (ESA, 2024)

Evoluzione di Φ-Sat-1, è un CubeSat 6U progettato per ospitare a bordo molteplici applicazioni AI in ambito osservazione della Terra, continuando il lavoro che si era aperto con OPS-SAT (vedi Figura 1.5). Grazie a un computer di bordo potenziato (lo stesso usato in Φ-Sat-1) e al framework NanoSat MO, Φ-Sat-2 può eseguire diversi algoritmi in parallelo, elaborando e processando i dati alla fonte[36]. Invece di downlinkare immagini e telemetria grezze, il satellite effettua localmente analisi avanzate e invia a Terra solo informazioni sintetiche essenziali, con benefici rilevanti in efficienza di banda e rapidità decisionale[37].

Φ-Sat-2 è stata lanciata con quattro applicazioni AI attive: un'app di compressione di immagini basata su CNN per ottimizzare la trasmissione dei dati a terra, un algoritmo di rilevamento delle nubi in tempo reale per evitare acquisizioni di immagini inutilizzabili, un'app per l'individuazione e classificazione di imbarcazioni navali per la sorveglianza marittima contro la pesca illegale e infine una per generare street maps a partire da immagini satellitari (Sat2Map) necessarie in situazioni d'emergenza quali alluvioni o terremoti[36]. Due ulteriori applicazioni, vincitrici della OrbitalAI challenge[38], sono state caricate in seguito: un'app di allerta incendi (PhiFire), la quale analizza immagini termiche individuando focolai e zone bruciate in tempo reale e un algoritmo per identificare anomalie negli ecosistemi marini (ad esempio rilevare fioriture algali anomale o sversamenti di petrolio)[39].

Capitolo 2

Il progetto GammaSky

Nel contesto della rivelazione dei Terrestrial Gamma-ray Flashes e dello sviluppo di tecniche di Edge AI, nasce GammaSky. GammaSky è un progetto sperimentale attualmente in fase di sviluppo presso l'INAF – OAS, concepito come un setup sperimentale composto da un rivelatore a scintillazione, un sistema di acquisizione basato su conversione analogico-digitale (ADC) e un'unità di edge computing per il processamento dei dati. L'obiettivo è validare una catena completa di rivelazione ed elaborazione in tempo reale di eventi gamma ad alta energia, gettando le basi per futuri sistemi autonomi da impiegare su piattaforme spaziali compatte. L'idea chiave è spostare l'analisi dei dati direttamente a bordo del dispositivo di acquisizione; in questo modo, si riduce la quantità di informazioni da trasmettere a terra e il sistema può reagire autonomamente ai fenomeni di interesse. GammaSky è pensato come dimostratore tecnologico per futuri progetti in ambito spaziale, come ad esempio nanosatelliti, con un'architettura progettata per mantenere peso e consumi contenuti e capace di elaborare localmente i dati, inviando a Terra solo informazioni filtrate. Ciò consentirebbe di prendere decisioni in tempo reale, ad esempio analizzando autonomamente i segnali di interesse per la missione o inviando allerte scientifiche[1] in presenza di eventi come un TGF o un GRB,

[1]Nel contesto delle missioni scientifiche, per *allerta* si intende una notifica automatica e tempestiva, inviata verso centri di controllo e la comunità scientifica, che segnala il rilevamento di un fenomeno

senza richiedere il downlink dei dati e ulteriore processamento a terra, riducendo i tempi necessari all'emissione dell'allerta.

GammaSky trae vantaggio dall'esperienza maturata con il progetto Gamma-Flash[42] e, al contempo, ambisce a portare avanti il lavoro sui TGF iniziato in INAF – OAS con AGILE, introducendo però significative innovazioni tecnologiche. In particolare, impiega rivelatori allo stato dell'arte e integra modelli di machine learning su piattaforme embedded, per migliorare la precisione e l'autonomia dello strumento. Il progetto mira a dimostrare la possibilità di identificare e analizzare in tempo reale un TGF o un Gamma-ray Glow (GRG)[2] durante un temporale, capacità che con le due missioni sopracitate richiedeva analisi a posteriori. Per testare queste potenzialità in un contesto reale, il setup sperimentale è stato installato all'interno di una cupola protettiva presso l'Osservatorio Climatico "O. Vittori" sul Monte Cimone (2165 m s.l.m.)[3], mostrato in Figura 2.1, un sito ideale per l'osservazione di fenomeni gamma atmosferici da terra. Il team di GammaSky ha inoltre sviluppato librerie software per supportare lo sviluppo, il training e il testing di modelli di machine learning. Fanno parte del "pacchetto", infatti, un simulatore di forme d'onda, librerie con algoritmi standard per la ricostruzione e il processamento dei dati e utilities per la verifica e comparazione delle performance dei modelli rispetto agli algoritmi tradizionali

Il predecessore di GammaSky, Gamma-Flash (ASI/INAF), ha costituito un passo importante verso questo obiettivo. Mediante una rete di rivelatori di raggi gamma e neutroni installati presso l'osservatorio sul Cimone, Gamma-Flash ha permesso di osservare e analizzare un GRG, sebbene solo in fase di post-processing[43]. Questo esperi-

transiente di particolare interesse (ad es. GRB o TGF) e permette di attivare rapidamente osservazioni o analisi di follow-up. Queste allerte possono essere generate in formato machine-readable (notices) o come circolari testuali e vengono distribuite tramite reti dedicate, come il General Coordinates Network (GCN)[40][41].

[2] I *Gamma-ray Glows* sono emissioni di raggi gamma di lunga durata associate a temporali, tipicamente generate da elettroni accelerati nei campi elettrici atmosferici. A differenza dei TGF, che sono brevissimi lampi gamma, i GRG possono durare da frazioni di secondo fino a decine di secondi, risultando quindi più facili da osservare con rivelatori a scintillazione.

[3] `https://www.isac.cnr.it/it/node/7813`

mento ha quindi confermato la fattibilità di studiare da terra i fenomeni gamma legati ai temporali e ha evidenziato la necessità di strumenti più autonomi e capaci di analisi in tempo reale. Il setup sperimentale di GammaSky, rispetto a Gamma-Flash, apporta aggiornamenti sotto un punto di vista tecnologico. Infatti, è stato aggiornato sostituendo i tradizionali PMT con fotomoltiplicatori al silicio (SiPM), più compatti, a basso consumo e con migliore rapporto segnale/rumore. L'uso di cristalli scintillatori accoppiati ai SiPM consente di osservare sia fenomeni atmosferici (TGF, GRG) sia eventi cosmici (GRB), sfruttando algoritmi di trigger e analisi a bordo. Lo sviluppo di modelli di machine learning per la rilevazione dei TGF a terra rappresenta quindi un banco di prova per la futura implementazione di sistemi di AI a bordo di nanosatelliti, aprendo la strada alla rivelazione autonoma di GRB e altri fenomeni transienti direttamente dallo spazio.

Figura 2.1: Parte dell'Osservatorio Climatico "O. Vittori" (ISAC-CNR) sul Monte Cimone (2165 m s.l.m.) che ospita vari esperimenti atmosferici.

2.1 Architettura hardware e setup sperimentale

L'architettura hardware di GammaSky è stata sviluppata con particolare attenzione a consumi contenuti e capacità di elaborazione locale, replicando in piccolo una piattaforma scientifica spaziale e autonoma. In Gamma-Flash, i segnali venivano rivelati tramite tubi fotomoltiplicatori (PMT) e successivamente elaborati su un *Main Control Computer* (MCC) dedicato, in cui veniva eseguita la pipeline software di analisi dati. Per GammaSky è stata adottata una soluzione più adatta ad applicazioni embedded e a future missioni spaziali: i detector PMT sono stati sostituiti con detector SiPM, i quali operano a bassa tensione, mentre il MCC è stato rimpiazzato da un edge computer capace di eseguire modelli di intelligenza artificiale direttamente a bordo. I SiPM richiedono tensioni di alimentazione molto inferiori rispetto ai PMT tradizionali, tanto da poter essere alimentati tramite una semplice porta USB o piccoli alimentatori dedicati. Al contrario, i PMT necessitano di survoltori e circuiti ad alta tensione che non solo aumentano l'ingombro complessivo, ma introducono anche problematiche di isolamento e interferenze elettromagnetiche, potenzialmente in grado di degradare il segnale. L'edge computer, d'altra parte, consente di eseguire in locale i modelli di machine learning ottimizzati per l'analisi in tempo reale. Pur mantenendo un consumo energetico contenuto, deve integrare acceleratori hardware adeguati, come GPU o FPGA nel SoC, in grado di assicurare elevate prestazioni di calcolo e flessibilità per diversi scenari applicativi di edge computing. Il setup sperimentale utilizzato in GammaSky, mostrato in Figura 2.2 all'interno della cupola dell'osservatorio climatico sul Monte Cimone, è quindi composto dai seguenti componenti:

- **Detector SiPM**[4]: il prototipo impiega un rivelatore integrato Luxium basato su cristallo scintillatore di ioduro di sodio (Nal), accoppiato a un fotomoltiplicatore al silicio (SiPM). Il modulo è ottimizzato per applicazioni portatili e a basso consu-

[4] `https://luxiumsolutions.com/radiation-detector-assemblies/sipm-integrated-detectors`

mo, alimentato tramite porta USB a 5 V. Il sistema fornisce un segnale analogico proporzionale all'energia depositata nel cristallo, con un intervallo dinamico compreso tra 9 keV e 6 MeV, coprendo quindi buona parte del range di interesse per la rivelazione di TGF. Il rivelatore mantiene una risposta stabile anche al variare della temperatura, grazie a un circuito di compensazione automatica che corregge il guadagno nell'intervallo operativo -20 °C ÷ +50 °C. Il setup attuale utilizza un singolo rivelatore, sufficiente per la validazione della catena di acquisizione e della pipeline di analisi.

- **Scheda di acquisizione Red Pitaya STEMlab 125-14**[5]: dispositivo basato su SoC Xilinx Zynq-7020 con ADC (Analogue-to-Digital Converter) a 14 bit e frequenza di campionamento di 125 MS/s. La CPU ARM dual-core gestisce il controllo e l'acquisizione dei dati. Un modulo GPS fornisce segnali per la sincronizzazione temporale assoluta, garantendo timestamp con precisione di pochi microsecondi. La Red Pitaya digitalizza il segnale analogico generato dal detector SiPM, tramite l'ADC e lo prepara per il processamento. Il consumo complessivo della scheda è di circa 10 W.

- **Edge computer NVIDIA Jetson Orin Nano**[6]: piattaforma embedded ad alte prestazioni, equipaggiata con una CPU a 6 core ARM, 8 GB di memoria LPDDR5 e una GPU NVIDIA Ampere con 1024 core CUDA e 32 tensor core. Riceve i dati digitalizzati dalla Red Pitaya, li processa in tempo reale ed esegue inferenza con modelli di machine learning ottimizzati. Il profilo di potenza configurabile (minimo 7 W, massimo 25 W) permette di bilanciare performance e consumo energetico, rendendola adatta per l'esecuzione di algoritmi di AI in contesti a risorse limitate. Grazie alla combinazione di CPU multicore e GPU, la Jetson è in grado di gestire

[5] `https://redpitaya.com/stemlab-125-14`

[6] `https://www.nvidia.com/it-it/autonomous-machines/embedded-systems/jetson-orin/nano-super-developer-kit/`

contemporaneamente la pipeline di acquisizione, il pre-processing e l'esecuzione di modelli di deep learning con latenze ridotte.

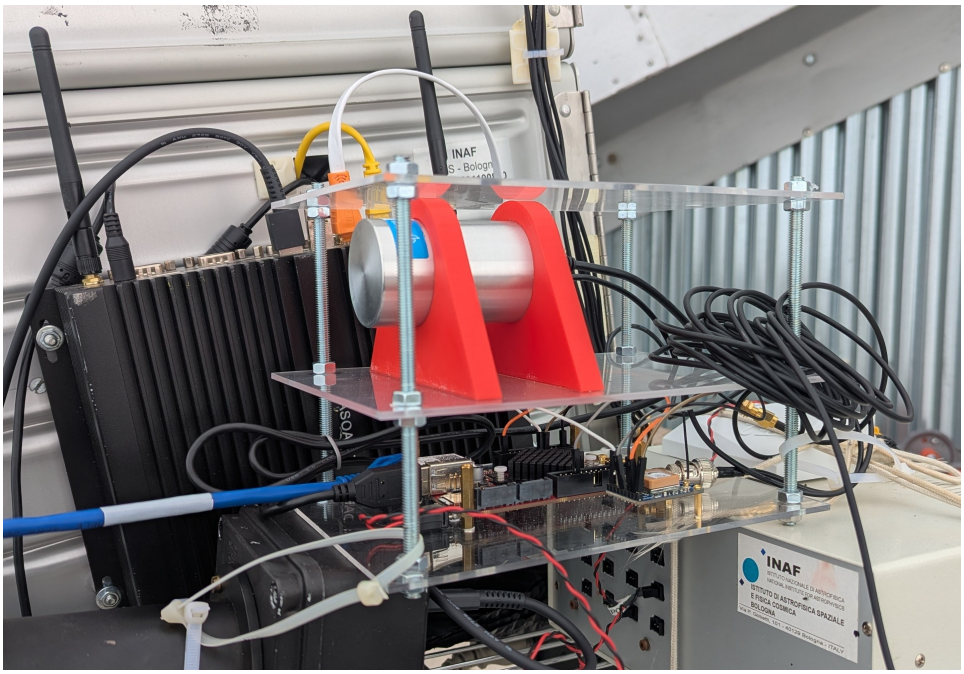


Figura 2.2: Parte del setup sperimentale di GammaSky presente all'interno della cupola dell'osservatorio sul Cimone. In basso si può notare la scheda Red Pitaya, mentre in alto il detector SiPM.

Nel setup installato presso l'osservatorio, il rivelatore è connesso alla scheda Red Pitaya all'interno della cupola, mentre l'unità di elaborazione Jetson Orin Nano, collegata via Ethernet alla Red Pitaya, è collocata nel centro di calcolo dell'osservatorio. La scelta di mantenere la Jetson in un ambiente controllato è dovuta al fatto che la comunicazione tra i due dispositivi avviene tramite rete, rendendo non necessario il montaggio diretto in cupola. In questo modo si evita di esporre il dispositivo a condizioni ambientali potenzialmente critiche, come basse temperature, elevata umidità o formazione di condensa, che potrebbero compromettere l'affidabilità e la durata dell'esperimento.

2.2 Funzionamento del sistema in tempo reale

Dal punto di vista funzionale, GammaSky implementa pipeline di acquisizione e elaborazione dati capaci di operare in streaming continuo, con bassa latenza, così da individuare eventi gamma e analizzarli in real time tramite machine learning. Il flusso dei dati attraverso il sistema avviene in più stadi sequenziali (vedi Figura 2.3):

- **Rivelazione e digitalizzazione**: al momento della rivelazione di un fotone gamma, l'elettronica del rivelatore genera un rapido impulso di tensione, della durata di pochi microsecondi, la cui forma temporale può essere approssimata come la risposta di un filtro $CR-(RC)^n$[44]. Questo segnale analogico viene inviato tramite cavo coassiale alla scheda Red Pitaya[42], dove l'ADC campiona la forma d'onda, producendo un insieme di dati digitali (una traccia temporale discreta) rappresentativo dell'impulso rilevato. Con forma d'onda (waveform in inglese) si intende l'aspetto del segnale generato dall'elettronica del detector, quando questo interagisce con fotoni o elettroni. La waveform contiene informazioni utili per ricostruire energia e tempo dell'evento.

- **Trigger ed estrazione dell'evento**: la traccia digitale passa attraverso una logica di trigger implementata sull'FPGA della Red Pitaya. Questo circuito verifica in tempo reale se il segnale soddisfa certi criteri predefiniti (ad esempio supera una certa soglia). In caso affermativo, l'FPGA "dichiara" un evento valido e isola i dati dell'impulso corrispondente (una finestra di campioni intorno al picco) per l'elaborazione successiva[42]. Questo primo filtraggio hardware permette di ridurre la mole di dati, scartando il rumore di fondo o impulsi non significativi, e garantisce che solo gli eventi potenzialmente interessanti proseguano nella pipeline.

- **Marcatura temporale**: per ogni evento validato, il sistema associa un timestamp assoluto basato sul clock sincronizzato tramite GPS. La parte software di acquisizione in esecuzione sulla CPU ARM della Red Pitaya riceve dal modulo GPS sia

il segnale Pulse Per Second (PPS) sia i pacchetti NMEA 0183[7] trasmessi tramite interfaccia UART. Il software decodifica da questi pacchetti la data e l'ora correnti e utilizza il segnale PPS per sincronizzare il clock interno, compensando le derive temporali dovute alla limitata precisione dell'oscillatore di bordo. In questo modo, a ciascuna forma d'onda acquisita viene associato un riferimento temporale accurato per l'evento rilevato.[42]. In questo modo, ogni flash gamma registrato è identificato univocamente nel tempo, consentendo analisi di coincidenza con eventi (come ad esempio fulmini) registrati da altre stazioni.

- **Trasmissione dei dati al nodo di elaborazione**: i dati dell'evento (che possono includere la forma d'onda grezza oppure parametri riassuntivi detti di "housekeeping") vengono impacchettati e inviati dalla Red Pitaya all'edge computer Jetson Orin Nano per il processing avanzato. La comunicazione avviene attraverso interfaccia di rete (Ethernet), utilizzando il protocollo TCP. In sostanza, quindi, in GammaSky la Red Pitaya agisce da "producer" dei dati, mentre la Jetson da "consumer".

- **Elaborazione edge e inferenza AI**: lo stream di eventi validati e marcati temporalmente viene inviato in tempo reale alla Jetson, dove viene eseguita la pipeline software di analisi (RTA-DP, descritta nel Capitolo successivo). In questa fase, ogni evento è rappresentato dalla sua forma d'onda digitale proveniente dal rivelatore scintillatore. L'obiettivo principale è la ricostruzione accurata dei parametri dell'impulso, in particolare l'energia depositata (area sotto la curva) e il tempo di arrivo, i quali sono fondamentali per identificare e classificare fenomeni gamma transienti. Tradizionalmente, questa ricostruzione viene effettuata con algoritmi deterministici basati sull'integrazione discreta dell'impulso (ad es. con il metodo

[7]Il protocollo *NMEA 0183* è uno standard di comunicazione seriale ampiamente usato nei ricevitori GPS. Fornisce stringhe ASCII contenenti informazioni di tempo, posizione e stato del segnale. Il segnale PPS invece genera un impulso digitale al secondo, sincronizzato con il tempo GPS, che consente di allineare con alta precisione i clock locali.

dei trapezi[8]), i quali richiedono tarature manuali e possono risultare sensibili al rumore[46]. Un limite importante di questi metodi emerge in presenza di eventi in sovrapposizione (pile-up): quando due impulsi si avvicinano nel tempo, la forma risultante può deformarsi in modo imprevedibile, rendendo difficile stimare l'intensità corretta. GammaSky adotta invece reti neurali convoluzionali (CNN) ottimizzate per piattaforme embedded. Questi modelli analizzano direttamente le waveform e stimano i parametri chiave dell'evento in modo più robusto rispetto agli algoritmi deterministici tradizionali, mostrando una maggiore capacità di adattamento in presenza di rumore, distorsioni e variazioni dovute a eventi in pile-up. Esistono già approcci avanzati per la gestione del pile-up in sistemi di rivelazione gamma[47], ma la loro applicazione in contesti di analisi in tempo reale su dispositivi a risorse limitate rappresenta tuttora una sfida aperta.

Le CNN vengono eseguite in inferenza direttamente sulla CPU della Jetson, sfruttando l'elaborazione parallela e mantenendo latenze molto ridotte, compatibili con un sistema real-time a risorse limitate. Questa scelta apre la strada a ulteriori sviluppi: una volta garantita una ricostruzione precisa e veloce dei singoli eventi, sarà possibile applicare modelli di machine learning più complessi, ad esempio per il riconoscimento automatico di sequenze anomale (anomaly detection) o pattern caratteristici di fenomeni come i TGF, con l'obiettivo di generare allerte direttamente a bordo.

[8]Il cosiddetto *metodo dei trapezi* si riferisce al filtro di shaping digitale proposto da Jordanov et al.[45], impiegato per ricostruire impulsi provenienti da amplificatori "charge-sensitive". Il filtro opera sui campioni discreti del segnale e ne trasforma la forma esponenziale in una risposta trapezoidale, la cui altezza risulta proporzionale all'energia dell'evento. Questa tecnica consente di stimare l'ampiezza in modo più stabile e meno sensibile al rumore rispetto a una misura diretta del picco, riducendo gli effetti del cosiddetto "deficit balistico", ma richiede una corretta taratura dei parametri di shaping ed è sensibile a eventi in sovrapposizione (pile-up).

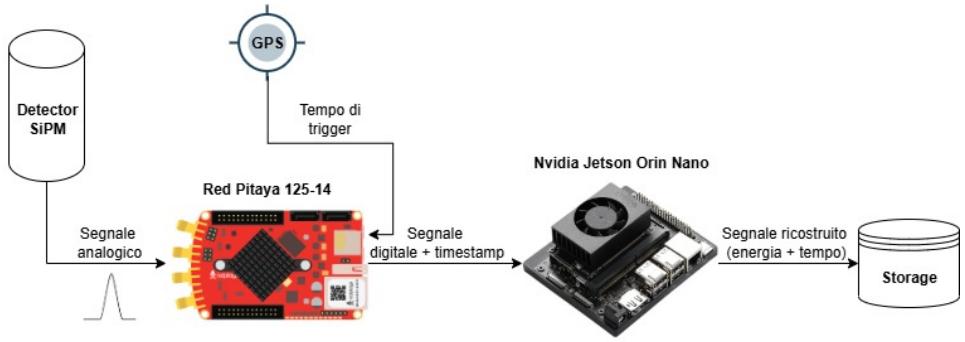


Figura 2.3: Schema funzionale rappresentante il setup sperimentale di GammaSky.

2.3 Modello dei dati

Il modello dei dati adottato in GammaSky organizza le informazioni acquisite e processate lungo la catena in differenti livelli, in base al contenuto informativo che possiedono. Questa struttura multi-livello costituisce l'interfaccia comune tra i vari moduli coinvolti nel sistema. A partire dal dato grezzo generato all'interno del Red Pitaya, l'informazione viene progressivamente arricchita, filtrata e trasformata fino a ottenere parametri fisici, stimati dal modello di ML, direttamente utilizzabili per l'analisi.
I livelli di dato definiti sono i seguenti:

- **R0 (Raw Level 0)**: è il dato grezzo prodotto dal *Data Acquisition Module* (DAM)[9] in esecuzione sulla scheda Red Pitaya. Contiene un header con metadati di acquisizione (run, configurazione, timestamp, stato del GPS) e un buffer di 16 384 campioni ADC, memorizzati come interi unsigned a 16 bit rappresentati in complemento a 2. Il livello R0 costituisce l'input diretto alla pipeline di elaborazione.

- **HK (Housekeeping)**: pacchetto periodico (tipicamente a 0.2 Hz) contenente dei flag che riportano lo stato del sistema di acquisizione. Questi includono l'informa-

[9]Il DAM è il componente software che viene eseguito sulla scheda Red Pitaya e si occupa dell'acquisizione delle waveform provenienti dai rivelatori a scintillazione, della loro serializzazione e dell'invio tramite rete verso la pipeline di analisi.

zione sulla validità del segnale GPS (PPS e stringhe NMEA), contatori di waveform, stato del trigger e parametri operativi. I dati di housekeeping accompagnano i dati scientifici delle waveform come informazione ausiliaria.

- **DL0 (Data Level 0)**: primo livello di archiviazione strutturata, salvato in formato `HDF5`. Contiene le waveform grezze acquisite, ciascuna memorizzata come dataset indipendente nel gruppo `/waveforms`, insieme ai principali metadati di run, configurazione e timestamp. I dati scientifici sono salvati dopo de-complementazione a 2 e rappresentano i valori digitalizzati della tensione in ingresso.

- **DL1 (Data Level 1)**: livello di pre-processing che riduce il volume dei dati scientifici. Vengono conservate solo le forme d'onda rilevanti (ad esempio quelle contenenti un impulso valido), insieme a parametri estratti e a un sottoinsieme dei dati di housekeeping. I file DL1 sono anch'essi organizzati in formato `HDF5` e strutturati per facilitare l'accesso vettoriale ai dati e l'analisi successiva.

