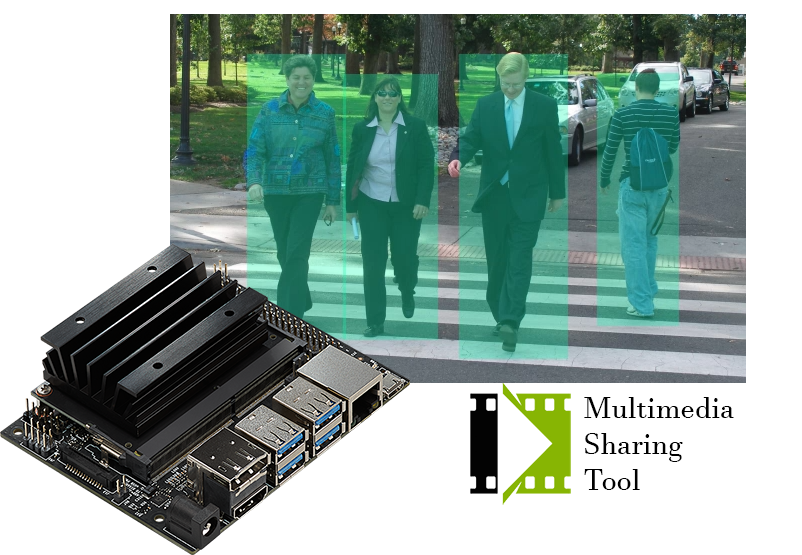
Università degli studi di Modena e Reggio Emilia

Dipartimento di Ingegneria “Enzo Ferrari”

Corso di Laurea Magistrale in Ingegneria Informatica



Intrusion Detection in ambito CUDA e streaming dei dati elaborati in Real-time con Nvidia Jetson Nano

pROGETTO DI CORSO

Werther Berselli | High Performance Computing | a.a 2018/2019

# Sommario:

1. Introduzione
2. Casi d’uso
   1. Stato dell’arte
   2. Scelte di progetto
3. Implementazione e funzionamento
   1. Librerie utilizzate
   2. Schema a blocchi
   3. Funzionalità accelerabili
   4. Funzionalità non accelerabili
   5. Metriche
4. Test e risultati
   1. Osservazioni sui test effettuati
   2. Profiling funzionalità CUDA e ARM su Nvidia Jetson Nano
5. Conclusioni
   1. Ottimizzazioni e miglioramenti futuri

# Introduzione:

In questa relazione viene presentata un’applicazione per effettuare Intrusion Detection, manipolazioni e conseguente codifica e streaming di frame multipli utilizzando un’architettura Client-Server.

L’applicazione è stata pensata per poter essere il più generale possibile offrendo un’alta possibilità di personalizzazione; inoltre ogni ambito è stato progettato per poter sfruttare il più possibile tecniche di parallelismo ad alte prestazioni che è possibile ottenere sfruttando un’architettura basata su CUDA.

Il sistema hardware di riferimento per il progetto è l’Nvidia Jetson Nano, ma è possibile utilizzare l’applicazione su qualsiasi sistema Unix Cuda-capable.

È possibile fare intrusion detection partendo da un file video, da una porzione di schermo o da una camera, così come il cambio di risoluzione (downscaling o upscaling), della scala colori dei pixel e applicare un filtro di blur per rendere non riconoscibili i soggetti inviati mediante lo stream video.

Il progetto è ottenibile dalla repository di GitHub, comprensiva delle istruzioni per installare le dipendenze necessarie e per eseguire la build: <https://github.com/Werther158/multimedia_sharing_tool>

# Casi d’uso:

Il caso d’uso più tipico è in ambito embedded, utilizzando una camera per catturare i frame e un dispositivo embedded come server che si occupa di effettuare l’Intrusion detection e creare lo stream video da inviare a un client connesso in rete (tipicamente wireless).