- **DL2 (Data Level 2)**: rappresenta il prodotto finale della pipeline di elaborazione scientifica. Ogni evento fisico viene descritto tramite parametri ricostruiti, in particolare il tempo di arrivo e l'energia depositata. I file sono organizzati come tabelle `HDF5`, dove ogni riga corrisponde a un evento e contiene le informazioni fisiche di interesse.

I file DL0, DL1 e DL2 sono strutturati secondo lo standard `HDF5`[10] (*Hierarchical Data Format*), un formato ampiamente utilizzato in ambito scientifico per l'archiviazione di grandi quantità di dati eterogenei. HDF5 consente di organizzare i dati in modo gerarchico tramite *group*, *dataset* e *attribute* e di integrare nello stesso file sia i dati numerici che le informazioni di metadato (strumentazione, configurazione, timestamp, ecc.). Grazie alla sua efficienza e portabilità, il formato HDF5 è adottato in molti campi,

[10] `https://www.hdfgroup.org/solutions/hdf5/`

come l'astrofisica, dove l'organizzazione modulare e la compressione sono fondamentali per gestire volumi elevati di dati acquisiti in continuo.

Di seguito, un approfondimento sui modelli di dato particolarmente rilevanti per questo lavoro.

2.3.1 Formato R0

Il livello `R0` è il primo elemento del modello dati e rappresenta il punto di ingresso del flusso di elaborazione. Ogni pacchetto R0 è prodotto dal modulo DAM in esecuzione sulla Red Pitaya e costituisce l'input diretto per la pipeline RTA-DP, che ne gestisce lo streaming e il processamento. I dati (le waveform grezze digitalizzate dal sistema di acquisizione) sono memorizzati in strutture dati C++, le quali vengono poi serializzate in un buffer binario di un messaggio ZeroMQ (libreria introdotta nella Sezione 3.1.1). Ogni pacchetto R0 è composto da due parti principali:

- **Header**: contiene informazioni sulla sessione di acquisizione, identificativi vari, lo stato del GPS, i timestamp correnti e parametri interni del firmware come la decimazione, l'offset del trigger e la dimensione totale del buffer contenente le waveform. La Tabella 2.1 mostra i campi principali dell'header:

Tabella 2.1: Campi principali dell'header del pacchetto R0.

Campo	Tipo	Descrizione	Esempio
`Type`	uint8	Tipo di pacchetto	0xA1
`SubType`	uint8	Sottotipo di pacchetto	0x01
`SessionID`	uint16	ID della sessione di acquisizione	1
`ConfigID`	uint16	ID della configurazione hardware	3
`TimeSts`	uint8	Stato del time tag GPS	0x00
`PPSSlice`	uint8	Indice PPS usato per il timestamp	0
`Min`	uint8	Minuti	49
`Sec`	uint8	Secondi	32
`Usec`	uint32	Microsecondi (da PPS)	199657
`Equalization`	uint8	Livello di equalizzazione ADC	0
`Decimation`	uint16	Fattore di decimazione ADC	8
`CurrOff`	uint32	Offset del campione finale nel buffer	16384
`TrigOff`	uint32	Offset del trigger nel buffer	8200
`Size`	uint32	Dimensione totale del buffer waveform	16384

- **Payload**: un buffer di 16 384 campioni consecutivi acquisiti dall'oscilloscopio che viene eseguito sull'FPGA del Red Pitaya, memorizzati come interi a 16 bit unsigned complementati a 2, che rappresentano l'intera finestra di acquisizione. Il payload contiene a sua volta un piccolo header il quale consiste principalmente di `Type` e `SubType`, in modo da poter distinguere di che pacchetto si tratta. La waveform contiene il segnale proveniente dal rivelatore a scintillazione, campionato alla frequenza dell'ADC e già pronto per la ricostruzione successiva.

2.3.2 Formato DL0

Il DL0 è un archivio strutturato in formato `HDF5` che contiene l'insieme delle waveform acquisite. Il file presenta un gruppo principale `/waveforms`, all'interno del quale ogni dataset (`wf_000000`, `wf_000001`, ...) rappresenta una singola forma d'onda acquisita. I file seguono la convenzione di naming:

`wf_runId_XXXXX_configId_ZZZZZ_YYYY-MM-DDTHH:mm:ss.SSSSS.h5`

La Figura 2.4 mostra i principali campi dei metadati associati ad una specifica waveform acquisita, derivati dal software di acquisizione di Gamma-Flash[48], da cui GammaSky eredita parte dell'infrastruttura.

Name	Type	Array Size	Value[50](...)
Day	64-bit integer	Scalar	0
Dec	64-bit integer	Scalar	1
Eql	64-bit integer	Scalar	1
HH	64-bit integer	Scalar	0
Month	64-bit integer	Scalar	0
PPSSliceNO	64-bit integer	Scalar	0
SampleNo	64-bit integer	Scalar	1000
TITLE	String, length = 4, padding = H5T_STR_NULLTERM, cset = H5T_CSET_UTF8	Scalar	wf_1
TimeSts	64-bit integer	Scalar	0
TriggerOffset	64-bit integer	Scalar	10191
VERSION	String, length = 3, padding = H5T_STR_NULLTERM, cset = H5T_CSET_UTF8	Scalar	2.0
Year	64-bit integer	Scalar	0
configID	64-bit integer	Scalar	0
dt	64-bit floating-point	Scalar	8.0E-9
mm	64-bit integer	Scalar	0
rp_id	64-bit integer	Scalar	0
runid	64-bit integer	Scalar	0
sessionID	64-bit integer	Scalar	162
ss	64-bit integer	Scalar	0
tend	64-bit floating-point	Scalar	536.000007992
tstart	64-bit integer	Scalar	536

Figura 2.4: Contenuto di un file DL0 riguardante una waveform denominata `wf_000001`. Si possono osservare i vari campi associati alla forma d'onda e i relativi valori.

La gestione del timestamp nei DL0 è articolata in modo da fornire valori precisi e coerenti. Il DAM combina infatti diverse informazioni temporali provenienti sia dall'FPGA sia dal modulo GPS della Red Pitaya. Al momento del trigger, il sistema registra un tempo locale ad alta risoluzione, mentre in parallelo un modulo di sincronizzazione aggiorna periodicamente il tempo assoluto utilizzando il segnale PPS e le stringhe NMEA del GPS. Il timestamp finale viene quindi ricavato a partire dal tempo GPS quando il segnale è valido, oppure dal clock locale in caso di perdita di sincronizzazione. Questo meccanismo consente di associare a ogni waveform un riferimento temporale coerente e, quando il GPS è stabile, accurato all'ordine dei microsecondi.

Ogni DL0, oltre ai metadati, ha associato anche un buffer contenente i valori effettivi della waveform. La struttura di ogni buffer segue un formato chiave–valore, dove la chiave è l'indice progressivo del campione e il valore rappresenta l'ampiezza del segnale

digitalizzato. Una waveform tipica presenta una regione iniziale stabile, generalmente considerata come rumore, seguita da un rapido incremento d'ampiezza (il picco) e da un successivo decadimento, corrispondente al rilascio di energia nel cristallo scintillatore. Viene riportato parte del buffer riguardante la waveform vista in precedenza (`wf_000001`):

```
{
  "0": 125,
  "1": 104,
  "2": 115,
  "3": 133,
  ...
  "22": 1641,
  "23": 1653,
  "24": 1670,
  ...
  "46": 119,
  "47": 133,
  "48": 119,
  "49": 133
}
```

2.3.3 Formato DL2

Il formato `DL2` rappresenta l'output finale della pipeline di elaborazione e quindi del modello dati. Contiene le informazioni fisiche delle waveform ricostruite a partire dal modello R0, come l'energia depositata e il tempo di arrivo di ciascun evento. Il formato utilizzato è `HDF5` e contiene una tabella unica di eventi, dove ogni riga corrisponde a una waveform elaborata (vedi Tabella 2.2 con i campi principali). I file DL2, i quali vengono salvati su disco alla fine del processo di inferenza, seguono la convenzione di naming:

`dl2_runID_XXXXX_seqnum_YYYYY.h5`

Tabella 2.2: Campi principali della tabella di un file DL2.

Campo	**Tipo**	**Descrizione**	**Esempio**
`n_waveform`	float32	Identificativo della forma d'onda analizzata	2.0
`tstart`	float32	Tempo d'inizio dell'evento in secondi UNIX	2.6128e9
`integral1`	float32	Integrale dell'area sottesa al picco della waveform	89435.16

Questo livello fornisce quindi la rappresentazione compatta e fisicamente interpretabile dei dati scientifici prodotti da GammaSky. La pipeline in sostanza, riceve come input dati raw R0 contenenti le waveform grezze, esegue su di esse la ricostruzione tramite rete neurale convoluzionale (CNN) ottimizzata per l'esecuzione su NVIDIA Jetson Orin Nano, e produce in uscita i file DL2.

Capitolo 3

Real-Time Analysis DataProcessor (RTA-DP)

Il *Real-Time Analysis DataProcessor* (RTA-DP)[49] è un framework per il processamento distribuito di dati in tempo reale, sviluppato presso l'INAF – OAS, con l'obiettivo di fornire un'infrastruttura software modulare, scalabile e adattabile a diversi scenari sperimentali. Il sistema consente di costruire pipeline di analisi basate su flussi di dati, composte da processi indipendenti che comunicano tra loro attraverso canali ad alte prestazioni. In tale contesto, RTA-DP è stato progettato per supportare l'implementazione di pipeline di analisi di dati in tempo reale nel campo delle osservazioni di raggi gamma, per strumenti basati a terra, garantendo al contempo elevato throughput, bassa latenza e flessibilità architetturale. Il codice sorgente del framework e le istruzioni per l'esecuzione sono disponibili pubblicamente nel repository GitHub ufficiale: `https://github.com/ASTRO-EDU/rta-DataProcessor`.

Grazie alla sua struttura modulare, RTA-DP consente di configurare pipeline adatte a diversi contesti sperimentali, assicurando parallelismo, scalabilità e controllo distribuito dei processi. La versione iniziale del framework, implementata in Python, è utilizzata come pipeline software per alcuni progetti in cui OAS è coinvolto, tra cui l'*Online*

Observation Quality System (OOQS) dell'*ASTRI Mini-Array Project*[50] e il già citato GammaSky. Riguardo quest'ultimo, è attualmente eseguito su un server dedicato, in attesa dell'ottimizzazione necessaria per il deployment su piattaforme di edge computing a basso consumo di risorse. Tale architettura ha costituito la base per le successive evoluzioni del framework, tra cui la versione ottimizzata in C++ impiegata proprio in GammaSky.

3.1 Architettura generale del sistema

L'architettura del framework è modulare e organizzata attorno a un insieme di entità funzionali con responsabilità distinte e complementari. Il *Producer* costituisce la sorgente dei dati in ingresso, i quali vengono ricevuti dal *DataProcessor*, ossia il nodo di calcolo responsabile dell'elaborazione, che incapsula al suo interno i componenti operativi. Il *Supervisor* funge da orchestratore locale: coordina l'avvio e il ciclo di vita delle unità di lavoro, gestisce la configurazione runtime e raccoglie ed esporta informazioni di stato; i compiti di esecuzione parallela sono demandati a insiemi di *WorkerManager*, i quali organizzano pool omogenei di *Worker* incaricati delle singole operazioni di processamento (deserializzazione, processing, analisi e scrittura dei risultati). Il controllo della pipeline è affidato a un componente di coordinamento denominato *Coordinator System*, il quale è in grado di comandare e gestire più DataProcessor simultaneamente. Infine, il sottosistema di *Monitoring* centralizza e pubblica statistiche sul processamento e la telemetria dei nodi del sistema, utili per la supervisione e per il debugging. La struttura del sistema è completamente configurabile tramite file JSON, che definiscono la topologia dei processi, i canali di comunicazione, le code a priorità, il numero di worker e le modalità di logging e monitoraggio.

Oltre alla sua architettura modulare, RTA-DP è concepito per essere facilmente riutilizzabile e adattabile a diversi casi d'uso. Ogni applicazione può infatti specializzare i

suoi componenti principali per definire il tipo di processamento richiesto. In particolare, i worker rappresentano le unità elementari di elaborazione e possono essere estesi per implementare task specifici, come operazioni di pre–processing, filtraggio o analisi mediante modelli avanzati come quelli di machine learning. I WorkerManager coordinano gruppi di worker specializzati, mentre il supervisor può essere personalizzato per controllare diverse combinazioni di WorkerManager e adattare il pre–processing dei dati in funzione del caso d'uso. In questo modo, il framework fornisce un'infrastruttura generale per la gestione della comunicazione, del parallelismo e del monitoraggio, lasciando però libertà di scelta su come implementare la logica di elaborazione.

3.1.1 Librerie utilizzate

RTA-DP si concentra su un insieme limitato di librerie esterne Python, scelte per supportare la comunicazione, la serializzazione e la gestione dei flussi di dati in tempo reale. La combinazione di queste librerie consente di gestire architetture distribuite, flessibili e affidabili per l'elaborazione in tempo reale di grandi volumi di dati. Quelle utilizzate sono:

- **Apache Avro**[1]: framework di serializzazione dei dati sviluppato nell'ambito del progetto Apache Hadoop. Fornisce un formato binario compatto ed efficiente per la trasmissione di grandi quantità di dati, insieme a un'interfaccia semplice per la definizione delle strutture tramite schemi JSON. Ogni schema descrive la struttura e i tipi di dato previsti, garantendo compatibilità e coerenza tra sistemi eterogenei e linguaggi diversi. Avro supporta inoltre un'ampia gamma di tipi complessi (record, array, mappe, unioni) e permette la *schema evolution*, la quale permette di aggiornare o estendere le definizioni dei dati senza compromettere la compatibilità con versioni precedenti.

[1] `https://avro.apache.org/`

- **ZeroMQ (Zero Message Queue)**[2]: libreria di messaggistica ad alte prestazioni progettata per la realizzazione di applicazioni distribuite e concorrenti. Fornisce un'infrastruttura di comunicazione leggera e scalabile, semplificando lo sviluppo di sistemi complessi grazie a un'API di alto livello per la gestione dei socket. ZeroMQ supporta diversi pattern di messaggistica, ciascuno adatto a differenti modelli di comunicazione:

  - **PUSH/PULL**: modello di comunicazione unidirezionale in cui uno o più produttori (`PUSH`) inviano messaggi a uno o più consumatori (`PULL`). È utilizzato per distribuire il carico di lavoro in modo bilanciato tra più processi o thread (ad esempio, i worker della pipeline).
  - **PUB/SUB (Publisher/Subscriber)**: modello basato sulla pubblicazione e sottoscrizione, in cui un `publisher` invia messaggi a uno o più `subscriber`. Questi ultimi ricevono soltanto i messaggi relativi agli argomenti o canali a cui sono iscritti. È impiegato per la trasmissione in broadcast di eventi o dati di interesse comune.
  - **REQ/REP (Request/Reply)**: pattern di tipo client-server che implementa uno scambio sincrono di messaggi, dove un client (`request`) invia una richiesta e il server (`reply`) risponde. È utile nei casi in cui sia necessario un feedback esplicito o un controllo puntuale delle comunicazioni.

  ZeroMQ è agnostica rispetto al linguaggio di programmazione e al trasporto, funzionando su protocolli come TCP, UDP e IPC, e garantisce bassa latenza e throughput elevato. In RTA-DP, è impiegata come strato di comunicazione tra i processi della pipeline, facilitando il trasferimento rapido di dati e messaggi di controllo tra componenti distribuiti. Il pattern di comunicazione utilizzato è specificato nel file di

[2] `https://zeromq.org/`

configurazione JSON del framework. ZeroMQ è spesso impiegata in combinazione con Avro, che gestisce la serializzazione dei dati trasmessi nei messaggi.

- **Apache Kafka**[3]: piattaforma di streaming distribuita sviluppata da Apache Software Foundation, progettata per gestire flussi di dati in tempo reale su larga scala con elevata tolleranza ai guasti. Kafka funge da message broker distribuito, basato su un modello `publish/subscribe`, in cui i produttori pubblicano messaggi su specifici *topic* e i consumatori si iscrivono per riceverli. Il sistema garantisce persistenza e replicazione dei dati, permettendo la gestione di elevati volumi di traffico con scalabilità orizzontale e bassa latenza. In alcune configurazioni di RTA-DP, Kafka viene utilizzato come alternativa o complemento a ZeroMQ per gestire pipeline distribuite su più nodi o esperimenti che richiedono buffering, tracciabilità e resilienza ai guasti. In combinazione con il formato Avro, consente la serializzazione efficiente dei messaggi e la gestione centralizzata degli schemi tramite *Schema Registry*.

- **JSON**: JSON (JavaScript Object Notation)[4] è un formato di interscambio dati leggero, di tipo testuale e facilmente leggibile sia da esseri umani sia da macchine. È "schema-less", quindi non richiede una struttura predefinita, e supporta tipi di dati semplici come stringhe, numeri, array e oggetti. Grazie alla sua semplicità, è ampiamente utilizzato per file di configurazione, interfacce API e scambio di messaggi tra componenti software. In RTA-DP, JSON viene impiegato per la definizione dei parametri di configurazione, per i messaggi di controllo e per il monitoraggio del sistema. I log generati permettono di tracciare in tempo reale lo stato dei processi e delle code, supportando il debugging e l'analisi delle prestazioni. Rispetto ad Avro, JSON privilegia la leggibilità e la facilità di utilizzo, mentre Avro viene adottato per la serializzazione efficiente e compatta dei dati binari.

[3] `https://kafka.apache.org/`
[4] `https://www.json.org/json-en.html`

3.1.2 Componenti principali

Il framework è costituito da un `DataProcessor`, il quale a sua volta è composto da diversi componenti principali che cooperano per realizzare le pipeline di analisi in tempo reale. Ogni DataProcessor esegue il processamento di uno o più tipi di elaborazione sullo stesso tipo di dato e dispone di un unico *Supervisor* e di un insieme di *WorkerManager*. Ogni Supervisor gestisce `n` WorkerManager, e ciascun WorkerManager coordina `M` *Worker* in parallelo (vedi Figura 3.2). Tutti e tre i componenti del DataProcessor implementano una macchina a stati che regola le fasi operative (inizializzazione, gestione dei cicli di attesa, processamento e chiusura), la quale consente un monitoraggio coerente del sistema durante l'esecuzione (in Figura 3.1 è riportata la macchina a stati del WorkerManager).

- **Supervisor**: è il componente centrale del sistema e gestisce l'orchestrazione dei dati e degli altri componenti. Il Supervisor è responsabile dell'inizializzazione dell'intera pipeline: istanzia i WorkerManager previsti e richiama ognuno di essi per creare i rispettivi worker, secondo le specifiche definite nel file JSON di configurazione. Può operare secondo due modelli di esecuzione, specificati nel file di configurazione: `Thread` (esecuzione concorrente nello stesso spazio di memoria) oppure `Process` (esecuzione isolata tra istanze).

  A runtime, riceve i flussi di dati dai producer attraverso socket ZeroMQ, secondo il pattern di comunicazione indicato nel file di configurazione. Questi pattern definiscono la modalità di interfacciamento del DataProcessor con l'esterno, ovvero con altri moduli o processi che utilizzano ZeroMQ per la trasmissione dei dati. In particolare:

  – il pattern **PUSH/PULL** implementa una comunicazione unidirezionale di tipo *many-to-one* o *load-balanced*, in cui uno o più produttori inviano messaggi a un singolo consumatore; è tipicamente utilizzato per distribuire il carico di eventi tra più nodi di processamento;

– il pattern **PUB/SUB** realizza una comunicazione *one-to-many*, in cui un *publisher* invia i dati a più *subscriber* interessati, utile in scenari in cui diversi moduli devono ricevere simultaneamente lo stesso flusso di dati.

Questi pattern riguardano esclusivamente la comunicazione esterna del DataProcessor. La comunicazione interna tra i tre componenti avviene invece tramite code concorrenti e meccanismi di scambio di comandi dedicati. In questo modo, il Supervisor riceve i dati dai socket ZeroMQ, li inserisce nelle code interne a priorità differente (low priority – LP e high priority – HP) per la gestione separata di eventi ordinari e critici, e i WorkerManager li prelevano per distribuirli ai rispettivi worker per l'elaborazione. I risultati prodotti vengono infine inviati verso l'esterno attraverso gli endpoint ZeroMQ configurati per la trasmissione, secondo il tipo di interfaccia e di dato specificato nel file JSON di configurazione.

Il Supervisor inoltre, pubblica periodicamente messaggi di monitoraggio e, in base ai comandi di controllo che riceve dal centro di comando, esegue funzionalità differenti:

– `shutdown`: interrompe immediatamente il funzionamento del sistema e termina tutti i processi attivi;

– `cleanedshutdown`: esegue una chiusura controllata, attendendo che tutte le code non vuote vengano liberate prima di arrestare il sistema;

– `startprocessing`: avvia l'elaborazione dei dati impostando lo stato del sistema su "Processing" e abilitando l'elaborazione nei WorkerManager;

– `stopprocessing`: sospende l'elaborazione dei dati mantenendo il sistema attivo, riportando lo stato a "Waiting";

– `startdata`: abilita la ricezione dei dati dai producer, impostando il flag `stopdata = false`;

  - `stopdata`: interrompe temporaneamente la ricezione dei dati dai producer, impostando il flag `stopdata = true`;
  - `start`: avvia simultaneamente la ricezione e l'elaborazione dei dati;
  - `stop`: ferma sia la ricezione che l'elaborazione dei dati;
  - `reset`: riporta il sistema allo stato iniziale (come fa `stop`), pulisce tutte le code dei WorkerManager e riporta lo stato a "Waiting";
  - `getstatus`: restituisce informazioni sullo stato corrente del Supervisor e dei WorkerManager.

  In sostanza quindi, il Supervisor controlla l'esecuzione della pipeline e il lifetime della stessa, in base ai messaggi di comando che riceve. Prima dell'inoltro dei dati, esegue una fase di pre-processing (la quale può essere modificata se necessario) specifica al tipo con il quale il sistema avrà a che fare, il quale è specificato nel file JSON di configurazione:

  - per dati di tipo `filename`, effettua l'apertura del file contenente la lista di eventi, i quali vengono caricati su una coda;
  - per dati di tipo `binary`, esegue una decodifica del payload e carica il risultato sulla coda;
  - per dati di tipo `string`, inoltra direttamente il messaggio senza ulteriori elaborazioni.

- **WorkerManager**: ogni WorkerManager gestisce un gruppo di worker che eseguono task di elaborazione in parallelo. Gestisce i dati attraverso code concorrenti (riempite man mano dal Supervisor), li distribuisce ai worker attraverso code interne e raccoglie i risultati nelle proprie code di output. Ogni WorkerManager mantiene una coda per i risultati associata e invia periodicamente al Supervisor

metriche di performance per il monitoraggio (vedi un esempio di messaggio di monitoraggio nella Sezione 3.1.4). Questa struttura a pool consente di parallelizzare il lavoro e bilanciare il carico tra le unità di elaborazione, aumentando l'efficienza complessiva della pipeline.

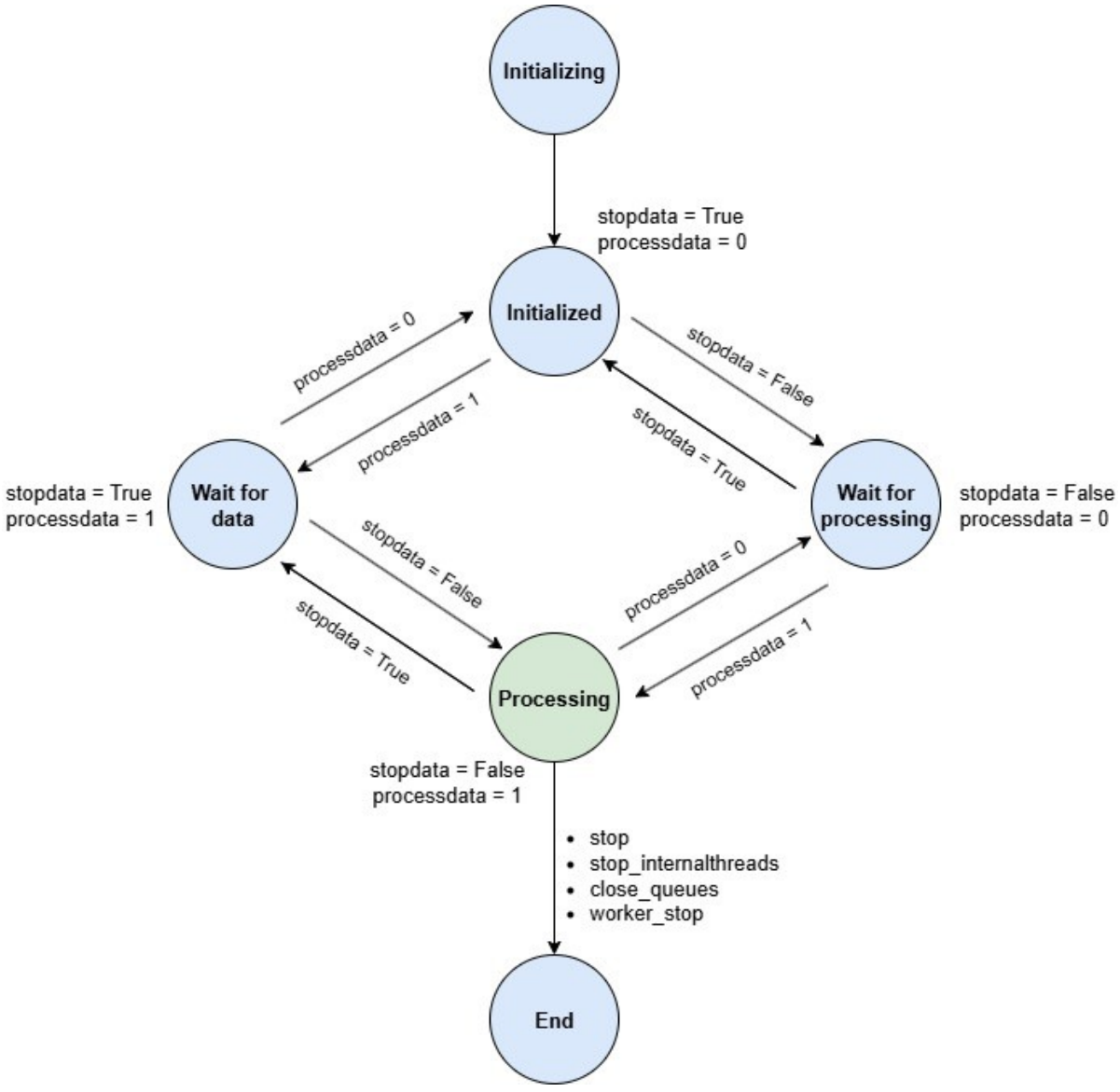


Figura 3.1: Macchina a stati del WorkerManager, con i principali stati e transizioni che regolano il ciclo di elaborazione dei dati all'interno del DataProcessor.

- **Worker**: i worker rappresentano le unità di calcolo effettive del sistema, dove avviene il vero e proprio processamento dei dati. Ogni worker eredita da una classe base astratta che ne definisce l'interfaccia e implementa un metodo dedicato per l'elaborazione dei dati in ingresso. Tutti i worker gestiti dallo stesso WorkerManager sono istanze equivalenti e applicano lo stesso algoritmo di elaborazione in parallelo sul flusso di dati assegnato.

  A seconda della configurazione, i worker possono operare su file, pacchetti binari o stringhe, e inseriscono i risultati nelle code di output LP o HP in base alla priorità dell'evento. Essi costituiscono il punto in cui si concretizza il caso d'uso specifico del DataProcessor: qui vengono implementate le logiche di calcolo e analisi che possono spaziare da semplici operazioni di pre-processing fino a modelli complessi di analisi avanzata o inferenza basata su machine learning. Grazie a questa architettura modulare, è possibile adattare facilmente il comportamento dei worker a diversi contesti applicativi (ad esempio, per la decodifica dei dati grezzi, l'estrazione di feature o l'analisi ad alto livello).

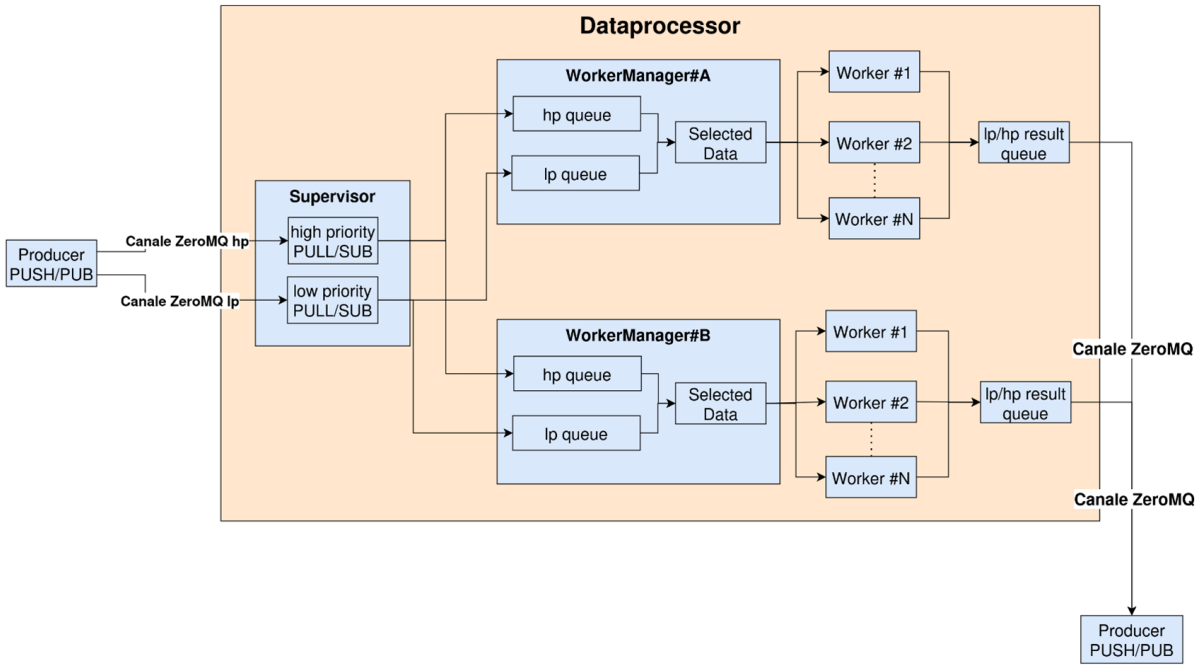


Figura 3.2: Schema generale del DataProcessor che illustra il flusso di dati e messaggi all'interno del framework RTA-DP: i dati prodotti dai Producer vengono ricevuti dal Supervisor (Consumer), instradati ai WorkerManager e successivamente elaborati in parallelo dai Worker, fino alla produzione dei risultati sui canali di output (Crediti: INAF – OAS).

3.1.3 Configurazione

L'architettura di RTA-DP è completamente configurabile tramite un file in formato JSON, che definisce i parametri operativi del sistema e la topologia della pipeline. Il file di configurazione rappresenta un elemento chiave del framework: consente infatti di descrivere in modo modulare la struttura della pipeline, con la dichiarazione della configurazione di ogni DataProcessor e dei componenti di controllo e monitoraggio. Per ogni DataProcessor, viene specificato il tipo di dato da elaborare, le modalità di comunicazione (pattern ZeroMQ) con il mondo esterno (quindi con il producer e con chi "consumerà" l'output), il numero di worker da istanziare e i socket di rete utilizzati per la trasmissione dei dati e dei messaggi di monitoraggio.

Viene riportato di seguito un esempio di file di configurazione per uno specifico DataProcessor:

```
{
  "processname": "RTADP1",
  "dataflow_type": "binary",
  "processing_type": "thread",
  "datasocket_type": "pubsub",
  "data_lp_socket": "tcp://127.0.0.1:5564",
  "data_hp_socket": "tcp://127.0.0.1:5565",
  "command_socket": "tcp://127.0.0.1:5568",
  "monitoring_socket": "tcp://127.0.0.1:5561",
  "manager": [
    {
      "result_socket_type": "pubsub",
      "result_dataflow_type": "binary",
      "result_lp_socket": "tcp://127.0.0.1:5566",
      "result_hp_socket": "tcp://127.0.0.1:5567",
      "num_workers": 5,
      "name": "Rate",
      "name_workers": "worker"
    }
  ],
  "logs_path": "/tmp/",
  "logs_level": 5
}
```

Il contenuto del file JSON può essere suddiviso in tre sezioni principali:

- **Parametri generali del DataProcessor**
  - `processname`: nome identificativo del DataProcessor.
  - `dataflow_type`: indica il tipo di dato da gestire. Può assumere tre valori:
    - `filename`: il dato è rappresentato da un percorso a un file contenente la lista di eventi da elaborare;
    - `binary`: il dato è una sequenza di byte da decodificare;
    - `string`: il dato è una stringa testuale, che può contenere valori numerici, array o messaggi JSON.
  - `processing_type`: definisce la modalità di esecuzione del DataProcessor. Può essere `thread` oppure `process`.
  - `datasocket_type`: specifica il pattern ZeroMQ di comunicazione implementato dai canali, tipicamente `pubsub` o `pushpull`, a seconda dell'architettura della pipeline.
  - `logs_path` e `logs_level`: specificano rispettivamente il percorso in cui salvare i file di log e il livello di dettaglio dei messaggi registrati. Ogni Supervisor genera file di log contenenti informazioni sull'elaborazione dei dati, eccezioni ed errori. Il parametro `logs_level` può assumere valori da 1 a 5, dove 1 indica solo errori critici e 5 il massimo livello di verbosità.
- **Socket e canali di comunicazione**
  - `data_lp_socket` e `data_hp_socket`: definiscono gli endpoint TCP utilizzati per i canali di comunicazione rispettivamente a bassa (LP) e alta priorità (HP). Questa distinzione consente di separare flussi di dati ordinari da quelli critici o urgenti.
  - `command_socket`: punto di connessione per la ricezione dei comandi operativi (`start`, `stop`, `shutdown`, ecc.).

  - `monitoring_socket`: canale dedicato alla trasmissione dei dati di monitoraggio (stato dei processi, utilizzo risorse, eventi processati). In configurazioni distribuite, il sistema può impiegare uno o più *forwarder* per aggregare e inoltrare i monitoring point verso un nodo centrale.

- **Configurazione dei WorkerManager**
  - Ogni elemento dell'array `manager` definisce un WorkerManager con i relativi parametri di output e parallelizzazione:
    * `result_socket_type`: tipo di comunicazione adottata per l'invio dei risultati (es. `pubsub`);
    * `result_dataflow_type`: formato dei dati di output (es. `string` o `binary`);
    * `result_lp_socket` e `result_hp_socket`: socket TCP dedicati ai canali di output a bassa e alta priorità;
    * `num_workers`: numero di worker che il WorkerManager deve istanziare;
    * `name` e `name_workers`: identificatori testuali utilizzati per etichettare i processi e i thread.

3.1.4 Comunicazione e formati

Tutti i messaggi scambiati tra i componenti del framework RTA-DP condividono una struttura comune in formato JSON. Ogni messaggio è composto da due sezioni principali:

- un **header**, che contiene i campi standard di identificazione e instradamento del messaggio;
- un **body**, che racchiude il contenuto vero e proprio (payload) o i parametri operativi.

L'**header** include le seguenti informazioni:

- `type`: tipo di messaggio (0 = *command*, 1 = *monitoring*, 2 = *alarm*, 3 = *configuration*, 4 = *log*, 5 = *info*);
- `subtype`: categoria o sottotipo specifico del messaggio;
- `time`: timestamp del momento di generazione del messaggio;
- `pidsource`: identificativo del processo sorgente;
- `pidtarget`: identificativo del processo destinatario (può essere "*" per broadcast);
- `priority`: livello di priorità del messaggio (es. *low* o *high*).

La sezione **body**, invece, varia in base alla tipologia di messaggio e può contenere dati di monitoraggio, log, eventi o pacchetti di dati scientifici. Le principali tipologie di messaggi gestite dal sistema sono le seguenti:

- **Command**: messaggi di controllo e gestione del sistema. Permettono di inviare comandi come `start`, `stop`, `shutdown` o `reset` ai vari processi del framework per l'avvio oppure l'arresto dell'acquisizione dei dati e dell'elaborazione. Alcuni comandi possono agire sull'intera pipeline (ad esempio `pidtarget = "*"`), mentre altri sono diretti a componenti specifici. Un esempio di comando usato per comunicare a tutti i processi attivi di fermare il processamento degli eventi potrebbe essere:

```
{
    "header": {
      "type": 0,
      "subtype": "stopprocessing",
      "time": "2024-01-12T12:34:56Z",
      "pidsource": "processA",
      "pidtarget": "*",
```

```
    "priority": "high"
  }
}
```

- **Monitoring**: messaggi periodici usati per il monitoraggio in tempo reale delle prestazioni e dello stato dei componenti attivi nel sistema. Contengono informazioni come lo stato del processo (es. *Processing*, *Idle*, *Stopped*), l'utilizzo di CPU e memoria, la dimensione delle code di input e output e varie metriche riguardanti i worker (come il numero di eventi processati). Un esempio di messaggio di monitoraggio prodotto da un WorkerManager può essere:

```
{
    "header": {
        "type": 1,
        "time": 1761303743.5273652,
        "pidsource": "R0toDL2-r0dl2_wm",
        "pidtarget": "*"
    },
    "workermanagerstatus": "Processing",
    "procinfo": {
        "cpu_percent": 99.9,
        "memory_usage": [
            186490880,
            20438589440,
            0,
            19686834176,
            0
```

```
            ]
        },
        "queue_lp_size": 0,
        "queue_hp_size": 0,
        "stopdatainput": false,
        "queue_lp_result_size": 0,
        "queue_hp_result_size": 0,
        "workersstatusinit": 0,
        "workersstatus": 2.0,
        "workersname": "worker",
        "worker_rates": {"0": 0.0},
        "worker_tot_events": {"0": 380.0},
        "worker_status": {"0": 2}
    }
```


- **Alarm, Log e Info**: messaggi informativi o diagnostici utilizzati per notificare eventi anomali, condizioni operative o semplici messaggi di log. Ogni Supervisor, WorkerManager e Worker può generare monitoring points, alarms, logs e informations.

- **Dati di input e output**: pacchetti di dati scientifici da elaborare. Come già detto possono essere di tre tipi principali: **filename** che contiene il nome di un file di dati da elaborare, **binary** che rappresenta dati binari arbitrari (ad esempio waveform digitalizzate) e **string** il quale contiene una stringa o un flusso di testo strutturato (ad esempio messaggi JSON o Avro).

Capitolo 4

Reingegnerizzazione e ottimizzazione di RTA-DP in C++

In questo capitolo viene descritto il lavoro svolto durante la prima parte del tirocinio presso INAF – OAS, con l'obiettivo di rendere funzionante e performante il prototipo C++ del framework RTA-DP. La necessità di migrare e ottimizzare rispetto alla versione in Python è dettata dai requisiti di latenza e consumo tipici di ambienti di edge computing e dall'esigenza di integrare nativamente librerie C++ per la comunicazione e l'ottimizzazione. Il lavoro ha preso come punto di partenza una bozza C++ non funzionante e l'ha trasformata in un framework stabile, manutenibile e configurabile.

4.1 Limitazioni dell'implementazione originale in Python

La versione originale del framework RTA-DP, sviluppata in Python, costituiva un valido prototipo funzionale e ha permesso di definire la logica generale della pipeline e dei flussi di comunicazione. Tuttavia, durante le prime fasi di test sono emerse diverse limitazioni che ne impedivano l'utilizzo in contesti con una limitazione di risorse. Python, pur

essendo un linguaggio estremamente efficace per la prototipazione, lo sviluppo rapido e l'analisi dati, introduce inevitabilmente un overhead dovuto alla natura interpretata e al modello di concorrenza, che ne limita l'uso in contesti ad alte prestazioni o con vincoli hardware rigidi.

Per rispondere a queste esigenze, si è reso necessario un porting completo in C++, linguaggio adatto a tali piattaforme visto che consente un controllo più diretto sulla gestione della memoria, sul parallelismo e sulle ottimizzazioni a basso livello. Questo approccio ha reso possibile ottenere prestazioni compatibili con i requisiti dei sistemi embedded e delle architetture di calcolo real-time. Di seguito sono riportate le principali limitazioni riscontrate nella versione originale e le motivazioni tecniche che hanno guidato la reingegnerizzazione in C++:

- **Prestazioni e latenza**: l'esecuzione in ambiente interpretato e la presenza del Global Interpreter Lock (GIL)[1] introducono inevitabilmente un overhead che limita la prevedibilità dei tempi di esecuzione. In un contesto real–time, dove la latenza deve essere costante e controllata, queste variabilità compromettono il comportamento del sistema. Il porting in C++ consente di ottenere codice compilato nativamente e di sfruttare le ottimizzazioni del compilatore, migliorando sensibilmente il throughput.

- **Controllo della memoria e uso delle risorse**: Python delega la gestione della memoria al garbage collector, rendendo meno prevedibili i tempi di allocazione e rilascio. Su dispositivi con risorse limitate è invece essenziale avere un controllo esplicito sul consumo di memoria e sulle allocazioni. C++ permette di gestire in modo diretto il ciclo di vita delle risorse, adottando strategie di allocazione personalizzate per contenere il memory footprint complessivo.

[1] Il *Global Interpreter Lock* (GIL) è un meccanismo dell'interprete Python che permette l'esecuzione di un solo thread alla volta per proteggere le strutture interne del linguaggio. Sebbene semplifichi la gestione della memoria, limita l'effettivo parallelismo dei thread "CPU-bound".

- **Multithreading e parallelismo**: il GIL di Python limita l'efficacia del multithreading nei task CPU-bound, impedendo l'esecuzione concorrente di più thread all'interno dello stesso processo. Sebbene il modulo `multiprocessing` consenta di aggirare parzialmente questo vincolo creando processi separati, tale approccio introduce un overhead aggiuntivo in termini di memoria e comunicazione inter-processo, risultando meno efficiente in sistemi con risorse limitate. In C++, il multithreading è nativo e supportato da primitive efficienti (`std::thread`, `std::mutex`, `std::condition_variable`), che permettono di sfruttare pienamente i core disponibili condividendo lo stesso spazio di memoria e riducendo il costo di sincronizzazione, aspetto fondamentale per massimizzare le prestazioni in ambienti edge e real-time.

- **Integrazione con librerie e acceleratori**: la scelta del C++ è stata guidata anche dalla prospettiva di un futuro caso d'uso del framework RTA-DP, ovvero l'analisi di eventi gamma mediante inferenza in tempo reale basata su modelli di machine learning (vedi Capitolo 5). Poiché il dispositivo edge scelto per l'esecuzione del framework è una NVIDIA Jetson, è stato ritenuto opportuno utilizzare un linguaggio pienamente compatibile con le librerie di ottimizzazione e inferenza NVIDIA (come LiteRT e CUDA). Tali librerie forniscono interfacce native in C/C++ altamente performanti, non direttamente accessibili o pienamente efficienti in Python. L'integrazione diretta in C++ consente inoltre di gestire buffer binari senza copie intermedie e di ridurre la latenza nella comunicazione con i dispositivi di accelerazione hardware.

- **Librerie di supporto**: la logica di comunicazione basata su ZeroMQ era già presente nella versione Python, ma la riscrittura in C++ ha permesso di gestire in modo più efficiente le risorse e di integrare librerie pensate per ambienti ad alte prestazioni. In particolare, l'introduzione di `spdlog`[2] ha consentito di implementare

[2] *spdlog* è una libreria C++ open source per il logging ad alte prestazioni. Offre un'interfaccia semplice

un sistema di logging configurabile a runtime, con livelli di dettaglio selezionabili e la possibilità di disattivare completamente l'output durante le fasi di elaborazione. Questo approccio riduce l'impatto del logging sulle prestazioni e contribuisce a una maggiore stabilità complessiva del sistema.

4.2 Stabilizzazione e refactoring iniziale

La prima fase del lavoro ha avuto come obiettivo la messa in funzione del prototipo C++ di RTA-DP, che nella versione di partenza non risultava eseguibile. Il codice presentava numerose criticità strutturali e logiche che impedivano la corretta inizializzazione dei componenti, la gestione dei thread e la sincronizzazione tra i moduli. L'attività è quindi iniziata con un'analisi sistematica dei crash e delle eccezioni rilevati in fase di esecuzione, utilizzando strumenti di debugging come `gdb`[3] e un sistema di logging temporaneo, al fine di individuare le sezioni di codice affette da comportamenti indefiniti o accessi a memoria non validi. Una volta individuate le principali cause di instabilità, è stato eseguito un processo di refactoring mirato a garantire il corretto funzionamento del framework e del ciclo di vita dei componenti.

4.2.1 Correzione delle criticità strutturali

Tra i problemi iniziali più rilevanti si sono verificati diversi casi di `segmentation fault` durante l'avvio del framework, causati da dereferenziazioni di puntatori nulli o dall'utilizzo di oggetti non ancora istanziati. Un esempio tipico riguardava la variabile `manager` nella funzione `Supervisor::listen_for_result()`, utilizzata prima della corretta inizializzazione e responsabile dell'arresto immediato del programma. Per evitare tale com-

e thread-safe, supporta sink multipli (console, file, syslog) e consente di configurare il livello di verbosità a runtime, minimizzando l'overhead anche in applicazioni real-time.

[3] `gdb` (GNU Debugger) è un debugger open source per programmi C, C++ e altri linguaggi, che consente di ispezionare lo stato di esecuzione, impostare breakpoint e analizzare le cause di crash o comportamenti anomali.

portamento è stato introdotto un controllo esplicito sullo stato del puntatore e un ciclo di attesa con ritardo temporizzato, che assicura la disponibilità del componente prima del suo utilizzo:

```
for (auto& manager : manager_workers) {
    int attempt = 0;
    while (manager == nullptr && attempt < 10) {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        attempt++;
    }
}
```

L'origine di questi errori era però più profonda e riconducibile alla mancanza di un ordine deterministico nell'inizializzazione dei componenti. Durante l'avvio del sistema, il framework non garantiva che i `WorkerManager` fossero creati prima dei relativi worker e dei thread di servizio, con conseguenti accessi a strutture non ancora pronte. Per risolvere definitivamente la criticità, è stata definita la sequenza esplicita di avvio:

`start_managers()` → `start_workers()` → `start_service_threads()`.

Questa modifica ha eliminato i crash all'avvio e ha reso prevedibile la sequenza di inizializzazione del sistema, garantendo che ogni componente venga istanziato e registrato prima dell'avvio delle unità dipendenti.

4.2.2 Gestione del multithreading, sincronizzazione e stabilità

Una parte significativa del lavoro è stata dedicata alla correzione dei problemi legati al multithreading e alla gestione del ciclo di vita dei thread e dei componenti, che rappresentavano una delle principali cause di instabilità del framework. Durante le prime esecuzioni, l'invio del comando di terminazione globale (`shutdown all`) o l'interruzione tramite `CTRL+C` (SIGINT) causavano frequentemente `segmentation fault`, eccezioni del tipo "*invalid argument*" e, in alcuni casi, errori come "*cannot join thread before it is started*". Questi comportamenti indicavano che la terminazione dei thread non era

gestita in modo ordinato e che la sincronizzazione tra i componenti del sistema era incompleta: diversi thread rimanevano bloccati in attesa su risorse condivise o su socket ZeroMQ, impedendo la chiusura pulita del framework.