Un altro caso d’uso dell’applicazione è quello di elaborare un video già presente e in riproduzione in real-time sullo schermo del sistema Server; in tal caso l’applicazione fornisce la possibilità di definire una regione di interesse (ROI) che dovrà essere elaborata e inviata al sistema Client.

Infine viene anche data la possibilità all’utente di utilizzare un file video in formato matroska, su cui poter fare Intrusion detection, elaborazione, codifica e invio al Client mediante lo stream video. Per i file video viene anche tenuto in considerazione l’audio; esso infatti viene inviato separatamente ma in contemporanea rispetto ai frame, in modo da essere riprodotto sul sistema Client.

## Stato dell’arte:

Ad oggi i sistemi di videosorveglianza sono per lo più composti da una videocamera e da un sistema di registrazione su memorie esterne come dischi e nastri; solamente i sistemi più avanzati prevedono anche l’utilizzo di sensori per rilevare il movimento ed eventualmente far scattare un allarme. Questi sensori spesso non sono però accurati, in quanto rilevando qualsiasi tipo di movimento è facile che si attivino in presenza di falsi positivi. Pochissimi poi impiegano l’utilizzo di tecniche di visione per fare Intrusion detection sui frame catturati sfruttando reti deep, poichè questo è un ambito ancora relativamente recente.

Questi sistemi, da ricerche effettuate sono molto costosi ed equipaggiati da potenti cpu e ampie memorie a fronte del notevole workload che grava sull’hardware. L’obiettivo di questo progetto è quello di realizzare un sistema di Intrusion detection sulla falsa riga dei sistemi già esistenti, che però sfrutti l’accelerazione offerta da una piattaforma embedded basata su CUDA (come l’Nvidia Jetson Nano) e contemporaneamente che possa evitare l’onere della registrazione video in locale ma che consenta, mediante un encoding in real-time, di eseguire uno streaming sfruttando la rete locale o globale.

Lo stato dell’arte attuale in termini di codifica video è l’H.265, anche se non è ancora largamente supportato da tutti i dispositivi; l’H.264 risulta ancora molto utilizzato essendo anch’esso molto avanzato ed efficiente, ma anche supportato da quasi la totalità dei dispositivi esistenti.

Per fare streaming audio-video in tempo reale il protocollo più utilizzato e più indicato è l’RTSP (Real Time Streaming Protocol), che per questo compito risulta essere più efficiente di TCP/IP e più affidabile di UDP.

## Scelte di progetto:

Il sistema è stato progettato per essere modulare, in modo da poter introdurre facilmente nuove funzionalità e modifiche alle funzioni già presenti.

Come scelta progettuale si è deciso di equipaggiare il sistema con una GUI, per rendere più accessibile e semplice l’utilizzo dell’applicazione da parte di utenti esterni. La GUI è stata realizzata con l’IDE utilizzato per implementare l’intero progetto, QT Creator.

La GUI è stata realizzata interamente supportando la lingua inglese e italiana, selezionabili nella finestra principale; nelle finestre sono stati inseriti dei Tooltip per chiarire all’utente eventuali dubbi riguardanti le voci che vengono visualizzate a schermo.

La codifica viene fatta seguendo lo standard H.264 e lo streaming audio-video viene eseguito su protocollo RTSP.

# Implementazione e funzionamento:

Il linguaggio scelto è il C++, lo stesso utilizzato per le funzionalità implementate che sfruttano il CUDA.

Sono stati implementati diversi thread concorrenti utilizzando il costrutto Qthread offerto da Qt. I thread vengono sincronizzati mediante l’uso di semafori della libreria POSIX *semaphore.h*.

La struttura di comunicazione dei dati tra i thread produttori e consumatori e verso il processo che gestisce lo streaming attuato con Ffmpeg è stata realizzata mediante l’uso di Named Pipe Unix (anche dette FIFO).