Per risolvere tali problemi, è stata riscritta la logica di gestione dei thread all'interno delle classi `Supervisor`, `WorkerManager` e `WorkerThread`. La variabile di controllo globale `continueall`, in precedenza dichiarata come semplice `bool`, è stata sostituita da un tipo atomico per garantire la corretta propagazione dei segnali di stop a tutti i thread concorrenti:

```
std::atomic<bool> continueall;
```

Nel metodo `stop_all()` del `Supervisor`, la variabile viene impostata a `false` non appena ricevuto il comando di terminazione, permettendo a tutti i thread di completare in modo controllato le operazioni in corso:

```
void Supervisor::stop_all() {
    continueall = false;
    std::cout << "Stopping all workers and managers..." << std::endl;
    command_stop();
    std::this_thread::sleep_for(std::chrono::milliseconds(100));

    for (auto &manager : manager_workers) {
        manager->stop();
    }
    std::cout << "All workers, managers and internal threads terminated.";
}
```

Nel distruttore della classe `WorkerThread` è stato aggiunto un `mutex` per rendere sicura la distruzione dell'oggetto `worker` anche in presenza di più thread concorrenti, evitando accessi simultanei a memoria non più valida:

```
WorkerThread::~WorkerThread() {
    {
        std::lock_guard<std::mutex> lock(stop_worker_mutex);
        if (worker) {
            delete worker;
            worker = nullptr;
        }
```

```
    }
    if (!_stop_event) {
        stop();
    }
}
```

Per impostazione predefinita, la chiamata `recv()` di ZeroMQ è bloccante, ovvero il thread rimane in attesa finché non viene ricevuto un nuovo messaggio. Questo comportamento, se non gestito, può impedire la corretta terminazione del programma, poiché i thread di ricezione restano sospesi durante la chiusura del sistema. Per evitare che i socket ZeroMQ rimanessero bloccati in attesa di nuovi messaggi durante la chiusura, sono stati introdotti timeout di un secondo nelle chiamate `recv()` su `socket_lp_data` e `socket_hp_data`. In questo modo, il loop di ricezione può controllare periodicamente il flag di stop ed uscire correttamente anche in presenza di segnali di terminazione (incluso `SIGINT`). La gestione del segnale di interruzione è stata migliorata verificando il valore di ritorno di `recv()` nelle funzioni `listen_for_lp_data()` e `listen_for_hp_data()`, così da interrompere il ciclo principale in caso di errore (`EINTR`) oppure per la ricezione del comando `CTRL+Z`:

```
int timeout = 1000;  // Timeout in millisecondi

if (datasockettype == "pubsub") {
    socket_lp_data = new zmq::socket_t(context, ZMQ_SUB);
    socket_lp_data->connect(config["data_lp_socket"].get<std::string>());
    socket_lp_data->setsockopt(ZMQ_SUBSCRIBE, "", 0);
    socket_lp_data->setsockopt(ZMQ_RCVTIMEO, &timeout, sizeof(timeout));
}

void Supervisor::listen_for_lp_data() {
    while (continueall) {
        if (!stopdata) {
            zmq::message_t data;
            zmq::recv_flags flags = zmq::recv_flags::none;
            try {
                auto result = socket_lp_data->recv(data, flags);
            }
            catch (const zmq::error_t& e) {
```

```
                if (zmq_errno() == EINTR) {
                    break;  // Interruzione del ciclo su segnale
                } else {
                    std::cerr << "ZMQ exception: " << e.what() << std::endl;
                    throw;
                }
            }
        }
    }
    std::cout << "End listen_for_lp_data" << std::endl;
}
```

Una parte fondamentale della stabilizzazione ha riguardato la riscrittura del distruttore del `Supervisor`, che in precedenza lasciava thread e socket aperti, causando crash o deadlock durante la chiusura del framework. La nuova implementazione gestisce in modo deterministico la terminazione dei thread, rimuovendo gli utilizzi di `detach()` e sostituendoli con `join()`, e la chiusura di tutti i socket ZeroMQ, includendo un controllo di errore per ciascuno di essi. Infine, il contesto ZeroMQ viene esplicitamente arrestato e chiuso dopo la terminazione completa dei thread e il rilascio dei socket, garantendo una deallocazione ordinata e prevenendo accessi a risorse non più valide:

```
Supervisor::~Supervisor() {
    if (lp_data_thread.joinable()) lp_data_thread.join();
    if (hp_data_thread.joinable()) hp_data_thread.join();
    if (result_thread.joinable())  result_thread.join();

    // Chiusura socket di comando
    if (socket_command) {
        try { socket_command->close(); }
        catch (const zmq::error_t& e) {
            std::cerr << "Error while closing socket_command: {}" << e.what() << std::endl;
        }
        delete socket_command; socket_command = nullptr;
    }

    // Chiusura socket di input LP e HP
    if (socket_lp_data) {
        try { socket_lp_data->close(); }
```

```
18          catch (const zmq::error_t& e) {
19              std::cerr << "Error while closing socket_lp_data: {}" << e.
    what() << std::endl;
20          delete socket_lp_data; socket_lp_data = nullptr;
21      }
22      if (socket_hp_data) {
23          try { socket_hp_data->close(); }
24          catch (const zmq::error_t& e) {
25              std::cerr << "Error while closing socket_hp_data: {}" << e.
    what() << std::endl;
26          }
27          delete socket_hp_data; socket_hp_data = nullptr;
28      }
29
30      // Chiusura socket di risultato LP e HP
31      if (!socket_lp_result.empty()) {
32          for (auto* socket : socket_lp_result) {
33              if (socket) {
34                  try { socket->close(); }
35                  catch (const zmq::error_t& e) {
36                      std::cerr << "Error while closing socket_lp_result: {}"
     << e.what() << std::endl;
37                  }
38                  delete socket; socket = nullptr;
39              }
40          }
41          socket_lp_result.clear();
42      }
43      if (!socket_hp_result.empty()) {
44          for (auto* socket : socket_hp_result) {
45              if (socket) {
46                  try { socket->close(); }
47                  catch (const zmq::error_t& e) {
48                      std::cerr << "Error while closing socket_hp_result: {}"
     << e.what() << std::endl;
49                  }
50                  delete socket; socket = nullptr;
51              }
52          }
53          socket_hp_result.clear();
54      }
55
56      // Chiusura socket di monitoraggio
57      if (socket_monitoring) {
```

```
58          try { socket_monitoring->close(); }
59          catch (const zmq::error_t& e) {
60              std::cerr << "Error while closing socket_monitoring: {}" << e.
    what() << std::endl;
61          }
62          delete socket_monitoring; socket_monitoring = nullptr;
63      }
64
65      // Chiusura del contesto ZeroMQ
66      zmq_ctx_shutdown(context.handle());
67      try { context.close(); }
68      catch (const zmq::error_t& e) {
69          std::cerr << "Error while closing ZMQ Context: {}" << e.what() <<
    std::endl;
70      }
71
72      // Deallocazione logger
73      if (logger) { delete logger; logger = nullptr; }
74  }
```

4.3 Nuova politica di gestione delle code thread-safe

Con la stabilizzazione del framework e la risoluzione delle principali criticità strutturali, la fase successiva del lavoro si è concentrata sull'ottimizzazione della gestione delle code di comunicazione, elemento centrale per garantire la corretta distribuzione dei dati tra i thread e mantenere prestazioni stabili in scenari real-time.

Durante le prime esecuzioni del framework, uno dei problemi più ricorrenti era rappresentato da altri errori di segmentazione generati nelle funzioni di invio e ricezione dei dati, in particolare in `Supervisor::send_result()`. Il crash si verificava quando un thread tentava di accedere o rimuovere un elemento da una coda già svuotata da un altro thread concorrente. In precedenza infatti, le code erano implementate come semplici `std::queue` condivise tramite `std::shared_ptr`, senza alcun meccanismo di protezione rispetto all'accesso simultaneo. Ad esempio, la chiamata combinata di `front()` e `pop()`, risultava quindi non atomica: tra la lettura dell'elemento in testa e la sua rimozione po-

teva intervenire un altro thread, invalidando il riferimento e causando un `segmentation fault`.

Per risolvere definitivamente questo problema è stata introdotta una nuova classe generica `ThreadSafeQueue<T>` (Listato 4.1), che ridefinisce le operazioni standard delle code garantendo la sicurezza dei dati in contesti concorrenti. La classe, sviluppata come header indipendente e poi integrata nei moduli principali del framework, contiene:

- un `std::mutex` per la protezione dell'accesso alla coda;
- una `std::condition_variable` per sincronizzare i thread in attesa;
- un flag di stato `_stop` per gestire correttamente la fase di chiusura;
- un insieme di metodi quali `push()`, `front()`, `get()`, `pop()`, `empty()`, `size()` e `notify_all()`, ridefiniti per garantire atomicità e consistenza.

Il cuore della soluzione è il metodo `get()`, che combina in un'unica operazione atomica la lettura e la rimozione di un elemento dalla coda, evitando la necessità di chiamate separate a `front()` e `pop()` e prevenendo così race conditions tra thread. In precedenza, il codice nel metodo `Supervisor::send_result()` utilizzava due chiamate separate:

```
1 auto data = result_queue->front();
2 result_queue->pop();
```

Ora l'accesso è stato sostituito da una singola istruzione:

```
1 auto data = result_queue->get();
```

eliminando così la possibilità di conflitti tra thread concorrenti e rendendo il codice più chiaro e sicuro. Inoltre, le funzioni di accesso impiegano una `condition_variable` per gestire in modo efficiente la sincronizzazione tra thread produttori e consumatori: i thread che tentano di prelevare dati da una coda vuota vengono sospesi automaticamente fino a quando un nuovo elemento non viene inserito (attraverso una `push()` che invoca `notify_one()`) oppure fino a quando non viene impostato il flag di arresto. In questo

modo si evita il ricorso a cicli di attesa attiva (busy waiting) e si garantisce un risveglio immediato e controllato dei thread al verificarsi di una condizione utile o di terminazione.

```
template <typename T>
class ThreadSafeQueue {
private:
    std::queue<T> queue;
    mutable std::mutex mtx;
    std::condition_variable condvar;
    bool _stop = false; // Flag per indicare l'arresto

public:
    ThreadSafeQueue() = default;
    ~ThreadSafeQueue() = default;

    void push(const T& value) {
        std::lock_guard<std::mutex> lock(mtx);
        queue.push(value);
        condvar.notify_one();
    }

    T front() {
        std::unique_lock<std::mutex> lock(mtx);
        condvar.wait(lock, [this] { return _stop || !queue.empty(); });
        if (_stop) {
            throw std::runtime_error("ThreadSafeQueue stopped");
        }
        return queue.front();
    }

    T get() {
        std::unique_lock<std::mutex> lock(mtx);
        condvar.wait(lock, [this] { return _stop || !queue.empty(); });
        if (_stop) {
            throw std::runtime_error("ThreadSafeQueue stopped");
        }
        T value = queue.front();
        queue.pop();
        return value;
    }

    void pop() {
        std::unique_lock<std::mutex> lock(mtx);
        condvar.wait(lock, [this] { return _stop || !queue.empty(); });
```

```
42          if (_stop) {
43              return; // Esce dalla funzione e termina l'elaborazione
44          }
45          queue.pop();
46      }
47
48      bool empty() const {
49          std::lock_guard<std::mutex> lock(mtx);
50          return queue.empty();
51      }
52
53      size_t size() const {
54          std::lock_guard<std::mutex> lock(mtx);
55          return queue.size();
56      }
57
58      void notify_all() {
59          std::lock_guard<std::mutex> lock(mtx);
60          _stop = true;
61          condvar.notify_all();
62      }
63  };
```

Listing 4.1: La classe ThreadSafeQueue con la ridefinizione dei metodi di std::queue per gestire ottimalmente la concorrenza tra thread

Tutte le code utilizzate nel framework sono state quindi convertite in istanze di `ThreadSafeQueue`, condivise tra i componenti tramite `std::shared_ptr`:

```
low_priority_queue = std::make_shared<ThreadSafeQueue<std::vector<uint8_t
    >>>();
high_priority_queue = std::make_shared<ThreadSafeQueue<std::vector<uint8_t
    >>>();
result_lp_queue = std::make_shared<ThreadSafeQueue<std::vector<uint8_t>>>()
    ;
result_hp_queue = std::make_shared<ThreadSafeQueue<std::vector<uint8_t>>>()
    ;
```

Oltre a garantire la sicurezza nell'accesso concorrente ai dati, la nuova implementazione assicura anche una terminazione controllata e priva di blocchi. Quando il framework riceve un comando di arresto, ciascun componente del sistema invoca il metodo `notify_all()` su tutte le proprie code, risvegliando immediatamente i thread

eventualmente sospesi in attesa di nuovi dati:

```
    // Notify all threads that are waiting on the queues
    low_priority_queue->notify_all();
    high_priority_queue->notify_all();

    if (internal_thread && internal_thread->joinable()) {
        internal_thread->join();
    }
```

La chiamata a `notify_all()` imposta internamente il flag `_stop` e invia un segnale di risveglio a tutte le condition variables associate alle code. In questo modo, i metodi `get()`, `front()` e `pop()` rilevano la condizione di arresto, interrompono in modo sicuro l'attesa bloccante e restituiscono il controllo al flusso principale di terminazione. Fatto ciò è possibile richiamare la `join()` sui thread da terminare, la quale assicura quindi che tutte le operazioni pendenti vengano completate prima della chiusura definitiva, evitando situazioni di deadlock o memoria corrotta.

4.4 Logging configurabile

Durante la reingegnerizzazione del framework, una delle necessità emerse è stata la definizione di un sistema di logging flessibile, efficiente e configurabile. Nella versione originale in Python, come anche nelle prime versioni del porting C++, la registrazione dei log era gestita tramite semplici istruzioni `print()` o `std::cout`, distribuite in praticamente tutti i punti principali del codice. Questa soluzione, seppur utile nelle fasi iniziali di debug, introduceva un notevole overhead in fase di esecuzione, soprattutto in contesti real-time, dove l'I/O su console rappresenta un collo di bottiglia rilevante. In particolare, la scrittura su standard output all'interno di cicli frequenti o thread concorrenti può generare ritardi non trascurabili e influire negativamente sulla prevedibilità temporale del sistema.

Per risolvere queste criticità è stato introdotto un sistema di logging basato su

`spdlog`[4], una libreria C++ open source ad alte prestazioni, progettata per supportare applicazioni multi-thread e ambienti a bassa latenza. L'integrazione è stata realizzata attraverso una classe dedicata, denominata `WorkerLogger`, che incapsula la configurazione, la gestione e la formattazione dei messaggi di log, offrendo allo stesso tempo una piena modularità e la possibilità di attivare o disattivare il logging già in fase di compilazione.

4.4.1 Configurazione a compile-time e riduzione dell'overhead

Per evitare che il logging incidesse sulle prestazioni anche quando non necessario, è stata introdotta una macro di compilazione `ENABLE_LOGGING`, configurabile direttamente da `CMakeLists.txt`:

```
// Per abilitare il logging compilare con: cmake -DENABLE_LOGGING=ON ..
option(ENABLE_LOGGING "Enable logging" OFF)

if(ENABLE_LOGGING)
    add_compile_definitions(ENABLE_LOGGING)
endif()
```

In questo modo, quando la macro è definita a `OFF`, le chiamate ai metodi di logging vengono completamente escluse in fase di compilazione, riducendo il peso del file binario relativo e azzerando l'overhead a runtime. La classe `WorkerLogger` definisce infatti due versioni dei metodi principali: una effettiva, compilata solo se `ENABLE_LOGGING` è attiva, e una "vuota", che non esegue alcuna operazione in caso contrario:

```
#ifdef ENABLE_LOGGING
    void debug(const std::string& msg, const std::string& extra = "");
    void info(const std::string& msg, const std::string& extra = "");
    ...
#else
    void debug(const std::string&, const std::string& = "") {}
    void info(const std::string&, const std::string& = "") {}
    ...
#endif
```

[4] https://github.com/gabime/spdlog

Questa scelta consente di adattare il comportamento del framework a seconda del contesto: ad esempio durante lo sviluppo e i test è possibile abilitare un logging esteso, mentre nelle versioni destinate a piattaforme edge o ambienti di produzione il logging può essere completamente disattivato, massimizzando le prestazioni.

4.4.2 Struttura e funzionamento della classe WorkerLogger

La classe `WorkerLogger` utilizza i *sink* di `spdlog` per gestire in modo flessibile l'output dei log. Un sink rappresenta un canale di destinazione per i messaggi di log: ogni volta che viene generato un messaggio, esso viene inviato a uno o più sink registrati. Esistono diversi tipi di sink, come quello per la console, per i file, per i flussi di rete o per destinazioni personalizzate. Questo approccio modulare consente di combinare più output contemporaneamente (ad esempio, scrivere su file e mostrare su console) o di disattivare completamente la registrazione dei log. A seconda della configurazione, quindi, i messaggi possono essere scritti su file di log, visualizzati a terminale oppure ignorati del tutto. Grazie a questa implementazione, il sistema di logging è anche pienamente thread-safe e non blocca i thread concorrenti, anche in presenza di output simultanei. Ogni logger è identificato da un nome univoco e può essere configurato con un livello di verbosità compreso tra 1 (solo errori) e 5 (debug dettagliato).

Il costruttore accetta i parametri `logger_name`, `log_file`, `level` e `logging_mode`, che determinano la modalità operativa:

```
1  WorkerLogger::WorkerLogger(const std::string& logger_name,
2                             const std::string& log_file,
3                             spdlog::level::level_enum level,
4                             const std::string& logging_mode) {
5      std::vector<spdlog::sink_ptr> sinks;
6
7      if (logging_mode == "file" || logging_mode == "both") {
8          auto file_sink = std::make_shared<spdlog::sinks::basic_file_sink_mt
       >(log_file, true);
9          file_sink->set_pattern("[%Y-%m-%d %H:%M:%S.%e] [%l] %v");
10         sinks.push_back(file_sink);
```

```
    }

    if (logging_mode == "console" || logging_mode == "both") {
        auto console_sink = std::make_shared<spdlog::sinks::
    stdout_color_sink_mt>();
        console_sink->set_pattern("[%Y-%m-%d %H:%M:%S.%e] %^[%l]%$ %v");
        sinks.push_back(console_sink);
    }

    if (!sinks.empty()) {
        logger = std::make_shared<spdlog::logger>(logger_name, sinks.begin
    (), sinks.end());
        logger->set_level(level);
    } else {
        auto null_sink = std::make_shared<spdlog::sinks::null_sink_mt>();
        logger = std::make_shared<spdlog::logger>(logger_name, null_sink);
    }
}
```

4.4.3 Configurazione via JSON e livelli di verbosità

La modalità di logging è configurabile a runtime tramite il file di configurazione JSON del framework che definisce i parametri: `logs_path`, `logging` (equivalente al parametro `logging_mode` del costruttore di `WorkerLogger`) e `logs_level`:

```
{
    "logs_path": "/tmp/",
    "logging": "both",
    "logs_level": 5,
    "comment": "logging=file|console|both|none"
}
```

I log vengono salvati nel percorso specificato da `logs_path`, mentre il campo `logging` controlla la modalità (solo file, solo console, entrambi o nessuno). Il parametro `logs_level` stabilisce invece la soglia di dettaglio: valori più bassi limitano l'output a errori e messaggi critici, mentre livelli superiori abilitano la stampa di informazioni diagnostiche dettagliate, utili per il debugging o l'analisi del comportamento del sistema. `spdlog` adotta infatti una gerarchia di livelli di log che a runtime stabilisce quali messaggi ven-

gono effettivamente registrati in base alla soglia impostata. I livelli, dal meno al più verboso, sono i seguenti:

- **off**: nessun messaggio viene registrato.
- **critical**: registra solo i messaggi critici.
- **err**: include `err` e `critical`.
- **warn**: include `warn`, `err` e `critical`.
- **info**: include `info`, `warn`, `err` e `critical`.
- **debug**: include `debug`, `info`, `warn`, `err` e `critical`.
- **trace**: registra tutti i messaggi, dal più dettagliato al meno.

Grazie a questa gerarchia, il sistema consente di regolare la quantità di informazioni prodotte in base alle esigenze operative: durante lo sviluppo è possibile attivare livelli più verbosi (`debug` o `trace`), mentre per il deployment è meglio limitarli a `err` o `critical` per ridurre l'overhead e mantenere i log più puliti.

L'uso pratico dei livelli di log può essere visto ad esempio nei blocchi di gestione delle eccezioni, dove è consigliabile mostrare i messaggi critici riguardanti gli errori sollevati durante l'esecuzione:

```
catch (const std::exception& e) {
    logger->critical("Exception in listen_for_result: {}", e.what());
}
```

oppure:

```
catch (const std::exception& e) {
    logger->error("Exception while sending results for manager at index {}:
     {}", indexmanager, e.what());
}
```

Analogamente, per la normale esecuzione del flusso applicativo, i messaggi informativi vengono registrati con il livello `info`, utile per il monitoraggio dell'attività:

```
logger->info("End listen_for_result: {}", globalname);
```

In precedenza, messaggi di questo tipo venivano gestiti tramite output standard, ad esempio:

```
std::cout << "End listen_for_result" << std::endl;
```

L'adozione di `spdlog` consente quindi di sostituire l'output diretto su console con un sistema di logging strutturato, filtrabile e thread-safe, migliorando la tracciabilità e l'analisi del comportamento del sistema in contesti complessi e multi-thread.

4.5 Gestione dei messaggi generici

Nelle prime versioni, il framework `RTA-DP` prevedeva una gestione rigida e fortemente tipizzata dei dati in transito, pensata per formati di messaggi a lunghezza fissa e con struttura definita staticamente. Con l'evoluzione del sistema e la necessità di supportare flussi di dati eterogenei provenienti da sorgenti differenti, è stato necessario introdurre un modello più flessibile, capace di gestire messaggi binari di dimensione variabile in modo sicuro e scalabile.

Il framework, infatti, non si occupa di interpretare il contenuto dei dati, ma di fornire un'infrastruttura generica per il loro scambio e trattamento in tempo reale. La comunicazione avviene tramite la libreria ZeroMQ, che consente di trasmettere messaggi binari o testuali tra i vari moduli in modo asincrono e thread-safe. In questo contesto, il framework è completamente agnostico rispetto al formato dei dati: essi possono rispettare uno standard scientifico (ad esempio CCSDS[5]), essere codificati in JSON o costituire un buffer binario arbitrario.

[5]CCSDS: Consultative Committee for Space Data Systems, standard di telemetria e scambio dati in ambito spaziale.

4.5.1 Standardizzazione del formato dei messaggi

Per supportare la variabilità dei formati e delle dimensioni dei dati, ogni messaggio in ingresso inizia con un campo di 4 byte che rappresenta la dimensione complessiva del payload. Questa informazione consente al framework di allocare dinamicamente un buffer di dimensione adeguata, copiando successivamente il contenuto effettivo del messaggio all'interno della memoria. La scelta progettuale permette di gestire in modo uniforme dati di tipo eterogeneo, evitando accessi fuori dai limiti e semplificando la logica di parsing nei moduli applicativi:

```
int32_t size;
memcpy(&size, data.data(), sizeof(int32_t));  // Lettura della dimensione

std::vector<uint8_t> vec(size); // Buffer per contenere il messaggio
memcpy(vec.data(), static_cast<const uint8_t*>(data.data()) + sizeof(
    uint32_t), size);
```

Questa logica consente di gestire in modo sicuro messaggi binari di lunghezza variabile, indipendentemente dal loro contenuto. Il framework, una volta ricostruito il messaggio completo, lo passa ai moduli applicativi (come i vari worker) per le successive operazioni di interpretazione o elaborazione.

In un'ottica di ottimizzazione futura, l'allocazione dinamica della memoria potrà essere sostituita da una gestione statica basata su buffer preallocati, definiti in fase di pre-compilazione. Tale approccio permetterebbe di ridurre l'overhead di allocazione e frammentazione, migliorando le prestazioni complessive del framework in scenari di carico elevato.

4.5.2 Serializzazione e costruzione dei messaggi

Lato producer, è stato introdotto un meccanismo simmetrico di costruzione dei messaggi che consente di serializzare qualunque struttura dati in forma binaria, anteponendo la

dimensione totale all'inizio del buffer. In questo modo, il framework può inviare messaggi di qualunque tipo (testuale o binario) senza conoscerne la semantica:

```
template <typename T>
std::vector<uint8_t> serializeMessage(const T& message) {
    int32_t size = sizeof(T);
    std::vector<uint8_t> buffer(sizeof(int32_t) + size);
    memcpy(buffer.data(), &size, sizeof(int32_t));  // Scrive la size
    memcpy(buffer.data() + sizeof(int32_t), &message, size); // Scrive il
    payload
    return buffer;
}
```

Questo approccio consente di mantenere la compatibilità tra moduli diversi e di garantire una gestione coerente dei messaggi, indipendentemente dal loro contenuto. Il framework si occupa quindi della serializzazione, trasmissione e ricezione dei dati, lasciando alle implementazioni specializzate il compito di interpretare e processare le informazioni trasportate.

La logica di elaborazione dei messaggi è demandata ai thread `Worker`, i quali prelevano i dati dalle code condivise e li processano in base alla funzione applicativa associata. Il framework fornisce le primitive necessarie per la gestione concorrente dei dati, il coordinamento dei thread e la sincronizzazione degli accessi alle code, ma non impone alcuna semantica sul contenuto dei messaggi. Un worker può quindi essere configurato per:

- filtrare o validare i dati ricevuti;
- applicare algoritmi di pre-processing o trasformazione;
- inoltrare i risultati verso altri moduli del framework o verso pipeline esterne di analisi.

Questa architettura modulare consente di mantenere il nucleo del framework indipendente dal dominio applicativo, favorendo il riutilizzo del codice e la specializzazione in contesti diversi, come flussi di dati sperimentali o simulazioni di acquisizione.

4.6 Ottimizzazioni e miglioramento delle prestazioni

Una volta stabilizzato il framework e risolte le principali criticità strutturali e di sincronizzazione, è emersa la necessità di ridurre il consumo di risorse hardware, in particolare l'uso della CPU e della memoria RAM e di incrementare maggiormente le prestazioni del sistema per un processing real-time a bassa latenza. Durante le prime esecuzioni, l'osservazione tramite strumenti di monitoraggio forniti da Linux come `top` e `htop`[6] ha mostrato che il sistema manteneva un carico CPU costantemente vicino al 100% nella maggior parte dei core anche in condizioni di inattività, mentre la memoria allocata cresceva rapidamente in presenza di cicli di elaborazione prolungati. Questo comportamento era dovuto principalmente a:

- cicli di attesa (`while`) non bloccanti che consumavano CPU anche in idle;
- thread di servizio (come il monitoraggio) attivi costantemente, indipendentemente dal carico effettivo;
- logging pesante, che introduceva un overhead notevole soprattutto in presenza di più thread concorrenti;
- copie ridondanti dei buffer durante il parsing e la serializzazione dei pacchetti.

Riduzione del carico CPU e ottimizzazione dei cicli di attesa

Molti thread del framework eseguivano cicli di polling o attese attive senza alcuna forma di sincronizzazione o temporizzazione, generando un consumo costante di risorse. In assenza di una pausa esplicita, i thread rimanevano in esecuzione continua, controllando ripetutamente lo stato delle variabili di controllo e delle code senza mai rilasciare il controllo della CPU. Questo comportamento, noto come *busy waiting*, porta i core a

[6] `top` e `htop` sono strumenti Linux per il monitoraggio in tempo reale dei processi e dell'utilizzo di CPU e memoria; `htop` offre un'interfaccia più interattiva e leggibile rispetto a `top`.

rimanere costantemente al 100% di utilizzo, anche quando il sistema non elabora effettivamente dati. L'introduzione di brevi pause (`sleep`) all'interno di questi cicli consente invece al sistema operativo di sospendere temporaneamente il thread, liberando la CPU per altre attività e riducendo drasticamente il consumo energetico. Con questa semplice modifica, il carico dei core è passato dal 100% a valori prossimi allo zero, anche con la pipeline a pieno utilizzo.

Un caso tipico era quello dei cicli principali del `Supervisor`, che verificava in continuo la disponibilità di comandi o dati senza sospendersi, mantenendo la CPU impegnata anche in assenza di traffico. L'introduzione di pause temporizzate mirate ha permesso di mantenere il comportamento reattivo del sistema senza gravare sulla CPU:

```
while (continueall) {
    listen_for_commands();
    std::this_thread::sleep_for(std::chrono::seconds(1)); // Riduce l'uso
    CPU in idle
}
```

In modo analogo, anche durante la fase di "cleaned shutdown" venivano effettuati controlli continui sulle code dei `WorkerManager` fino al loro svuotamento, per chiudere il sistema in modo sicuro. Una semplice pausa tra una verifica e l'altra ha portato a un'ulteriore riduzione del carico CPU, senza impatti negativi sui tempi di terminazione:

```
while (manager->getLowPriorityQueue()->size() != 0 ||
       manager->getHighPriorityQueue()->size() != 0) {
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
}
```

Anche il thread di monitoraggio, vista la sua esecuzione continua, è stato ottimizzato tramite intervalli configurabili in modo da "alleggerire" la CPU tra un'iterazione di ciclo e l'altra:

```
void MonitoringThread::run() {
    while (!stop_event) {
        json monitoring_data = monitoringpoint.get_data();
        std::string monitoring_data_str = monitoring_data.dump();
        zmq::message_t message(monitoring_data_str.begin(),
```

```
    monitoring_data_str.end());  // Create ZMQ message
6         socket_monitoring.send(message, zmq::send_flags::none);  // Send
    the message
7         std::this_thread::sleep_for(std::chrono::seconds(1));  // Sleep for
     1 second
8     }
9 }
```

Ottimizzazione della memoria

Un altro intervento rilevante per il miglioramento delle prestazioni del framework ha riguardato la riduzione delle copie di dati e una più efficiente distribuzione del carico di elaborazione tra i thread. Nelle prime versioni, il framework effettuava numerose copie di pacchetti e vettori binari (`std::vector<uint8_t>`) durante il parsing e il trasferimento tra le code, generando un overhead significativo dovuto alle operazioni di copia e riallocazione dinamica. Con la nuova gestione, le funzioni di serializzazione e parsing (come `serializePacket()` e `pushPacketToQueue()`) sono state riorganizzate per lavorare direttamente su vettori preallocati, riducendo le `memcpy()` e le allocazioni temporanee. Inoltre, l'inserimento dei pacchetti nelle code sfrutta le "move semantics" implicite del linguaggio: quando il risultato di `serializePacket()` (che restituisce un oggetto temporaneo di tipo `std::vector<uint8_t>`) viene passato alla coda, il contenuto viene spostato anziché copiato, evitando duplicazioni di memoria:

```
1 manager->getLowPriorityQueue()->push(serializePacket(*packet_wf));
```

Impatto del sistema di logging configurabile

Il nuovo sistema di logging basato su `spdlog` ha avuto un impatto positivo anche sul piano delle prestazioni. La possibilità di disattivarlo completamente tramite la macro `ENABLE_LOGGING` consente di compilare versioni leggere del framework prive di log a runtime, ideali per l'esecuzione su dispositivi embedded o a basso consumo. La disattivazione

del logging ha comportato una riduzione sensibile dei tempi di esecuzione e latenza, in particolare nei moduli con cicli ad alta frequenza di logging (come i `WorkerManager` o `WorkerThread`).

In particolare, la disattivazione selettiva del logging ha portato ai seguenti miglioramenti:

- riduzione dell'overhead di `std::cout` e delle stampe concorrenti su console;
- incremento del throughput nelle pipeline a pieno carico;
- migliore controllo del livello di verbosità e selettività dei messaggi di log.

Capitolo 5

Estensione di RTA-DP con ML per il caso d'uso GammaSky

Dopo la fase di stabilizzazione e ottimizzazione descritta nel capitolo precedente, il framework `RTA-DP` è stato esteso e adattato per essere impiegato nel progetto `GammaSky`. RTA-DP implementa la pipeline di analisi e ricostruzione in tempo reale, gestendo lo streaming delle waveform acquisite e la ricostruzione fisica degli eventi gamma atmosferici attraverso l'inferenza basata su modelli di machine learning ottimizzati per l'esecuzione su hardware a risorse limitate.

L'integrazione di GammaSky ha richiesto una serie di modifiche che non hanno alterato la struttura di base del framework, ma ne hanno esteso le funzionalità per supportare un flusso dati specifico (vedi Sezione 2.3). In particolare, il sistema di streaming e serializzazione è stato adattato per supportare i messaggi provenienti dal modulo di acquisizione `DAM` in esecuzione su `Red Pitaya`. Il fulcro del lavoro però, è stata la creazione di un nuovo worker dedicato all'inferenza di modelli di ML ottimizzati tramite tecniche di *quantizzazione*. È stato quindi definito un flusso dati completo, dal modello `R0` (waveform grezze) al formato `DL2` (lista dei fotoni ricostruiti), con output compatibili con i formati scientifici standard come `HDF5`.

5.1 Adattamento del producer e dello streamer

Per abilitare e testare il funzionamento della pipeline RTA-DP applicata per GammaSky, è stato necessario per prima cosa sostituire il `CCSDSProducer`, precedentemente impiegato nella fase di validazione del framework generico, con il Data Acquisition Module (`DAM`). Testare la pipeline processando direttamente i dati che arrivano dal DAM non è fattibile se non si ha a disposizione la scheda di acquisizione ed il detector. Per poter semplificare i test, è stato quindi realizzato uno streamer ad alte prestazioni denominato `GFSE`, utilizzato per testare la pipeline in condizioni di pieno carico. Lo streamer legge i file `DL0` da disco (prodotti da un altro software di acquisizione ereditato da GammaFlash), ricostruisce i pacchetti secondo il formato `R0` definito dal DAM (un pacchetto binario contenente header e payload) e invia i dati corrispondenti sotto forma di messaggi tramite socket `ZeroMQ` in modo continuato (fino a 200 pacchetti al secondo), per validare il comportamento e la stabilità della pipeline. Questa procedura è stata eseguita principalmente come stress test per la fase di inferenza real-time e per la scrittura dei file DL2 in output.

5.1.1 Adozione di ZeroMQ e nuovo formato di pacchetto

Una delle principali modifiche apportate ai due producer ha riguardato il protocollo di comunicazione. La logica basata su socket `TCP` tradizionali è stata sostituita con un modello `PUB/SUB` implementato tramite `ZeroMQ`, garantendo maggiore efficienza e scalabilità per l'interfacciamento con i consumer. Per consentire la compatibilità tra il DAM e RTA-DP, è stato definito un formato di pacchetto con header uniformato, implementato nel file `packet.h`, valido per entrambi i sistemi. Ogni pacchetto, trasmesso come messaggio, segue la struttura:

- un prefisso di 4 byte contenente la dimensione totale del pacchetto;

- un header comune (`HeaderDams`), che identifica sorgente, tipo, sequenza e CRC del messaggio;
- un payload che può assumere due forme principali:
  - `Data_HkDams`, per i pacchetti di housekeeping;
  - `Data_WaveHeader` + `Data_WaveData`, per i pacchetti contenenti waveform.

La definizione sintetica delle strutture è riportata di seguito:

```
class HeaderDams {  // Header comune ad entrambi i packets
public:
    uint8_t  start;     // 0x8D
    uint8_t  apid;
    uint16_t sequence;
    uint16_t runID;
    uint16_t size;      // Dimensione payload
    uint32_t crc;       // Controllo
};

class Data_HkDams {
public:
    uint8_t  type;      // 0x03
    uint8_t  subType;   // 0x01
    uint8_t  state;
    uint8_t  flags;
    uint32_t waveCount;
    struct timespec ts; // Timestamp di acquisizione
};

class Data_WaveHeader {
public:
    uint8_t  type;      // 0xA1
    uint8_t  subType;   // 0x01
    uint16_t sessionID;
    uint16_t configID;
    uint8_t  timeSts, ppsSliceNo, year, month, day, hh, mm, ss;
    uint32_t us;
    struct timespec ts; // Tempo assoluto
    uint32_t dec, currOff, trigOff, size;
};

```

```
33 class Data_WaveData {
34 public:
35     uint8_t  type;      // 0xA1
36     uint8_t  subType;   // 0x02
37     uint8_t  spare0, spare1;
38     uint32_t buff[1020]; // I valori della waveform
39 };
```

Questa nuova struttura ha semplificato la gestione dei messaggi nel consumer permettendo di allocare dinamicamente la memoria in base al valore di `size` letto nei primi 4 byte del prefisso e, per questo caso d'uso, di identificare immediatamente il tipo di dato contenuto nel messaggio ricevuto. In questo modo, il meccanismo di gestione di messaggi generici introdotto nel framework (vedi Sezione 4.5) viene qui applicato a un formato di dati specifico, permettendo a RTA-DP di integrarsi direttamente con il sistema di acquisizione di GammaSky.

5.1.2 Invio di pacchetti completi

A seguito di questo aggiornamento, l'integrazione ha richiesto altre modifiche simmetriche per entrambi i producer. Nel DAM, i file `tchandler_wave.cpp` e `tchandler_hk.cpp` sono stati aggiornati per generare pacchetti completi (header + payload) preceduti dal prefisso di dimensione. La principale differenza rispetto alle versioni precedenti (come quella usata per Gamma-Flash) riguarda la gestione dei pacchetti. Nella precedente implementazione basata su TCP puro, il flusso era soggetto a frammentazione in funzione della MTU[1] e non vi era alcuna garanzia di corrispondenza tra una singola `send()` e una singola `recv()`. Ciò richiedeva la ricostruzione esplicita dei messaggi lato ricezione. Con l'introduzione di ZeroMQ, il quale è comunque basato su TCP, questa complessità viene però eliminata: la libreria fornisce un livello di astrazione che garantisce la consegna atomica dei messaggi, evitando frammentazioni e semplificando la logica di comunicazione.

[1] La *Maximum Transmission Unit* (MTU) è la dimensione massima, in byte, di un singolo pacchetto trasmissibile su un link di rete senza frammentazione.

Questo ha reso possibile unificare header e payload in un unico buffer dati. La funzione `sendWaveform()` crea ora un singolo blocco dati contenente l'header (`HeaderDams`) e la sezione dati (`Data_WaveHeader` + `Data_WaveData`), evitando la delimitazione manuale dei messaggi che avveniva in precedenza. Un estratto della funzione è riportato di seguito:

```
// Definizione delle dimensioni (header + payload)
const size_t headerSize = sizeof(HeaderDams);
const size_t waveHeaderSize = sizeof(Data_WaveHeader);
const size_t waveDataSize = sizeof(Data_WaveData);
const size_t packetPayloadSize = headerSize + waveHeaderSize + waveDataSize
    ;
const size_t totalSize = sizeof(uint32_t) + packetPayloadSize;

// Costruzione del pacchetto unificato
uint8_t unifiedBuff[packetPayloadSize];
HeaderDams* header = reinterpret_cast<HeaderDams*>(unifiedBuff);
header->size = (uint16_t)(waveHeaderSize + waveDataSize);
header->encode(); // Calcolo del CRC

// Copia del contenuto del pacchetto
memcpy(unifiedBuff + headerSize, &waveHeader, waveHeaderSize);
memcpy(unifiedBuff + headerSize + waveHeaderSize, waveData, waveDataSize);

// Aggiunta del prefisso di dimensione
uint8_t finalBuff[totalSize];
uint32_t sizePrefix = packetPayloadSize;
memcpy(finalBuff, &sizePrefix, sizeof(uint32_t));
memcpy(finalBuff + sizeof(uint32_t), unifiedBuff, packetPayloadSize);

// Invio tramite ZeroMQ
if (g_ctrlServer.getState() == TcpServer::STT_ACTIVE)
    g_ctrlServer.send(finalBuff, totalSize);
```

Analogamente, nello streamer `gfse.py` è stata definita la funzione `create_unified_waveform_packet()`, che costruisce i pacchetti in modo coerente al nuovo formato, includendo sia i metadati che i dati binari della waveform. La funzione genera l'`HeaderDams` e il `Data_WaveHeader`, concatena i blocchi dati (`Data_WaveData`) e infine aggiunge un prefisso di 4 byte che indica la dimensione complessiva del pacchetto:

```python
def create_unified_waveform_packet(wform, srcid, npkt, crc_table):
    # HeaderDams (12 B) + Data_WaveHeader (44 B)
    header_payload = build_wave_header(wform)
    header_common  = build_header_common(wform, len(header_payload))
    unified_header = header_common + header_payload

    # Data_WaveData section: inner header + waveform samples
    waveform_data = struct.pack("<%dL" % len(wform.data), *wform.data)
    data_field = b"\xA1\x02\x00\x00" + waveform_data

    # Pacchetto finale con prefisso di lunghezza (4 B)
    unified_payload = unified_header + data_field
    size_prefix = struct.pack("<L", len(unified_payload))
    return size_prefix + unified_payload
```

Anche in questo caso il messaggio risultante viene inviato su un socket `ZMQ PUB`, con il consumer (il supervisor) che li riceverà tramite un socket `ZMQ SUB`. Questa soluzione ha reso possibile un'integrazione trasparente tra i due sistemi: lo stesso codice di parsing nel consumer di RTA-DP può ora gestire pacchetti provenienti sia dal DAM sia dallo streamer GFSE, garantendo un formato comune di comunicazione e una pipeline uniforme.

5.2 Ottimizzazione e porting del modello di ML per Edge Computing

Come descritto nel Capitolo 2, nel caso d'uso GammaSky, l'uso di intelligenza artificiale per l'analisi di fenomeni gamma ad alta energia, come i TGF, deve avvenire direttamente su di un edge computer. Quest'ultimo, riceve in streaming le waveform digitalizzate dal DAM e si occupa di stimare in tempo reale i parametri fisici di interesse, in questo caso l'energia depositata (area sottesa al picco) nel rivelatore a partire dalla forma d'onda acquisita e il tempo di arrivo dell'impulso. In questo contesto non è sufficiente disporre di un modello di deep learning preciso: il modello deve anche essere sufficientemente leggero ed efficiente da poter essere eseguito con bassa latenza e throughput elevato su

un dispositivo a risorse limitate e con vincoli di consumo energetico.

Per questo scopo è stato implementato un modello di deep learning sviluppato internamente in INAF – OAS: una rete neurale convoluzionale (CNN) che, dato un segmento di waveform, stima l'integrale del picco, proporzionale all'energia dell'evento. Come discusso nella Sezione 2.2, tale valore non corrisponde semplicemente all'integrale diretto dell'impulso, ma a una misura ricostruita che deve rimanere stabile in presenza di rumore, variazioni della forma d'onda o eventi in sovrapposizione (pile-up), condizioni in cui i metodi deterministici tradizionali mostrano limiti significativi. La rete è stata quindi addestrata su un dataset di waveform simulate e successivamente compressa mediante tecniche di quantizzazione, utilizzando LiteRT[2] e la libreria TensorFlow Model Optimization. L'obiettivo è ottenere dei modelli che rispettino due vincoli fondamentali:

- mantenere una performance sufficiente per la ricostruzione dell'area della waveform nell'intervallo di energie di interesse per GammaSky;
- ridurre il costo computazionale e l'occupazione di memoria, in modo da eseguire inferenza in tempo reale sulla Jetson ed integrare il modello in un worker C++ del framework RTA-DP.

5.2.1 Dataset e pre–processing

Il dataset utilizzato per il training e per applicare le varie tecniche di ottimizzazione è composto da 50 000 waveform simulate, ognuna costituita da 1000 campioni interi (`int16`) e accompagnate da una label, la quale indica l'area sottesa all'impulso relativo rappresentata da un numero reale a doppia precisione (`float64`). Il dataset è stato suddiviso come segue:

[2]LiteRT è il framework di TensorFlow dedicato all'esecuzione di modelli di deep learning su dispositivi edge, embedded e mobile, ottimizzato per ridurre la latenza, il consumo di memoria e i requisiti computazionali dell'inferenza. Fornisce strumenti per la conversione dei modelli in un formato leggero (`.tflite`) e supporta tecniche di ottimizzazione come quantizzazione e pruning-aware training. Fino a settembre 2024 era noto con il nome di *TensorFlow Lite*.

- **Training set**: 70% delle waveform totali (34 999), con a sua volta un 20% del subset usato come validation set;
- **Test set**: 20% delle waveform totali (10 000), contenente campioni non usati nel training su cui valutare la rete;
- **Optimization set**: il 10% delle waveform rimanenti viene usato per applicare le varie tecniche di ottimizzazione.

Prima di essere forniti alla rete, i dati vengono normalizzati come segue:

- le waveform di input sono state scalate campione per campione e poi rimodellate in un tensore di dimensione $(N, 1000, 1)$, adatto all'input richiesto dai layer convoluzionali `Conv1D`;
- i target (le aree) sono stati scalati con un `MinMaxScaler` nell'intervallo $[-1, 1]$, in modo da rendere il problema numericamente coerente: sia il training sia il test (ri-scalato poi con i parametri del training) mostrano lo stesso intervallo di valori dopo la normalizzazione.

Questa fase di pre–processing è fondamentale per due motivi: da un lato permette alla rete di concentrarsi sulla forma del segnale piuttosto che sulle scale assolute, dall'altro semplifica la successiva fase di quantizzazione, riducendo il rischio di saturazione dei valori in formato intero.

5.2.2 Architettura della rete e fase di training

Il modello di partenza è una CNN monodimensionale profonda ma relativamente compatta (circa 25 600 parametri, $\simeq$ 110 kB in formato Keras), progettata come detto per stimare l'area della waveform a partire dalla forma d'onda normalizzata. L'architettura, riportata in Tabella 5.1, segue uno schema gerarchico di feature extraction ed è composto dai seguenti layer:

- otto blocchi convoluzionali 1D con kernel di dimensione 5 e funzione di attivazione `tanh`, ciascuno seguito da un livello di `MaxPooling1D`. Questa scelta, verificata sperimentalmente, ha mostrato prestazioni più robuste rispetto a soluzioni alternative basate su convoluzioni con `stride`, in quanto il max pooling tende a preservare le attivazioni più significative riducendo al contempo gli effetti del rumore;
- un layer di `Flatten` che proietta le feature in uno spazio vettoriale di dimensione 96;
- un layer denso intermedio con 16 neuroni e funzione di attivazione `tanh`;
- un layer denso finale con 1 neurone lineare che restituisce l'area normalizzata del picco.

Tabella 5.1: Architettura del modello Area Predictor. Tutti i layer convoluzionali e densi utilizzano come funzione d'attivazione `tanh`.

Layer	**Output shape**	**# parametri**
`Conv1D` + `tanh`	$(\text{None}, 1000, 4)$	24
`MaxPooling1D`	$(\text{None}, 500, 4)$	0
`Conv1D` + `tanh`	$(\text{None}, 500, 8)$	168
`MaxPooling1D`	$(\text{None}, 250, 8)$	0
`Conv1D` + `tanh`	$(\text{None}, 250, 16)$	656
`MaxPooling1D`	$(\text{None}, 125, 16)$	0
`Conv1D` + `tanh`	$(\text{None}, 125, 32)$	2592
`MaxPooling1D`	$(\text{None}, 62, 32)$	0
`Conv1D` + `tanh`	$(\text{None}, 62, 32)$	5152
`MaxPooling1D`	$(\text{None}, 31, 32)$	0
`Conv1D` + `tanh`	$(\text{None}, 31, 32)$	5152
`MaxPooling1D`	$(\text{None}, 15, 32)$	0
`Conv1D` + `tanh`	$(\text{None}, 15, 32)$	5152
`MaxPooling1D`	$(\text{None}, 7, 32)$	0
`Conv1D` + `tanh`	$(\text{None}, 7, 32)$	5152
`MaxPooling1D`	$(\text{None}, 3, 32)$	0
`Flatten`	$(\text{None}, 96)$	0
`Dense` + `tanh`	$(\text{None}, 16)$	1552
`Dense` (output)	$(\text{None}, 1)$	17
	Totale	**25 617**

Le dimensioni principali dei tensori in uscita si riducono progressivamente da $(1000, 4)$ a $(3, 32)$ dopo l'ultimo blocco di pooling, per poi essere trasformate in un vettore e mappate sullo spazio scalare dell'integrale. La combinazione di kernel piccoli e operazioni di pooling permette di catturare la struttura locale del segnale mantenendo contenuto il numero totale di parametri allenabili (25 617). La rete è stata addestrata con la seguente configurazione:

- funzione di loss: `Huber` con $\delta = 2$. Questa funzione, usata per i modelli di regressione, è stata scelta perché più robusta agli outlier della MSE e più veloce della MAE;

- massimo di 500 epoche di training, con *early stopping* sulla `val_loss` a `patience` di 10 epoche e salvataggio del modello migliore tramite `ModelCheckpoint`, sempre in termini di `val_loss`;
- ottimizzatore: `Adam` con learning rate dell'ordine di 10^{-4}.

Il training si è arrestato alla 183-esima epoca, con una training loss dell'ordine di 1.74×10^{-5} e una validation loss di 1.46×10^{-5}. L'andamento delle loss di training e di validazione è riportato in Figura 5.1. Si può osservare una diminuzione regolare di entrambe le curve e l'assenza di overfitting significativo, con le due loss che rimangono comparabili per tutta la durata dell'addestramento. I piccoli picchi osservabili lungo la validation loss sono coerenti con il contesto. Essi infatti dipendono dal rumore del dataset e dal fatto che il batch di validazione non è identico a ogni epoca: queste variazioni locali non indicano comunque instabilità del training.