Per questioni di semplicità e ordine tutto il progetto è stato implementato utilizzando l’inglese come lingua principle, sia per i nomi delle variabili, che per i commenti e i commit sulla repositori di GitHub.

Per prima cosa è necessario scegliere se avviare l’applicazione come Server (colui che invierà lo stream video) o come Client (colui che riceverà lo stream video). La finestra che si presenta permette di configurare il tipo di sorgente da utilizzare e le impostazioni da applicare sui frame. Normalmente è il Server a definire la configurazione da utilizzare ma è possibile anche delegare le impostazioni di configurazione al Client mediante una checkBox.

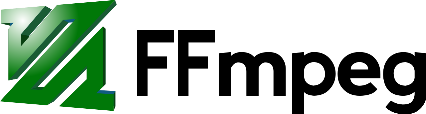
Una volta avviato il Server, il Client può collegarsi utilizzando la coppia indirizzo:porta del Server. Una volta che il collegamento è stato stabilito lo stato di connessione viene mostrato da una label nelle due finestre del Client e del Server in alto a destra. Da questo momento una connessione TCP/IP è in atto tra le due entità, che possono comunicare tramite messaggi; la connessione TCP/IP viene utilizzata anche per inviare comandi di inizio e fine stream, configurazioni dello stream e altre informazioni utili per l’applicazione, nascoste all’utente.

In una listBox nella finestra si trova (sia per il Client che per il Server) un riepilogo delle impostazioni attualmente impostate. In qualsiasi momento entrambi possono avviare lo stream video mediante un bottone. Entrambi potranno poi interrompere lo stream utilizzando lo stesso bottone.

## Librerie utilizzate:

 QT Creator è stato utilizzato come ambiente di sviluppo per il codice e per la GUI.

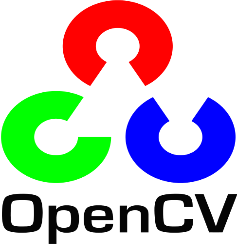
cURL development library è stata utilizzata per poter recuperare in fase di configurazione l’indirizzo pubblico relativo al sistema in uso e visualizzarlo sulla finestra di configurazione in modo da essere facilmente comunicabile all’altra entità (Client o Server).

 FFmpeg è una potente applicazione utilizzata mediante chiamate di sistema per eseguire la codifica e lo streaming delle immagini elaborate.

Cuda Toolkit è utilizzato dalle architetture CUDA per poter compilare ed eseguire kernel .cu su acceleratori Nvidia.

 Nvidia TensorRT viene sfruttata per poter utilizzare reti neurali in ambito CUDA.

Jetson-inference library viene utilizzata per eseguire l’Intrusion detection sui frame e rilevare eventuali persone nelle immagini tracciando dei bounding box.



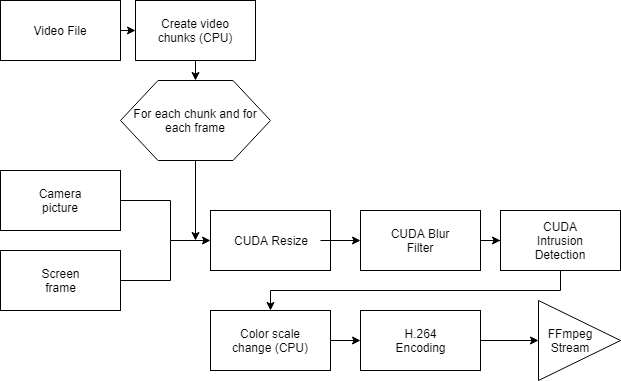
OpenCV con supporto a CUDA e pacchetto Contrib viene utilizzata per definire la Regione di Interesse (ROI) quando viene scelto di fare lo streaming di una porzione di schermo del Server, per fare il ridimensionamento dei frame e per applicare eventualmente un filtro di Blur di intensità più o meno marcata (definibile in fase di configurazione).