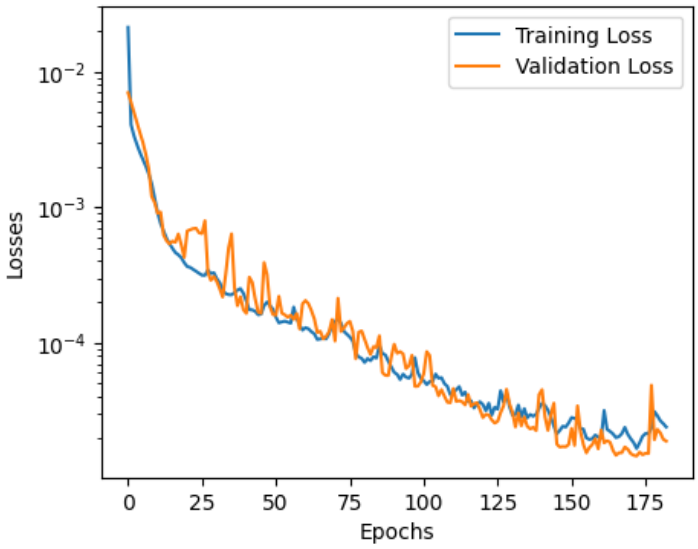


Figura 5.1: Andamento della loss di training e di validazione per il modello di CNN.

Un confronto tra valori reali e predetti sul test set (vedi Figura 5.2) mostra scarti tipicamente inferiori a poche unità percentuali (in spazio scalato $[-1, 1]$), sia per aree positive

che negative, a conferma della capacità del modello base di approssimare correttamente la relazione non lineare tra forma della waveform e area del picco.

```
integrals real:     [-0.28422842]
integrals predict:  [-0.28250363]

integrals real:     [-0.70016085]
integrals predict:  [-0.7011427]

integrals real:     [-0.68909766]
integrals predict:  [-0.6892585]

integrals real:     [-0.51646257]
integrals predict:  [-0.5178671]

integrals real:     [-0.03709665]
integrals predict:  [-0.03925757]

integrals real:     [-0.60408178]
integrals predict:  [-0.6036292]

integrals real:     [-0.20520212]
integrals predict:  [-0.20974502]

integrals real:     [0.30838901]
integrals predict:  [0.30466527]

integrals real:     [-0.72720446]
integrals predict:  [-0.72823536]

integrals real:     [-0.72987825]
integrals predict:  [-0.7287994]
```

Figura 5.2: Confronto tra le aree reali e i valori predetti dal modello su un sottoinsieme di esempi del test set.

5.2.3 Quantizzazione con LiteRT e confronto tra modelli

Per rendere il modello eseguibile in modo efficiente su edge computer, la rete addestrata è stata ottimizzata convertendola in formato `TensorFlow Lite` (LiteRT) e sottoposta a tecniche di quantizzazione post–training. La quantizzazione riduce la precisione numerica dei tensori del modello (i pesi e in parte le attivazioni) con l'obiettivo di diminuire

l'occupazione di memoria per allocare o archiviare la rete e il costo delle operazioni aritmetiche. In questo contesto sono stati considerati tre tipi di rappresentazione:

- **Modello originale di riferimento**: conversione diretta in formato a virgola mobile a singola precisione (`float32`) senza quantizzazione;
- **Quantizzazione float16**: i pesi vengono memorizzati in virgola mobile a mezza precisione;
- **Quantizzazione int8**: pesi e attivazioni sono quantizzati a 8 bit interi. Per questa modalità è stato utilizzato un *representative dataset* di waveform, ottenuto dall'optimization set (tuning set), per calibrare gli intervalli di quantizzazione.

La Figura 5.3 mostra un confronto diretto tra le dimensioni dei file dei tre modelli. Il modello originale (`float32`) occupa su disco 116.75 kB, la versione `float16` scende a 69.05 kB, mentre il modello `int8` raggiunge 50.10 kB. In termini relativi:

- la quantizzazione a `float16` riduce la dimensione del modello di circa il 41% rispetto all'originale;
- la quantizzazione a `int8` porta a una riduzione di circa il 57%.

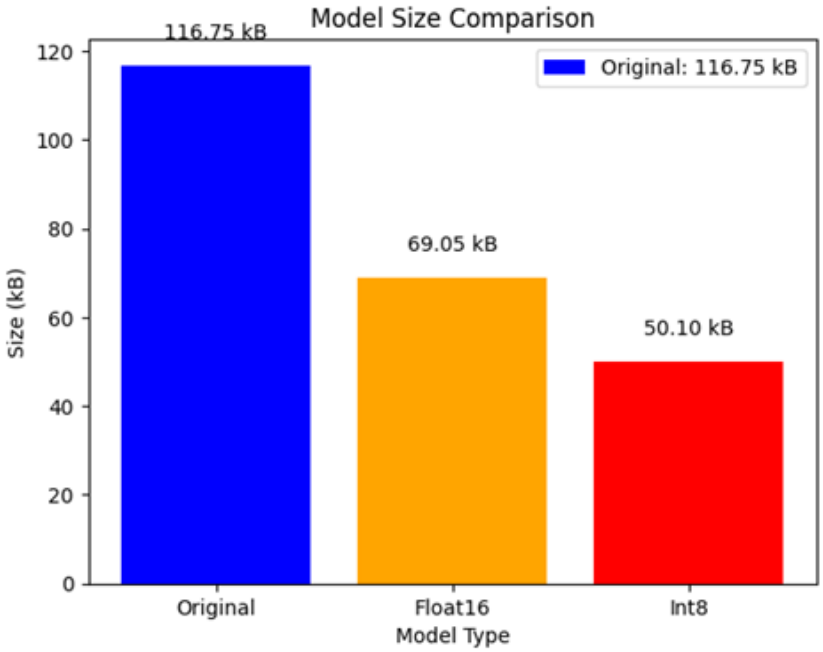


Figura 5.3: Confronto delle dimensioni dei modelli in formato TFLite: originale (`float32`), quantizzato `float16` e quantizzato `int8`.

Nel workflow di ottimizzazione non sono state applicate altre tecniche di ottimizzazione, come il pruning, al modello di rete neurale. Ciò è dovuto a vari motivi tecnici che rendono, nello specifico, questa tecnica poco vantaggiosa nel caso specifico. In primo luogo, il modello utilizzato è estremamente compatto (circa 25k parametri, per una dimensione di $\sim 100\,\text{kB}$ in formato `float32`). Introdurre sparsità in una rete così piccola non produce riduzioni significative né in memoria né in tempo di inferenza. Inoltre, il pruning adottabile tramite TensorFlow Model Optimization è di tipo non strutturato e LiteRT densifica i tensori al momento della conversione, eliminando di fatto qualsiasi beneficio computazionale. L'hardware della Jetson Orin Nano non sfrutta accelerazioni specifiche per reti prunate non strutturate, mentre supporta nativamente la quantizzazione FP16 e INT8, che garantiscono una reale riduzione dei costi computazionali. Per questi motivi l'ottimizzazione si è concentrata esclusivamente sulla quantizzazione, che rappresenta la tecnica più efficace per questo caso d'uso.

Dal punto di vista del deployment su edge, la combinazione tra modello compatto, quantizzazione (in particolare `float16` per un buon compromesso tra performance e peso) e runtime LiteRT con backend ottimizzato (attraverso la libreria `XNNPACK`) consente di eseguire inferenza su Jetson Orin Nano con un impatto limitato su memoria e consumo energetico, mantenendo al tempo stesso una precisione sufficiente per la ricostruzione dell'energia evento per evento.

5.3 Integrazione del modello di ML nella pipeline RTA-DP

Dopo la fase di ottimizzazione del modello, il passo successivo è stato integrare il modello quantizzato in LiteRT (`.tflite`) all'interno della pipeline RTA-DP, così da consentire l'esecuzione dell'inferenza in tempo reale direttamente sulla Jetson Orin Nano. In questa fase sono state affrontate tre attività fondamentali:

1. integrazione del modello quantizzato nel worker C++ del framework;
2. ottimizzazione dell'inferenza su CPU ARM tramite `XNNPACK`;
3. verifica del corretto flusso dati end-to-end (waveform grezze R0 $\rightarrow$ inferenza $\rightarrow$ aree predette).

L'integrazione avviene nella parte finale della pipeline, dove il worker legge dalla coda le waveform ricevute dal supervisor, esegue l'inferenza sul modello quantizzato e restituisce i valori degli integrali necessari alla generazione successiva dei file `DL2` in output. Il flusso operativo rimane quello definito per il caso d'uso generico del framework RTA-DP. Il producer (in questo caso `gfse.py`) invia via ZeroMQ messaggi organizzati tramite pacchetti binari serializzati in formato `R0` con i dati estratti dai file `DL0` di input, contenenti:

- un header comune sia per pacchetti waveform che housekeeping (`HeaderDams`);
- una struttura `Data_WaveHeader` seguita da una struttura `Data_WaveData`, la quale contiene il buffer di waveform compattato in parole `uint32_t`, secondo il formato definito in `packet.h`.

Ogni messaggio, come visto nella Sezione 5.1.1, è preceduto da un prefisso di 4 byte che contiene la dimensione del payload relativo. Il supervisor specializzato per questo task (`Supervisor1`) riceve il pacchetto, verifica la dimensione, identifica il tipo di dato accedendo ai campi dell'header `type/subtype`, ricostruisce un oggetto `WfPacketDams` contenente header e payload e inserisce il pacchetto serializzato nella coda a priorità dei worker. La ricostruzione del pacchetto è illustrata nello snippet seguente:

```
int32_t size;
memcpy(&size, data.data(), sizeof(int32_t));

const HeaderDams* h_ptr = reinterpret_cast<const HeaderDams*>(raw + sizeof(
    uint32_t));

const Data_WaveHeader* w_ptr = reinterpret_cast<const Data_WaveHeader*>(raw
     + sizeof(uint32_t) + sizeof(HeaderDams));

WfPacketDams packet;
packet.body.h = *h_ptr;
packet.body.w = *w_ptr;

std::memcpy(packet.body.d.buff, raw + sizeof(uint32_t) + sizeof(HeaderDams)
     + sizeof(Data_WaveHeader), U32_X_PACKET * sizeof(uint32_t));

lp_queue->push(serializePacket(packet));
```

Il worker dedicato all'inferenza (`Worker1`), d'altra parte, sarà quindi responsabile di:

1. leggere i pacchetti dalla coda;
2. estrarre le waveform;
3. pre-processarle normalizzando i campioni, in modo da allinearli alla logica usata nel training;

4. eseguire l'inferenza tramite LiteRT;

5. produrre il dato fisico da salvare in `DL2`.

5.3.1 Interprete LiteRT e delegate XNNPACK

Come preparazione dell'ambiente di inferenza e per massimizzare le prestazioni della Jetson Orin Nano, l'interprete LiteRT è stato configurato per utilizzare il delegate `XNNPACK`[3], progettato per accelerare reti piccole e medie su CPU ARM (architettura che possiede la Jetson) tramite microkernel ottimizzati e istruzioni SIMD NEON. L'inizializzazione avviene all'interno del costruttore di `Worker1`:

```
TfLiteModel* model = TfLiteModelCreateFromFile(model_path.c_str());

TfLiteXNNPackDelegateOptions xnn_opts = TfLiteXNNPackDelegateOptionsDefault
    ();
TfLiteDelegate* xnnpack = TfLiteXNNPackDelegateCreate(&xnn_opts);

TfLiteInterpreterOptions* opts = TfLiteInterpreterOptionsCreate();
TfLiteInterpreterOptionsAddDelegate(opts, xnnpack);

TfLiteInterpreter* interp = TfLiteInterpreterCreate(model, opts);
TfLiteInterpreterAllocateTensors(interp);
```

L'inferenza viene sempre eseguita internamente in `float32`, ma l'utilizzo di modelli quantizzati (`float16` o `int8`) riduce il footprint in memoria e migliora l'efficienza della cache. Nonostante LiteRT consenta l'utilizzo di thread multipli, nel caso di un modello estremamente compatto come quello utilizzato per questo caso d'uso, l'overhead supera il beneficio: è quindi stato adottato un approccio single-thread per interprete, sfruttando la parallelizzazione nativa del framework.

[3]XNNPACK è una libreria di microkernel ottimizzati per l'inferenza su CPU, sviluppata per accelerare operazioni tipiche del deep learning sfruttando istruzioni SIMD (NEON su ARM) e ottimizzazioni della gerarchia di memoria. In LiteRT funge da *delegate* per ridurre la latenza di modelli leggeri su dispositivi edge.

5.3.2 Preprocessing ed esecuzione dell'inferenza

Ogni waveform è rappresentata da 1000 "word" `uint32_t`, ciascuna contenente due campioni a 16 bit. Il worker ricostruisce i 2000 campioni originali tramite:

```
uint32_t word = aligned_buf[i];
uint16_t lo = word & 0xFFFF;
uint16_t hi = (word >> 16) & 0xFFFF;

float_wave[2*i]   = static_cast<float>(hi);
float_wave[2*i+1] = static_cast<float>(lo);
```

I campioni sono quindi normalizzati nell'intervallo $[-1, 1]$ tramite la stessa trasformazione MinMax usata durante il training:

```
x = std::clamp(x, in_min, in_max);
model_in[i] = 2.f * (x - in_min) / (in_max - in_min) - 1.f;
```

Questa operazione garantisce che i dati reali mantengano la stessa distribuzione statistica dei dati simulati utilizzati per addestrare la rete.

La fase di inferenza vera e propria viene quindi eseguita invocando l'interprete configurato in precedenza:

```
TfLiteInterpreterInvoke(interp_);
```

Analizzando una campione alla volta, il modello produce in output un singolo valore scalato nel range $[-1, 1]$. Per avere un senso fisico, lo scalare viene poi convertito nello spazio dell'area dell'impulso tramite la formula inversa della normalizzazione MinMax:

$$y_{\text{orig}} = \left(\frac{y_{\text{pred}} + 1}{2} \right) (out_{\max} - out_{\min}) + out_{\min} \tag{5.1}$$

Il valore y_{orig} rappresenta l'integrale della waveform ricostruito (quantità da cui si ricava l'energia dell'evento) e verrà successivamente salvato nei file `DL2`.

Per sfruttare appieno l'architettura ARM Cortex-A78 della Jetson Orin Nano, il file di configurazione per la compilazione `CMakeLists.txt` è stato esteso per includere una serie di flag di ottimizzazione specifici all'hardware su cui il framework viene eseguito:

- `-mcpu=cortex-a78` per generare codice ottimizzato per la CPU della Jetson;
- `-ftree-vectorize` per abilitare la vettorizzazione automatica;
- `-funsafe-math-optimizations` per velocizzare le operazioni floating-point;
- ottimizzazione globale `-O2`.

La build rileva automaticamente la piattaforma, ad esempio distinguendo se RTA-DP viene compilato su di una Jetson oppure all'interno di un Docker container. Questo garantisce portabilità e, al tempo stesso, massime prestazioni sulla piattaforma edge:

```
if(HOST_SYSTEM_PROCESSOR MATCHES "aarch64")
  set(OPT_FLAGS -O2 -mcpu=cortex-a78 -ftree-vectorize -funsafe-math-
    optimizations)
else()
  set(OPT_FLAGS -O2 -march=x86-64 -mtune=generic)
endif()
```

5.3.3 Validazione dell'integrazione

Prima della generazione dei file `DL2`, la corretta integrazione del modello LiteRT nella pipeline di ricostruzione è stata verificata direttamente a video, analizzando l'output della predizione dell'area del worker di inferenza per singola waveform (vedi Figura 5.4). Per ogni evento, il sistema stampa una serie di informazioni diagnostiche, tra cui:

- `REAL AREA`: il valore reale dell'area associata alla waveform, utilizzato come ground truth (la label). Per facilitarne l'accesso in questa fase di test, tale valore è stato temporaneamente memorizzato nel campo `us` della struttura `Data_WaveHeader` lato producer;
- `Predicted model output (inverse-scaled area)`: l'area predetta dal modello dopo l'applicazione dell'inversa della trasformazione MinMax (vedi Formula 5.1), espressa nello stesso dominio fisico della ground truth;

- `Predicted model output (scaled area ([-1, 1]))`: la stessa predizione nel dominio normalizzato MinMax, utile per verificare la coerenza con quanto osservato nel notebook di addestramento;
- `Output scaling details`: i parametri `min_area_value`, `max_area_value` e il relativo fattore di scala, che permettono di controllare che la trasformazione inversa sia stata applicata con gli stessi estremi usati in fase di training;
- `Waveform value range`: i valori minimo, massimo e medio della waveform ricostruita, utilizzati per verificare che i campioni rientrino nel range $[in\_min, in\_max]$ atteso e non subiscano saturazioni indesiderate (il valore massimo equivale al picco della forma d'onda);
- `Total inference time`: il tempo impiegato dall'invocazione e dall'esecuzione di `TfLiteInterpreterInvoke` per processare la singola waveform.

```
[Supervisor1] REAL AREA: 96774
[Worker1] Predicted model output (inverse-scaled area): 75901.3
[Worker1] Predicted model output (scaled area ([-1, 1])): -0.01065
[Worker1] Output scaling details: min_area_value=3418.73, max_area_value=149944, scale=73262.9
[Worker1] Waveform value range: min=0, max=2506, avg=111.072
[Worker1] Total inference time: 0.000450656s

[Supervisor1] REAL AREA: 148265
[Worker1] Predicted model output (inverse-scaled area): 137150
[Worker1] Predicted model output (scaled area ([-1, 1])): 0.825357
[Worker1] Output scaling details: min_area_value=3418.73, max_area_value=149944, scale=73262.9
[Worker1] Waveform value range: min=0, max=3769, avg=136.722
[Worker1] Total inference time: 0.000451584s

[Supervisor1] REAL AREA: 94689
[Worker1] Predicted model output (inverse-scaled area): 74479.2
[Worker1] Predicted model output (scaled area ([-1, 1])): -0.0300613
[Worker1] Output scaling details: min_area_value=3418.73, max_area_value=149944, scale=73262.9
[Worker1] Waveform value range: min=0, max=2453, avg=110.011
[Worker1] Total inference time: 0.0004765445s

[Supervisor1] REAL AREA: 61342
[Worker1] Predicted model output (inverse-scaled area): 56738.6
[Worker1] Predicted model output (scaled area ([-1, 1])): -0.272212
[Worker1] Output scaling details: min_area_value=3418.73, max_area_value=149944, scale=73262.9
[Worker1] Waveform value range: min=0, max=1630, avg=93.2685
[Worker1] Total inference time: 0.000439104s

[Supervisor1] REAL AREA: 5442
[Worker1] Predicted model output (inverse-scaled area): 3655.06
[Worker1] Predicted model output (scaled area ([-1, 1])): -0.996774
[Worker1] Output scaling details: min_area_value=3418.73, max_area_value=149944, scale=73262.9
[Worker1] Waveform value range: min=0, max=265, avg=65.3515
[Worker1] Total inference time: 0.000478784s
```

Figura 5.4: Esempio di log a video dell'inferenza effettuata da `Worker1` su Jetson Orin Nano, utilizzando il modello di ML quantizzato `float16`.

Dall'analisi qualitativa dei log si osserva che, per waveform con area positiva e negativa nel dominio normalizzato, i valori di *scaled area* rientrano correttamente nell'intervallo $[-1, 1]$ e presentano il comportamento atteso: eventi con area più grande generano valori prossimi agli estremi dell'intervallo, mentre eventi di energia più bassa cadono in regioni intermedie. L'*inverse-scaled area* risulta numericamente coerente con la ground truth: in diversi esempi l'area predetta differisce dalla reale solo di una frazione rispetto alla scala complessiva del segnale, con una tendenza a leggere sottostime o sovrastime che riproduce il comportamento già osservato per la valutazione del modello `float16`. Nello stesso log, il range dei valori di waveform (minimo, massimo, media) risulta compatibile con il dominio dei dati di training: i massimi sono compresi entro poche migliaia di conteggi e i minimi restano prossimi allo zero. Ciò conferma che l'operazione di unpacking

dei campioni da `uint32_t` a `float` e la successiva normalizzazione mediante lo stesso MinMax utilizzato in fase di training sono state implementate correttamente.

Le informazioni di scaling stampate a video (`min_area_value`, `max_area_value` e fattore di scala) coincidono con i parametri per il preprocessing delle label utilizzati nel notebook Python, garantendo che la trasformazione inversa sia applicata in modo coerente sia offline sia durante l'inferenza embedded. In questo modo, il confronto fra `REAL AREA` e `Predicted model output (inverse-scaled area)` avviene nello stesso sistema di unità e permette di verificare direttamente la bontà del porting del modello e i risultati dell'inferenza su valori non visti durante il training.

Infine, il campo `Total inference time` mostra che il tempo di inferenza per singola waveform è dell'ordine delle frazioni di millisecondo, in linea con quanto atteso per un modello di piccole dimensioni eseguito su CPU ARM con delegate XNNPACK. Sebbene l'analisi quantitativa dettagliata di latenza, throughput e consumo energetico sia rimandata al Capitolo 6, questa fase di validazione ha permesso di confermare:

- l'allineamento dei valori dell'area predetta con inferenza in real-time comparata con i valori di ground truth associati;
- il corretto unpacking e la corretta normalizzazione dei dati in ingresso;
- l'uso consistente dei parametri di scaling tra il notebook di training Python e implementazione C++;
- il corretto funzionamento del delegate XNNPACK su Jetson Orin Nano;
- la coerenza numerica tra le predizioni ottenute in ambiente Python e quelle prodotte in tempo reale dalla pipeline.

5.4 Scrittura dei file DL2 in output

L'ultimo stadio della pipeline di ricostruzione consiste nella generazione dei file `DL2` (vedi Sezione 2.3.3), che contengono la lista degli eventi ricostruiti a partire dalle waveform grezze contenute nei pacchetti `R0`. In questo passo, le aree degli impulsi predette dal modello quantizzato LiteRT vengono serializzate in un file con formato `HDF5` contenente una tabella di eventi. L'obiettivo è produrre una "*photon list*[4]" adatta alle analisi scientifiche successive del progetto GammaSky. In questo modo si chiude il flusso dati inaugurato dal producer: le waveform acquisite dal DAM vengono inviate in formato grezzo R0 in streaming al consumer, analizzate dal modello di ML integrato nei worker e infine scritte come tabella di eventi in un file DL2.

Per rendere configurabile la struttura dei file DL2, essa non è definita nel codice ma è stato introdotto un descrittore esterno in formato XML, `DL2model.xml`, il cui parsing avviene un'unica volta all'avvio del worker tramite la libreria `tinyxml2` e che definisce struttura e tipi dei campi della tabella di eventi. Il file descrive un dataset composto (`compound type`) memorizzato nel gruppo `/dl2/eventlist`, i cui campi principali sono: `n_waveform`, `tstart`, `integral1` e altri campi riservati a estensioni future (come altezza del picco, integrali secondari per waveform con più di un picco, temperatura del rivelatore, ecc.), i quali vengono inizializzati con un valore di placeholder (ad esempio `-1.0`) . Di seguito un estratto del modello XML:

```
<dataset name="eventlist" type="compound">
  <fields>
    <field name="n_waveform" type="float32"/>
    <field name="tstart"     type="float64"/>
    <field name="integral1"  type="float32"/>
    <field name="integral2"  type="float32"/>
    <field name="integral3"  type="float32"/>
```

[4]Una *photon list* è una rappresentazione basata su eventi individuali, in cui ciascun fotone rilevato è descritto da parametri fisici ricostruiti, come tempo di arrivo, energia e direzione di provenienza. Questo formato, tipico dell'analisi dei rivelatori per raggi gamma, conserva il massimo contenuto informativo e permette di svolgere successivamente analisi scientifiche flessibili (spettrali, temporali e di imaging) senza perdita di dati.

```
    <field name="halflife"   type="float32"/>
    <field name="temp"       type="float32"/>
  </fields>
</dataset>
```

Ogni evento ricostruito è rappresentato in memoria da una struttura C++ che rispecchia lo schema XML:

```
struct GFRow {
  float n_waveform;
  float mult;
  float tstart;
  float index_peak;
  float peak;
  float integral1;
  float integral2;
  float integral3;
  float halflife;
  float temp;
};
```

In questo modo l'aggiunta o la modifica di un campo nello schema DL2 richiede solo l'aggiornamento del file XML (e, se necessario, della struttura `GFRow`), mantenendo la logica di scrittura generica e riusabile.