## Schema a blocchi:

Nell’immagine seguente viene mostrato lo schema a blocchi che descrive il processo di streaming eseguito dall’applicazione, partendo dall’acquisizione della sorgente di input scelta fino alla codifica e allo streaming vero e proprio.

Le sorgenti Camera e Schermo prevedono l’acquisizione di un frame alla volta, con l’elaborazione e la fase di stream che intercorrono tra l’acquisizione di un frame e il successivo.

Un file video invece viene diviso in porzioni di frame e di audio (chunk), in base a un parametro scelto in fase di configurazione che definisce la lunghezza di ogni chunk desiderata (di default impostata a 10 secondi); il chunk viene elaborato frame per frame prima di eseguire lo stream dell’intero chunk e passare al successivo.



## Funzionalità accelerabili:

Tra le funzionalità che possono trarre vantaggio dell’accelerazione di un’architettura basata su CUDA, l’Intrusion detection è quella più importante. La classe CudaDetectionThread si occupa di mantenere un thread in attesa di un semaforo attivato dal thread principale, che al momento opportuno gli comunica che un frame (o un chunk) è in attesa di essere processato. Questo procedimento avviene in un ciclo che dura fino al termine dello streaming. Una volta che viene inviato il segnale di post al semaforo, il thread chiama per ogni frame la funzione detectOnImage(std::string file\_path) a cui viene passato come parametro il path del frame su cui fare detection. Questa funzione fa uso della libreria Jetson-inference attraverso la chiamata alla funzione *Detect()* sulla rete neurale. Alla fine viene emesso un Qt Signal per avvisare il thread principale che l’Intrusion detection è stato applicato e il frame o il chunk è pronto.

Il ciclo principale del *CudaDetectionThread* è il seguente:

while(detection\_running)

{

sem\_wait(&sem\_run);

if(single\_frame)

{

// Elaborate a single frame

file\_path = Configurations::current\_frame\_path + "/output.bmp";

detectOnImage(file\_path);

}

else

{

// Elaborate all chunk frames

QDir directory(QString::fromStdString

(Configurations::current\_frame\_path));

QStringList images = directory.entryList

(QStringList() << "\*.bmp", QDir::Files);

foreach(QString filename, images) {

detectOnImage(Configurations::current\_frame\_path

+ "/" + filename.toStdString());

}

}

emit detectionDone();

}

Di seguito la funzione detectOnImage(std::string file\_path):

void CudaDetectionThread::detectOnImage(std::string file\_path)

{

if( !loadImageRGBA(file\_path.c\_str(), reinterpret\_cast<float4\*\*>(&imgCPU),

reinterpret\_cast<float4\*\*>(&imgCUDA), &imgWidth, &imgHeight) )

{

printf("failed to load image '%s'\n", file\_path.c\_str());

return;

}

/\*

\* detect objects in image

\*/

detectNet::Detection\* detections;

detections = nullptr;

net->Detect(imgCUDA, static\_cast<uint32\_t>

(imgWidth), static\_cast<uint32\_t>(imgHeight), &detections);

// wait for the GPU to finish

CUDA(cudaDeviceSynchronize());

if(!saveImageRGBA(file\_path.c\_str(), reinterpret\_cast<float4\*>(imgCPU),

imgWidth, imgHeight, 255.0f))

printf("detectnet-console: failed saving %ix%i image to '%s'\n",

imgWidth, imgHeight, file\_path.c\_str(

La funzionalità di resize e l’applicazione del filtro di Blur ai frame sono fatte sfruttando il CUDA con l’ausilio di OpenCV all’interno della funzione imageScaleBlur(std::string frame\_path):

if(Configurations::frame\_size\_changed)

{

// Set new resolution to frame

size.width = Configurations::frame\_width;

size.height = Configurations::frame\_height;

cv::cuda::resize(input\_device, output\_device, size);

}

if(Configurations::blur\_effect != 0)