L'inizializzazione del modello DL2 viene eseguita una sola volta e il risultato viene condiviso tra tutti i worker. Per evitare race conditions, il blocco di inizializzazione è protetto da uno `std::mutex`, così che solo il primo thread effettui il parsing del file XML, mentre gli altri riutilizzano la struttura già creata, anche per ridurre l'overhead. Durante l'esecuzione, il worker popola un vettore `std::vector<GFRow>` condiviso tra thread, all'interno del quale inserisce una riga per ciascuna waveform elaborata. In questa prima implementazione solo alcuni campi vengono compilati: `n_waveform` assume l'indice progressivo della waveform nel batch, `tstart` è estratto dall'header del pacchetto R0 ricevuto (timestamp del campione), mentre `integral1` contiene l'area predetta dal modello dopo la riconversione dall'intervallo normalizzato $[-1, 1]$ allo spazio delle aree.

La generazione dei dati DL2 è integrata all'interno del metodo `processData()` di

`Worker1` (lo stesso in cui viene eseguita l'inferenza). Dopo l'inferenza sul modello LiteRT, per ciascuna waveform viene creato un record `GFRow`:

```
GFRow row;
row.n_waveform = dl2_data.size() + 1;   // Indice progressivo
row.mult       = -1.0f;
row.tstart     = static_cast<float>(packet->body.w.ts.tv_sec);
row.index_peak = -1.0f;
row.peak       = -1.0f;
row.integral1  = y_orig;   // L'area predetta (inverse MinMax)
row.integral2  = -1.0f;
...

dl2_data.push_back(row);
```

I record vengono accumulati in un vettore condiviso `dl2_data`. Quando il numero di eventi raggiunge una soglia configurabile (di default 1000 righe per file), il batch viene scritto su file e salvato su disco in formato HDF5 tramite la funzione `write_dl2_file()` (vedi Figura 5.5 per il contenuto di un file DL2). Quest'ultima costruisce il tipo composto HDF5 (`H5::CompType`) iterando sui campi definiti nel modello XML e inserendo i relativi membri con i corrispondenti offset nella struttura `GFRow`. La mappatura tra modello e struttura è attualmente basata su un insieme di controlli espliciti sui nomi dei campi, soluzione semplice ma sufficiente per garantire la coerenza tra il modello DL2 e il dataset HDF5 prodotto:

```
void Worker1::write_dl2_file(){
    H5::CompType mtype(sizeof(GFRow));

    for (const auto& field : model.fields) {
        if (field.name == "n_waveform")
            mtype.insertMember(field.name, HOFFSET(GFRow, n_waveform), H5::
    PredType::NATIVE_FLOAT);
        else if (field.name == "tstart")
            mtype.insertMember(field.name, HOFFSET(GFRow, tstart), H5::
    PredType::NATIVE_FLOAT);
        else if (field.name == "integral1")
            mtype.insertMember(field.name, HOFFSET(GFRow, integral1), H5::
    PredType::NATIVE_FLOAT);
        ...
```

```
12      }
13
14      H5::DataSet dataset = group.createDataSet(model.datasetName, mtype,
        dataspace);
15      dataset.write(data.data(), mtype);
16  }
17
18  std::vector<uint8_t> Worker1::processData(){
19      ...
20      if (dl2_data.size() >= 1000) {
21          std::string filename = getOutputPath() + "/dl2_output_batch" + std
        ::to_string(batch_counter++) + ".dl2.h5";
22
23          write_dl2_file(dl2_model, filename, dl2_data);
24          dl2_data.clear();
25      }
26      ...
27  }
```

In questo modo i batch di eventi accumulati in memoria vengono serializzati correttamente su file secondo il formato previsto. Completata la scrittura, il buffer viene svuotato e il worker riprende il flusso di elaborazione. I file prodotti vengono salvati in una directory di output configurabile tramite variabile d'ambiente e seguono una convenzione di naming del tipo:

`dl2_output_batchXXXX.dl2.h5`

	0									
	n_waveform	mult	tstart	index_peak	peak	integral1	integral2	integral3	halflife	temp
0	1.0	-1.0	2.350513...	-1.0	-1.0	89435.164	-1.0	-1.0	-1.0	-1.0
1	2.0	-1.0	2.407168...	-1.0	-1.0	28031.668	-1.0	-1.0	-1.0	-1.0
2	3.0	-1.0	2.459933...	-1.0	-1.0	50514.24	-1.0	-1.0	-1.0	-1.0
3	4.0	-1.0	2.512088...	-1.0	-1.0	35279.59	-1.0	-1.0	-1.0	-1.0
4	5.0	-1.0	2.565150...	-1.0	-1.0	59697.215	-1.0	-1.0	-1.0	-1.0
5	6.0	-1.0	2.618889...	-1.0	-1.0	18603.414	-1.0	-1.0	-1.0	-1.0
6	7.0	-1.0	2.673304...	-1.0	-1.0	124703.76	-1.0	-1.0	-1.0	-1.0
7	8.0	-1.0	2.727513...	-1.0	-1.0	21968.213	-1.0	-1.0	-1.0	-1.0
8	9.0	-1.0	2.783044...	-1.0	-1.0	53090.973	-1.0	-1.0	-1.0	-1.0
9	10.0	-1.0	2.838385...	-1.0	-1.0	140921.92	-1.0	-1.0	-1.0	-1.0
10	11.0	-1.0	2.89881...	-1.0	-1.0	101538.9...	-1.0	-1.0	-1.0	-1.0
11	12.0	-1.0	2.956197...	-1.0	-1.0	44387.258	-1.0	-1.0	-1.0	-1.0
12	13.0	-1.0	3.014388...	-1.0	-1.0	65987.08	-1.0	-1.0	-1.0	-1.0

Figura 5.5: Esempio di file DL2 prodotto dopo aver processato 1000 waveform. Il campo `integral1` contiene il valore dell'area ricostruita tramite ML.

La scrittura a batch di dimensione configurabile consente di controllare il trade-off tra:

- numero di file generati, in modo da limitare al minimo l'occupazione di memoria non volatile in dispositivi a risorse limitate;
- latenza di scrittura su disco, riducendo l'overhead di I/O su sistemi embedded;
- efficienza del canale di downlink (in scenari space-based).

Capitolo 6

Testing e analisi delle prestazioni della pipeline di GammaSky

Quest'ultimo capitolo descrive la fase di testing e valutazione della pipeline di GammaSky, dopo l'integrazione del modello di machine learning ottimizzato e della logica di scrittura dei file `DL2`. L'obiettivo della sperimentazione è verificare la capacità del sistema di sostenere un'elaborazione in tempo reale su piattaforma edge `NVIDIA Jetson Orin Nano`, in vista del futuro deployment all'interno del setup sperimentale di GammaSky presso l'Osservatorio sul Monte Cimone.

La campagna di test si è articolata in due direzioni principali. Da un lato, sono state misurate le prestazioni dell'inferenza del modello neurale su flussi continui di waveform, valutando latenza media, throughput e utilizzo di memoria, sia nel caso del modello originale `float32` sia nella variante `float16`. Dall'altro, è stato monitorato il comportamento delle risorse hardware del dispositivo (CPU, RAM, GPU e potenza assorbita), in condizioni sia di inattività sia di massimo carico, tramite il tool `jtop`. In aggiunta alla misurazione delle prestazioni, sono stati condotti test di integrazione end-to-end per verificare la correttezza funzionale dell'intero flusso di elaborazione, dall'invio dei pacchetti `R0` fino alla generazione dei corrispondenti file `DL2`, passando per la fase di

inferenza. Sono stati sviluppati appositi script di integrazione per automatizzare l'esecuzione dei test, permettendo di riprodurre condizioni di streaming a pieno carico (fino a 200 $waveform/s$) e di validare la robustezza della pipeline nel lungo periodo. I risultati ottenuti forniscono indicazioni utili sulle effettive capacità della piattaforma di eseguire ricostruzione in tempo reale nel contesto dell'analisi dei TGF e rappresentano un passaggio fondamentale verso l'adozione operativa del sistema.

6.1 Benchmark di inferenza del modello ML su Jetson Orin Nano

Per valutare le prestazioni del modello di ricostruzione dell'area della waveform, è stata eseguita una fase di benchmark sull'edge computer Jetson, utilizzando il worker adibito all'inferenza descritto nella Sezione 5.3. L'analisi si è concentrata su tre parametri principali:

- **tempo medio di inferenza** per singola waveform;
- **rate (throughput) di inferenza** espressa in Hz (la frequenza);
- **utilizzo massimo di memoria RAM** durante il processamento.

Per questa fase di confronto tra modelli, si è deciso di prendere in considerazione quello originale in formato `float32` non ottimizzato e quello quantizzato mediante *post-training quantization* in formato `float16`. Per entrambi i modelli sono state processate sequenze di 10 000 waveform consecutive, corrispondenti a circa 10 batch di scrittura di `DL2`, con un rate di input generato dal producer (in questo caso lo streamer-simulatore `gfse.py`) pari a circa 200 eventi/s, così da stressare il sistema in condizioni superiori ai requisiti previsti dal caso d'uso.

6.1.1 Metodologia di misura

La latenza di inferenza è stata misurata direttamente all'interno del worker tramite `clock_gettime()` basato sul timer `CLOCK_MONOTONIC_RAW`, calcolando la differenza temporale, come mostrato nello snippet seguente estratto da `Worker1.cpp`:

```
clock_gettime(CLOCK_MONOTONIC_RAW, &start);
TfLiteInterpreterInvoke(interp_);   // Inferenza
clock_gettime(CLOCK_MONOTONIC_RAW, &end);

double inference_time = timespec_diff(&start, &end);
```

Il consumo massimo di memoria durante la fase di inferenza è stato monitorato tramite la funzione `getrusage()`, aggiornando un contatore:

```
int current_memory = getMemoryUsage();
int previous_peak = peak_memory_kb.load();

while (current_memory > previous_peak) {
    if (peak_memory_kb.compare_exchange_weak(previous_peak, current_memory)
    ) {
        break;
    }
}
```

Le statistiche vengono accumulate in variabili atomiche condivise tra i worker e stampate periodicamente dopo un intervallo fissato di waveform elaborate (di default ogni 10 000 eventi):

```
[Worker1] ===== INFERENCE STATISTICS =====
[Worker1] Processed N waveforms
[Worker1] Average inference time: X.XXXXXXXXs
[Worker1] Inference rate: YYYYY.YY Hz
[Worker1] Peak memory usage: ZZZZZ KB
```

6.1.2 Risultati sperimentali

Una media dei risultati ottenuta su varie esecuzioni del sistema è riportata nella Tabella 6.1. Come evidenziato, le prestazioni dei due modelli risultano estremamente simili.

Tabella 6.1: Prestazioni di inferenza su 10 000 waveform.

Modello	**Avg. time**	**Rate**	**Peak RAM**
float32 (117 KB)	0.000451 s	2216 Hz	16996 KB
float16 quant. (70 KB)	0.000446 s	2242 Hz	17956 KB

Come si può osservare, la quantizzazione a `float16` non modifica in modo significativo il tempo di inferenza medio rispetto alla versione `float32`. Ciò è coerente con le seguenti considerazioni tecniche:

1. **Modello compatto**: la rete utilizza circa 25 000 parametri, rendendo marginale il peso computazionale delle operazioni a singola precisione (`float32`).

2. **Delegate XNNPACK e accelerazione hardware ARM NEON**: entrambe le rappresentazioni numeriche beneficiano degli stessi kernel ottimizzati per CPU ARM, con l'architettura Jetson che supporta calcolo vettoriale efficiente sia per FP16 che FP32.

3. **Overhead di LiteRT**: per modelli compatti l'orchestrazione del runtime (allocazione tensori, copy buffer, scheduling) è comparabile al tempo di calcolo.

4. **Batch unitario**: l'inferenza avviene "sample-by-sample", limitando i benefici derivanti dalla minore occupazione in memoria dei tensori FP16.

La quantizzazione a `float16` fornisce tuttavia un vantaggio rilevante in termini di footprint del modello (riduzione di circa il 40%), mantenendo risultati di inferenza equivalenti al modello originale. Come tecnica, può pertanto essere adottata in modo preferenziale nel deployment a bordo di dispositivi di edge computing come per il caso di GammaSky. Il test conferma che la piattaforma è in grado di sostenere un'elaborazione continua con throughput superiore a 2 200 eventi/s, valore di gran lunga superiore al requisito operativo della sorgente dati ($\simeq 200\ waveform/s$).

6.2 Utilizzo delle risorse hardware

In questa fase sono state analizzate le risorse hardware utilizzate dalla `NVIDIA Jetson Orin Nano` durante l'esecuzione della pipeline RTA-DP, al fine di verificarne l'efficienza energetica e la compatibilità con scenari operativi caratterizzati da vincoli stringenti (come piattaforme embedded o potenziali missioni CubeSat). Il monitoraggio è stato effettuato tramite lo strumento `jtop`, che consente di osservare in tempo reale il consumo di CPU, memoria, GPU, temperatura e potenza del sistema. Per simulare un ambiente "resource-constrained", la Jetson è stata impostata nella modalità di potenza minima (`NV Power [0]`, 15 W massimi), che limita il budget energetico disponibile al SoC.
Il confronto è stato condotto in due condizioni distinte:

1. **Jetson in idle**: nessun processo della pipeline attivo, con tutti i componenti di RTA-DP disattivati e con `ProcessMonitoring.py` non in esecuzione in background.

2. **Pipeline a pieno carico**: esecuzione contemporanea di parsing dei pacchetti, inferenza, generazione dei file DL2 e monitoraggio in background. Per stressare il sistema è stato utilizzato lo stesso scenario di test della Sezione 6.1, con il producer configurato per inviare fino a 200 $waveform/s$, un valore superiore ai requisiti previsti dal caso d'uso GammaSky.

La Tabella 6.2 riassume l'utilizzo delle principali risorse hardware nei due scenari.

Tabella 6.2: Confronto dell'utilizzo delle risorse hardware della Jetson Orin Nano in idle e durante l'esecuzione della pipeline completa.

Risorsa	**Idle**	**Full load**
CPU (RTA-DP utilizza solo 1 core su 6)	1–8%	40–55%
RAM utilizzata	∼750–800 MB	∼790–870 MB
GPU utilizzata	0%	0%
Potenza media (`VDD_IN`)	4.8 W	5.0 W
CPU Clock	729 MHz	729 MHz
Temperatura CPU	∼46.2°C	∼46.5°C

Consumo della CPU

L'utilizzo della CPU rappresenta uno dei parametri più critici in un sistema real-time su hardware embedded. Questo discorso è ancor più valido per RTA-DP, visto che tutte le operazioni principali (inferenza inclusa) vengono eseguite su CPU. L'analisi condotta tramite `jtop` evidenzia come:

- **in idle**, tutti e sei i core ARM Cortex-A78 operano in modalità alternata con un utilizzo compreso tra l'1% e l'8%, senza alcun processo e thread della pipeline in esecuzione;
- **con la pipeline attiva**, un solo core viene utilizzato in modo significativo (fino al 50–55% nei picchi), mentre gli altri rimangono pressoché inattivi.

Si osserva quindi che l'intero framework RTA-DP, incluso il processamento dei pacchetti, l'inferenza e la scrittura dei file DL2, riesce a funzionare sfruttando in modo limitato un singolo core. La CPU rimane dunque largamente sotto-utilizzata, anche durante lo stress test. Questo risultato è una conferma diretta dell'efficacia delle ottimizzazioni introdotte nella Sezione 4.6:

- eliminazione dei cicli di *busy waiting*;

- utilizzo di `sleep` nei cicli di esecuzione dei thread;
- riorganizzazione dei buffer e delle code per ridurre le copie;
- logging configurabile e disattivabile via `spdlog`.

Consumo di memoria RAM

In condizioni di inattività, la Jetson utilizza tra 750 MB e 800 MB di RAM, un valore in linea con quanto atteso per il sistema operativo e i processi di sistema. Durante l'esecuzione completa della pipeline, l'incremento del consumo di memoria risulta molto limitato: la RAM utilizzata aumenta infatti solo di alcune decine di megabyte, con un picco di circa 870 MB attribuibile ai principali task eseguiti dal DataProcessor.

Questo comportamento, come già visto in parte nella Sezione 6.1.2, è dovuto principalmente a:

- la dimensione ridotta del modello di machine learning e la sua versione `float16`;
- l'utilizzo di `LiteRT` con delegate `XNNPack`;
- le ottimizzazioni introdotte nella Sezione 4.6 che hanno ridotto allocazioni ridondanti, copie di buffer e overhead di logging.

Utilizzo della GPU e consumo energetico

La GPU della Jetson Orin Nano non viene mai utilizzata né in idle né a pieno carico. Questo è atteso, poiché l'esecuzione dell'inferenza avviene tramite il delegate `XNNPACK`, basato su ottimizzazioni SIMD NEON della CPU. L'assenza di carico GPU riduce drasticamente i consumi energetici, evita picchi termici e massimizza l'efficienza per sistemi power-limited.

Per quanto riguarda l'assorbimento energetico, la Jetson configurata in modalità 15 W presenta:

- **Idle**: potenza media (CPU e GPU) ∼0.58 W e ∼1.4 W (SoC), con potenza totale di ingresso (`VDD_IN`) pari a ∼4.8 W.
- **Pipeline attiva**: valori quasi identici, senza variazioni significative durante l'inferenza (CPU e GPU ∼0.68 W, SoC ∼1.4 W) e potenza di ingresso media stabile a 5.0 W.

La pipeline, anche a pieno carico, raggiunge massimo un terzo del budget energetico disponibile (15 W), dimostrando un'efficienza più che adeguata per applicazioni edge e missioni con severe limitazioni energetiche. Per contestualizzare tali consumi è utile un riferimento ai budget di potenza tipici per piattaforme CubeSat[51], così da valutare un'eventuale compatibilità del sistema con tali scenari spaziali:

- un **1U** dispone tipicamente di ∼1–2,5 W;
- un **2U** può fornire ∼2–5 W;
- un **3U** raggiunge valori dell'ordine di ∼7–20 W;
- un **6U**, in funzione della configurazione dei pannelli solari e dell'area effettivamente illuminata, può raggiungere alcune decine di watt (tipicamente ∼20–50 W).

Una Jetson Orin Nano operante attorno i 5 W di picco, insieme al resto del setup sperimentale GammaSky (SiPM e RedPitaya con modulo GPS, i quali consumano attorno 10 W), rientra nei limiti tipici di un CubeSat 3U–6U, possibilmente rendendo l'intero payload compatibile con missioni satellitari o comunque ambienti edge estremamente vincolati.

6.3 Test di integrazione dell'intera pipeline

Per verificare il corretto funzionamento nel suo insieme della pipeline RTA-DP applicata al caso d'uso GammaSky, è stato sviluppato un test di integrazione end-to-end (*system*

integration test) automatizzato. Questo test è stato concepito per assicurarsi che tutti i componenti del sistema cooperino correttamente quando avviati, sincronizzati, messi sotto carico e infine chiusi in modo ordinato, il tutto eseguito all'interno di un ambiente controllato basato su Docker container.

A differenza degli *unit test*, che validano moduli isolati tramite dipendenze simulate o input sintetici, un integration test esercita l'applicazione come un sistema completo, verificando comunicazioni reali (ZeroMQ), lettura e scrittura su file system (DL0 $\rightarrow$ DL2), gestione dei processi, scambio di segnali, corretta inizializzazione dei thread e shutdown ordinato. In un contesto come RTA-DP, caratterizzato da un flusso continuo di dati, interazioni concorrenti e componenti eterogenei scritti in linguaggi differenti, gli unit test non bastano a rilevare errori di orchestrazione, race conditions, problemi di temporizzazione o mancate sincronizzazioni tra i moduli. La correttezza del sistema dipende infatti non solo dalla logica interna del codice, ma dalla sequenza temporale e dal coordinamento dei processi. Gli integration test sono risultati dunque necessari per validare il comportamento end-to-end dell'intero framework.

Il test sviluppato verifica la sequenza completa del flusso operativo, avviando in sequenza e in modo controllato:

1. `ProcessMonitoring.py`, il servizio di monitoraggio dei processi;
2. `gfse.py`, che simula il DAM fungendo da producer ed inviando waveform in formato `R0` contenute in pacchetti `DL0` tramite socket ZeroMQ;
3. il consumer C++ (`ProcessDataConsumer1`), che esegue il pre-processing dei dati, inferenza ML e scrittura dei file `DL2`;
4. il modulo di controllo dei comandi, che avvia il processamento tramite il messaggio `start all`.

Questo flusso rappresenta un vero test end-to-end. Verifica che i componenti della pipeline si inizializzino correttamente, comunichino tra loro e rimangano operativi per un

intervallo di tempo significativo, simulando un utilizzo reale. La validazione finale si basa sul monitoraggio dello stato dei processi: se uno di essi termina inaspettatamente, il test fallisce, segnalando un possibile problema di configurazione, concorrenza o comunicazione interna alla pipeline. Possiamo dire che l'esecuzione adotta un approccio *black-box*. Il test non valuta infatti la correttezza degli algoritmi interni, ma osserva il comportamento complessivo del sistema come un unico blocco per un periodo configurabile (con un timeout predefinito di 200 s). Lo script gestisce inoltre l'intero ciclo di vita dei processi, incluso lo shutdown sequenziale automatico al termine del tempo di esecuzione, inviando i relativi segnali UNIX. Questo garantisce la corretta liberazione delle risorse ed evita situazioni indesiderate come processi zombie, porte occupate o code lasciate in stato incoerente.

Il cuore del test è contenuto nello script `test_integration_rtadp1.py`, che utilizza il framework Python `unittest` per avviare e monitorare l'intera pipeline di GammaSky all'interno di un ambiente Docker isolato. Ogni componente viene lanciato sequenzialmente tramite `subprocess` in un proprio gruppo di processi (`os.setsid`), così da permetterne una gestione indipendente e un successivo shutdown controllato. La logica utilizzata per l'avvio dei processi è generica e modulare: ogni componente è incapsulato in una funzione dedicata che registra l'handle del processo in una lista condivisa per consentire la gestione coordinata delle risorse. Lo snippet seguente mostra un esempio tipico di questa struttura:

```
def run_consumer(self):
    cmd = ['./ProcessDataConsumer1', self.rtaconfig]
    process = subprocess.Popen(
        cmd,
        cwd=str(CPP_DIR / 'build'),
        stdout=None, stderr=None,
        preexec_fn=os.setsid
    )
    self.processes.append(process)
    return process
```

L'inizializzazione completa della pipeline avviene nel metodo `test_full_integration()`, che rappresenta il flusso di avvio reale del sistema. Il test attende esplicitamente intervalli temporali tra un componente e il successivo per garantire che ogni modulo completi correttamente la propria fase di bootstrap (lettura configurazione, creazione socket, inizializzazione dei worker, spawn di thread interni, ecc.). Una volta avviati tutti gli elementi, viene inviato il comando globale `start all`, che attiva il flusso di elaborazione dei dati:

```
def test_full_integration(self):
    # 1. Avvio del monitoraggio
    monitoring_process = self.run_process_monitoring()
    time.sleep(2)

    # 2. Avvio del simulatore DAMS (gfse)
    simulator_process = self.run_dams_simulator(
        addr='127.0.0.1', port=1234,
        indir=str(TEST_DIR / 'dl0_simulated'),
        rpid=1, wform_sec=200
    )
    time.sleep(6)

    # 3. Avvio del consumer C++
    consumer_process = self.run_consumer()
    time.sleep(3)

    # 4. Invio del comando START a tutti i componenti
    start_process = self.send_start_command()

    # 5. Tempo di esecuzione utile del sistema
    time.sleep(200)

    # 6. Verifica se i processi siano ancora attivi
    self.assertEqual(monitoring_process.poll(), None)
    self.assertEqual(consumer_process.poll(), None)
    self.assertEqual(simulator_process.poll(), None)

    logger.info('Integration test completed successfully')
```

Una caratteristica rilevante del test è la procedura di chiusura controllata (*graceful shutdown*), che segue un approccio graduale garantendo la terminazione sicura e ordinata

dei processi secondo una gerarchia prestabilita:

1. prima il consumer C++, che gestisce code e worker multipli e che necessita di tempo aggiuntivo per completare eventuali chiusure di socket aperti o thread;
2. successivamente il producer-simulatore `gfse.py`;
3. infine il modulo di monitoraggio.

Il tutto è protetto da un sistema di intercettazione dei segnali di terminazione: prima uno shutdown "gentile" tramite `SIGINT`, lasciando al processo la possibilità di chiudere risorse e svuotare buffer ed eventuali code interne. Se il processo non risponde entro un tempo limite, viene inviato `SIGTERM`. Solo come ultima risorsa si usa `SIGKILL`, assicurando la chiusura totale del gruppo di processi (grazie all'utilizzo di `os.setsid` e `os.killpg`). Questo meccanismo evita la creazione di processi zombie o porte rimaste aperte, garantendo che ogni esecuzione successiva del test parta da un ambiente pulito. Parte della logica di shutdown via segnali UNIX è riportata di seguito:

```
def terminate_process(process, name, timeout=3):
    if not process or process.poll() is not None:
        return
    try:
        os.killpg(process.pid, signal.SIGINT)
        start = time.time()
        while time.time() - start < timeout:
            if process.poll() is not None:
                return
            time.sleep(0.1)

        # Escalation a SIGTERM
        process.terminate()
        start = time.time()
        while time.time() - start < 2:
            if process.poll() is not None:
                return
            time.sleep(0.1)

        # Ultima risorsa: SIGKILL
```

```
        process.kill()
        process.wait(timeout=1)

    except Exception:
        # Kill forzato in caso di errori imprevisti
        try:
            if process.poll() is None:
                process.kill()
                process.wait(timeout=1)
        except:
            pass
```

Una volta definita la procedura di terminazione singola, il test applica un ordine di shutdown rigoroso, seguendo le dipendenze reali del sistema:

```
# Step 1: Ferma prima il consumer con un timeout esteso
if consumer_process:
    terminate_process(consumer_process, "Consumer", timeout=15)
    time.sleep(5)

# Step 2: Ferma il DAMS simulator
if simulator_process:
    terminate_process(simulator_process, "DAMS Simulator")
    time.sleep(2)

# Step 3: Ferma il monitoring process
if monitoring_process:
    terminate_process(monitoring_process, "Process Monitoring")
    time.sleep(1)

# Step 4: Ferma eventuali processi rimanenti
for process in other_processes:
    if process and process.poll() is None:
        terminate_process(process, f"Other Process (PID: {process.pid})")

logger.info("Sequential cleanup completed")
```

L'utilizzo del container Docker migliora significativamente la riproducibilità dell'ambiente di test, poiché isola le dipendenze software e rende la pipeline eseguibile in modo coerente su diverse macchine di sviluppo. Pur non astraendo completamente dalle specificità hardware, l'approccio containerizzato facilita la validazione anche su dispositivi edge

che supportano questa tecnologia, riducendo la variabilità dovuta alla configurazione del sistema operativo e delle librerie.

Conclusioni e sviluppi futuri

In questo progetto di tesi sono state presentate e discusse le attività svolte durante il tirocinio presso l'Osservatorio di Astrofisica e Scienza dello Spazio di Bologna (OAS), parte dell'Istituto Nazionale di Astrofisica (INAF). Il lavoro svolto nel corso di quei mesi, ha portato alla progettazione, implementazione, ottimizzazione e validazione di una pipeline completa per l'analisi in tempo reale di eventi gamma atmosferici in ambiente embedded, basata sul framework RTA-DP e applicata al caso d'uso sperimentale GammaSky. Il percorso affrontato ha abbracciato aspetti di architettura software, gestione concorrente, ottimizzazione delle prestazioni, integrazione tra componenti eterogenei e machine learning su dispositivi a risorse limitate, con l'obiettivo di ottenere un sistema stabile, efficiente e adatto a contesti operativi con vincoli stringenti.

La prima fase del lavoro ha riguardato la stabilizzazione del framework C++, che nella sua versione iniziale presentava race conditions, accessi concorrenti non controllati, deadlock, terminazioni non pulite del sistema e uso inefficiente delle risorse hardware. Le problematiche iniziali sono state risolte attraverso un'approfondita revisione della gestione dei thread, dei meccanismi di sincronizzazione e della comunicazione tramite ZeroMQ. La riscrittura delle code interne tramite strutture *thread-safe*, la riorganizzazione del ciclo di vita dei thread e del contesto ZeroMQ e la riorganizzazione completa della fase di terminazione hanno permesso di trasformare un abbozzo preliminare in un sistema affidabile, capace di sostenere carichi elevati in streaming continuo. Le ottimizzazioni introdotte, unite alla riduzione delle copie di memoria e all'eliminazione dei cicli di busy

waiting, hanno portato a una drastica riduzione del carico della CPU, migliorando la stabilità dell'intero sistema e rendendolo più prevedibile anche in condizioni di esecuzione prolungata. La possibilità di disattivare completamente il sistema di logging tramite configurazione di compilazione ha permesso di ridurre ulteriormente l'overhead computazionale, ottimizzando il framework RTA-DP per un utilizzo su dispositivi a risorse limitate.

La seconda parte del progetto ha introdotto la creazione di una pipeline di ricostruzione per il progetto GammaSky, basata su RTA-DP. Il lavoro ha richiesto lo sviluppo di un worker che integra un modello di ML per la ricostruzione dell'area delle waveform direttamente a bordo dell'edge computer NVIDIA Jetson Orin Nano. Dopo una fase dedicata all'addestramento del modello neurale e alla sua ottimizzazione tramite tecniche di quantizzazione, il formato `float16` si è dimostrato il più adatto a un'implementazione in ambiente embedded, grazie al suo ottimo compromesso tra performance, dimensioni ridotte e velocità di inferenza. L'intero flusso di inferenza è stato adattato per funzionare in streaming, in modo da elaborare ogni waveform non appena disponibile, mantenendo una latenza minima e un throughput compatibile con i requisiti sperimentali. Parallelamente, è stata implementata la logica di generazione dei file `DL2` utilizzando il formato `HDF5` e un modello dati definito tramite file `XML`. Questa componente ha reso possibile la creazione di tabelle eventi ordinate, nelle quali ogni waveform elaborata viene trasformata in un evento fisico ricostruito, completo del valore di area predetta dal modello e del tempo di arrivo del flash gamma. Il workflow risultante copre così l'intero flusso `R0` $\rightarrow$ inferenza $\rightarrow$ `DL2`, rendendo la pipeline operativa e pronta per essere utilizzata in un contesto sperimentale reale.

La fase finale del lavoro ha riguardato l'analisi delle prestazioni e il testing del sistema. I benchmark condotti sulla Jetson Orin Nano hanno mostrato che sia il modello originale in `float32` sia la versione in `float16` garantiscono tempi di inferenza dell'ordine di pochi centinaia di microsecondi per waveform, permettendo di raggiungere una frequenza di

elaborazione superiore ai requisiti del caso d'uso. L'utilizzo di CPU e RAM è risultato estremamente contenuto, con un incremento trascurabile rispetto allo stato di inattività del dispositivo e un consumo energetico stabile che non supera i 5 W anche durante stress test prolungati.

Infine, è stato sviluppato un test di integrazione end-to-end che permette di validare il funzionamento dell'intero sistema all'interno di un ambiente Docker. Questo processo ha verificato il comportamento congiunto di tutti i componenti: monitoraggio, streaming di waveform, DataProcessor C++, inferenza, generazione dei file `DL2` e chiusura ordinata del sistema. Questa fase ha rappresentato la validazione finale del lavoro svolto, dimostrando la capacità della pipeline di operare in modo stabile e coordinato lungo tutto il ciclo di vita, condizione essenziale per la successiva integrazione nel setup sperimentale completo di GammaSky.

In conclusione, il progetto ha portato alla realizzazione di una pipeline completa e funzionante per la ricostruzione in tempo reale di eventi gamma atmosferici su edge device mediante tecniche di ML, dimostrando l'efficacia dell'approccio adottato e aprendo a una serie di sviluppi futuri che potranno arricchire il valore scientifico del sistema e ampliare gli scenari applicativi del framework RTA-DP.

Sviluppi futuri

Il lavoro svolto costituisce una base solida su cui costruire le evoluzioni future del progetto GammaSky. Il primo passo consisterà nell'integrazione della pipeline nel setup installato presso l'Osservatorio sul Monte Cimone, integrandola con l'apparato sperimentale completo composto dal rivelatore SiPM, dalla Red Pitaya dotata di modulo GPS e dalla Jetson. Questa fase permetterà di validare sul campo le prestazioni del sistema e di verificarne la robustezza in condizioni operative reali.

Un'evoluzione fondamentale del progetto riguarda il potenziamento degli algoritmi di

ricostruzione. La rete neurale attualmente impiegata fornisce una stima dell'integrale della waveform, ma non offre un vantaggio sostanziale rispetto a una semplice integrazione numerica: il suo ruolo principale è stato quello di dimostrare la fattibilità dell'esecuzione di modelli di ML all'interno della pipeline real–time. Il passo successivo consiste quindi nello sviluppo di tecniche di ricostruzione più avanzate, basate su modelli di intelligenza artificiale progettati per affrontare problematiche che i metodi deterministici non possono gestire in modo affidabile: identificazione e separazione di eventi in *pile-up*, ricostruzione di waveform parzialmente saturate e correzione automatica di segnali distorti o degradati dal rumore. Questi aspetti rappresentano il reale valore aggiunto dell'AI rispetto all'approccio tradizionale e costituiranno una direzione di ricerca nelle future evoluzioni del sistema. Al tempo stesso, la disponibilità delle *photon list* (`DL2`) apre la possibilità di implementare ulteriori DataProcessor dedicati all'analisi avanzata degli eventi ricostruiti, con tecniche avanzate di apprendimento automatico per l'analisi di time series tramite pattern recognition e anomaly detection, così da identificare automaticamente sequenze di eventi potenzialmente compatibili con fenomeni gamma come i TGF.

Un'ulteriore prospettiva riguarda la correlazione in tempo reale tra eventi gamma rilevati e dati relativi all'attività elettrica atmosferica, con l'obiettivo di migliorare la discriminazione dei fenomeni di origine meteorologica. In ottica futura sarà quindi necessario lo sviluppo di un ulteriore DataProcessor il cui task sarà quello di associare in tempo reale ciascun TGF rilevato dal detector, al relativo fulmine temporalesco che lo ha originato.

Infine, grazie ai risultati ottenuti sull'efficienza energetica dimostrata dal sistema, si apre la possibilità di considerare il framework non solo come parte del setup sperimentale di GammaSky presso il Monte Cimone, ma anche come candidato in futuri scenari a risorse particolarmente limitate, come potenziali missioni CubeSat, nei quali l'elaborazione a bordo tramite intelligenza artificiale rappresenta un argomento di crescente interesse. Oltre a tali impieghi, il framework RTA-DP potrà essere integrato anche in

altri software dedicati alla rivelazione gamma, come l'infrastruttura di analisi in tempo reale del Cherenkov Telescope Array Observatory (CTAO)[52], progetto cui INAF – OAS partecipa direttamente e dove la capacità di elaborare rapidamente grandi volumi di dati è essenziale per l'identificazione di fenomeni transienti e la generazione di allerte scientifiche.

Bibliografia

[1] J. R. Dwyer, D. Smith e S. Cummer, «High-Energy Atmospheric Physics: Terrestrial Gamma-Ray Flashes and Related Phenomena,» *Space Science Reviews*, pp. 133–196, 2012. DOI: 10.1007/s11214-012-9894-0.

[2] B.-B. Zhang, B. Zhang, K. Murase, V. Connaughton e M. S. Briggs, «How Long does a Burst Burst?» *The Astrophysical Journal*, vol. 787, n. 1, p. 66, 2014. DOI: 10.1088/0004-637x/787/1/66.

[3] G. J. Fishman, P. N. Bhat, R. Mallozzi et al., «Discovery of intense gamma-ray flashes of atmospheric origin,» *Science*, vol. 264, n. 5163, pp. 1313–1316, 1994. DOI: 10.1126/science.264.5163.1313.

[4] M. Marisaldi, F. Fuschino, C. Labanti et al., «Detection of terrestrial gamma ray flashes up to 40 MeV by the AGILE satellite,» *Journal of Geophysical Research: Space Physics*, vol. 115, n. A3, 2010. DOI: 10.1029/2009JA014502.

[5] M. S. Briggs, G. J. Fishman, V. Connaughton et al., «First results on terrestrial gamma ray flashes from the Fermi Gamma-ray Burst Monitor,» *Journal of Geophysical Research: Space Physics*, vol. 115, n. A7, 2010. DOI: 10.1029/2009JA015242.

[6] M. S. Briggs, S. Xiong, V. Connaughton et al., «Terrestrial gamma-ray flashes in the Fermi era: Improved observations and analysis methods,» *Journal of Geophy-*

sical Research: Space Physics, vol. 118, n. 6, pp. 3805–3830, 2013. DOI: 10.1002/jgra.50205.

[7] D. M. Smith, L. Lopez, R. P. Lin e C. P. Barrington-Leigh, «Terrestrial gamma-ray flashes observed up to 20 MeV,» *Science*, vol. 307, n. 5712, pp. 1085–1088, 2005. DOI: 10.1126/science.1107466.

[8] M. Marisaldi, F. Fuschino, C. Labanti et al., «AGILE Observations of Terrestrial Gamma-Ray Flashes,» *Il Nuovo Cimento C*, n. Online First, 2011. DOI: 10.1393/ncc/i2011-10870-5.

[9] NASA, *Spotting Terrestrial Gamma-Ray Flashes*, https://fermi.gsfc.nasa.gov/science/eteu/tgfs/, 2009.

[10] B. Grefenstette, D. Smith, B. Hazelton e L. Lopez, «First RHESSI terrestrial gamma ray flash catalog,» *Journal of Geophysical Research*, vol. 114, 2009. DOI: 10.1029/2008JA013721.

[11] Wikipedia, *Terrestrial gamma-ray flash*, https://en.wikipedia.org/wiki/Terrestrial_gamma-ray_flash.

[12] NASA, *NASA's Fermi Catches Thunderstorms Hurling Antimatter into Space*, https://www.nasa.gov/universe/nasas-fermi-catches-thunderstorms-hurling-antimatter-into-space/, 2011.

[13] M. B. Cohen, U. S. Inan e G. Fishman, «Terrestrial gamma ray flashes observed aboard the Compton Gamma Ray Observatory/Burst and Transient Source Experiment and ELF/VLF radio atmospherics,» *Journal of Geophysical Research: Atmospheres*, vol. 111, n. D24, 2006. DOI: 10.1029/2005JD006987.

[14] NASA, *Fermi GBM*, https://gammaray.msfc.nasa.gov/gbm/, 2008.

[15] M. Tavani, G. Barbiellini, A. Argan et al., «The AGILE Mission,» *Astronomy & Astrophysics*, vol. 502, n. 3, pp. 995–1013, 2009. DOI: 10.1051/0004-6361/200810527.

[16] P. Bastia, J. M. Poulsen, F. Monzani et al., «AGILE MCAL, the MINI-CALORIMETER,» *Astroparticle, Particle and Space Physics, Detectors and Medical Physics Applications*, pp. 904–908, 2006. DOI: 10.1142/9789812773678_0144.

[17] T. Neubert, N. Østgaard, V. Reglero et al., «The ASIM Mission on the International Space Station,» *Space Science Reviews*, vol. 215, n. 2, p. 26, 2019. DOI: 10.1007/s11214-019-0592-z.

[18] T. Neubert, N. Østgaard, V. Reglero, O. Chanrion, K. Ullaland e S. Yang, «A terrestrial gamma-ray flash and ionospheric ultraviolet emissions powered by lightning,» *Science*, vol. 367, n. 6474, pp. 183–186, 2020. DOI: 10.1126/science.aax3872.

[19] A. Di Giovanni, F. Arneodo, A. Al Qasim et al., «RAAD: LIGHT-1 CubeSat's payload for the detection of terrestrial gamma-ray flashes,» *Journal of Instrumentation*, vol. 18, n. 10, P10024, 2023. DOI: 10.1088/1748-0221/18/10/P10024.

[20] A. G. Wright, *The Photomultiplier Handbook*. Oxford University Press, 2017. DOI: 10.1093/oso/9780199565092.001.0001.

[21] R. Mirzoyan, «Use of SiPMs in Astro-Particle Physics and Space,» in *Ringberg Workshop on New Trends in Photo-Detection*, 2019. indirizzo: https://indico.cern.ch/event/791832/contributions/3356913/attachments/1871669/3080196/Mirzoyan-SiPM-ApP-Space-Ringberg-20-June-19.pdf.

[22] Qwerty123uiop, *PhotoMultiplierTubeAndScintillator.svg*, Wikimedia Commons, CC BY-SA 3.0, 2013. indirizzo: https://commons.wikimedia.org/wiki/File:PhotoMultiplierTubeAndScintillator.svg.

[23] Wikipedia, *Silicon photomultiplier*, https://en.wikipedia.org/wiki/Silicon_photomultiplier.

[24] D. Izzo, G. Meoni, P. Gómez, D. Dold e A. Zoechbauer, *Selected Trends in Artificial Intelligence for Space Applications*, 2022. indirizzo: https://arxiv.org/abs/2212.06662.

[25] S. Ungar, J. Pearlman, J. Mendenhall e D. Reuter, «Overview of the Earth Observing One (EO-1) Mission,» *IEEE Transactions on Geoscience and Remote Sensing*, vol. 41, pp. 1149–1159, 2003. DOI: 10.1109/TGRS.2003.815999.

[26] B. Denby e B. Lucia, «Orbital Edge Computing: Nanosatellite Constellations as a New Class of Computer System,» in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Association for Computing Machinery, 2020, pp. 939–954, ISBN: 9781450371025. DOI: 10.1145/3373376.3378473.

[27] G. Giuffrida, L. Fanucci, G. Meoni et al., «The Φ-Sat-1 Mission: The First On-Board Deep Neural Network Demonstrator for Satellite Earth Observation,» *IEEE Transactions on Geoscience and Remote Sensing*, vol. 60, pp. 1–1, 2021. DOI: 10.1109/TGRS.2021.3125567.

[28] G. Meoni, M. Märtens, D. Derksen et al., «The OPS-SAT case: A data-centric competition for onboard satellite image classification,» *Astrodynamics*, vol. 8, n. 4, pp. 507–528, 2024. DOI: 10.1007/s42064-023-0196-y.

[29] G. Giuffrida, L. Diana, F. Gioia et al., «CloudScout: A Deep Neural Network for On-Board Cloud Detection on Hyperspectral Images,» *Remote Sensing*, vol. 12, p. 2205, 2020. DOI: 10.3390/rs12142205.

[30] Wikipedia, *Phi-Sat-1*, https://en.wikipedia.org/wiki/Phi-Sat-1.

[31] C. Coelho, O. Koudelka e M. Merri, «NanoSat MO framework: When OBSW turns into apps,» in *IEEE Aerospace Conference 2017*, 2017. DOI: 10.1109/AERO.2017.7943951.

[32] Wikipedia, *OPS-SAT*, https://en.wikipedia.org/wiki/OPS-SAT.

[33] G. Labrèche, D. Evans, D. Marszk et al., «OPS-SAT Spacecraft Autonomy with TensorFlow Lite, Unsupervised Learning, and Online Machine Learning,» in *2022 IEEE Aerospace Conference (AERO)*, 2022, pp. 1–17. DOI: 10.1109/AERO53065.2022.9843402.

[34] A. S. Danielsen, T. A. Johansen e J. L. Garrett, «Self-Organizing Maps for Clustering Hyperspectral Images On-Board a CubeSat,» *Remote Sensing*, vol. 13, n. 20, 2021. DOI: 10.3390/rs13204174.

[35] M. E. Grøtte, R. Birkeland, E. Honoré-Livermore et al., «Ocean Color Hyperspectral Remote Sensing With High Resolution and Low Latency—The HYPSO-1 CubeSat Mission,» *IEEE Transactions on Geoscience and Remote Sensing*, vol. 60, pp. 1–19, 2022. DOI: 10.1109/TGRS.2021.3080175.

[36] A. Marin, C. Coelho, F. Deconinck, I. Babkina, N. Longépé e M. Pastena, «Phi-Sat-2: Onboard AI Apps for Earth Observation,» in *Space and Artificial Intelligence Conference 2021*, 2021.

[37] O. Cosmos, *Open Cosmos and ESA set new standard for space AI with the launch of AI app-powered Phisat-2 Earth Observation satellite*, https://www.open-cosmos.com/news/phisat-2-launch.

[38] ESA Φ-lab, *OrbitalAI Φsat-2*, https://platform.ai4eo.eu/orbitalai-phisat-2.

[39] European Space Agency, *New satellite to show how AI advances Earth observation*, https://www.esa.int/Applications/Observing_the_Earth/Phsat-2/New_satellite_to_show_how_AI_advances_Earth_observation.

[40] NASA, *What is GCN?* https://gcn.nasa.gov/docs/about.

[41] A. Bulgarelli, S. Caroff, A. Addis et al., «The Science Alert Generation system of the Cherenkov Telescope Array Observatory,» in *37th International Cosmic Ray Conference (ICRC 2021)*, 2021, p. 937. DOI: 10.22323/1.395.0937.

[42] A. Addis, A. Aboudan, A. Bulgarelli et al., «The Gamma-Flash real-time data pipeline for ground observation of terrestrial gamma-ray flashes,» in *Software and Cyberinfrastructure for Astronomy VII*, vol. 12189, SPIE, The International Society for Optical Engineering, 2022, ISBN: 9781510653597. DOI: 10.1117/12.2627961.

[43] P. Calabretto, A. Ursi, A. Tiberia et al., «The Gamma-Flash Program: high-energy radiation and particles in thunderstorms, lightning, and terrestrial gamma-ray flashes,» *Journal of Physics: Conference Series*, vol. 2985, p. 012 015, 2025. DOI: 10.1088/1742-6596/2985/1/012015.

[44] M. Nakhostin, «Recursive Algorithms for Real-Time Digital $CR-(RC)^n$ Pulse Shaping,» *IEEE Transactions on Nuclear Science*, vol. 58, n. 5, pp. 2378–2381, 2011. DOI: 10.1109/TNS.2011.2164556.

[45] V. T. Jordanov, G. F. Knoll, A. C. Huber e J. A. Pantazis, «Digital techniques for real-time pulse shaping in radiation measurements,» *Nuclear Instruments and Methods in Physics Research Section A*, vol. 353, pp. 261–264, 1994. DOI: 10.1016/0168-9002(94)91652-7.

[46] A. P. Jezghani, L. J. Broussard e C. B. Crawford, *A Recursive Method for Real-Time Waveform Fitting with Background Noise Rejection*, 2020. indirizzo: https://arxiv.org/abs/2012.05937.

[47] W. Li, Q. Zhou, Y. Zhang et al., «An Ultra-Throughput Boost Method for Gamma-Ray Spectrometers,» *Sensors*, vol. 22, n. 19, p. 7397, 2022. DOI: 10.3390/s22197397.

[48] A. Bulgarelli, A. Aboudan e A. Addis, *GAMMA-FLASH DACS Software Interface Control Document*, 2024. indirizzo: https://docs.google.com/document/d/1lBRKIZoel5JaIq7YLkpfv0_b22bQL1RXUJ8oG6OQVjE/edit?usp=sharing.

[49] A. Bulgarelli, *rta data processor v3.9.0*, 2024. indirizzo: https://docs.google.com/document/d/1mzNnIdKFq3DDZPdPwj-WKWdvd81IkWaEhMl26azeqq8.

[50] N. Parmiggiani, A. Bulgarelli, L. Castaldini et al., «The New Architecture of the Online Observation Quality System for the ASTRI Mini-Array Project,» in *ADASS XXXIV (2024) conference)*, 2025. DOI: 10.48550/arXiv.2507.15656.

[51] S. Arnold, R. Nuzzaci e A. Gordon-Ross, «Energy budgeting for CubeSats with an integrated FPGA,» *IEEE Aerospace Conference Proceedings*, 2012. DOI: 10.1109/AERO.2012.6187240.

[52] CTAO Central Organisation, *Cherenkov Telescope Array Observatory (CTAO)*, https://www.cta-observatory.org, 2025.