{

// Set a gaussian blur filter

cv::Ptr<cv::cuda::Filter> gauss = cv::cuda::createGaussianFilter

(output\_device.type(), output\_both.type(),

cv::Size(21, 21), Configurations::blur\_effect \* 2,

Configurations::blur\_effect \* 2);

gauss->apply(output\_device, output\_both);

output\_both.download(output\_host);

}

## Funzionalità non accelerabili:

Le funzionalità che utilizzano direttamente Ffmpeg non sono accelerabili in quanto sebbene quest’ultimo sia un tool molto potente, non offre la possibilità di accelerare la computazione tramite CUDA ma utilizza solamente la CPU. I processi di creazione dei chunk video, di encoding e streaming, attuati grazie all’uso di Ffmpeg rientrano in questa categoria.

## Metriche:

Il progetto ha richiesto un tempo di realizzazione di circa un mese e mezzo, da metà luglio alla fine di agosto 2019, superando la quota di 4400 righe di codice e 16 classi totali nella sua versione finale.

Per quanto riguarda la GUI le finestre dell’applicazione sono 5.

Sulla repository Github sono stati superati i 100 commit, di cui l’80% circa riguardanti il codice sorgente e il restante 20% riguardanti il file delle dipendenze esterne e della build Readme.md.

# Test e risultati:

Sono stati effettuati numerosi test di performance dell’applicazione, utilizzando tre diversi sistemi.

Ogni valore temporale nei grafici è stato calcolato come valore mediato di 5 test fatti in successione.

In ambito embedded è stato scelto un Nvidia Jetson Nano, basato su architettura Nvidia Maxwell con 128 core CUDA, una CPU Quad-core ARM cortex A57 e una memoria LPDDR4 da 4 GB a 64 bit.

Successivamente è stato utilizzato un notebook con una GPU Nvidia GT840m, basata su architettura Maxwell e con 384 core CUDA, una CPU Dual-core Intel I7 4500U e una memoria DDR3 da 8 GB a 64 bit.

Infine è stato utilizzato un sistema fisso con una GPU Nvidia GTX 1080, basata su architettura Pascal con 2560 core CUDA, una CPU Quad-core Intel I7 2600k e una memoria DDR3 da 16 GB a 64 bit.

Sono state testate le performance utilizzando un file video matroska con risoluzione 1280x720, una camera con risoluzione 1280x720 e una porzione di schermo di 450x240 pixel.

Di seguito sono riportate i grafici delle performance dei test effettuati, con indicati sull’asse delle ascisse il sistema in uso, sull’asse delle ordinate il tempo di esecuzione del task in oggetto. Sulle candele sono poi riportati i frame per secondo totalizzati in ciascun task.

**Test 1: Camera, Risoluzione: default, Color scale: 24 bpp, Blur: off, Intrusion detection: on**

**Test 2: Camera, Risoluzione: 426 x 240, Color scale: 8 bpp (gray), Blur: off, Intrusion detection: on**

**Test 3: Camera,** **Risoluzione: 426 x 240, Color scale: 8 bpp (gray), Blur: on, Intrusion detection: on**

**Test 4: Schermo, Risoluzione: default, Color scale: 24 bpp, Blur: off, Intrusion detection: on**

**Test 5: Schermo, Risoluzione: 426 x 240, Color scale: 8 bpp (gray), Blur: off, Intrusion detection: on**

**Test 6: Schermo, Risoluzione: 426 x 240, Color scale: 8 bpp (gray), Blur: on, Intrusion detection: on**

**Test 7: Video file, Risoluzione: default, Color scale: 24 bpp, Blur: off, Intrusion detection: off**

**Test 8: Video file, Risoluzione: default, Color scale: 24 bpp, Blur: off, Intrusion detection: on**

Nei due grafici seguenti il Jetson Nano non è presente poichè le sue prestazioni non sono sufficienti per sostenere l’elaborazione dei frame nei chunk e contemporaneamente applicare Intrusion detection, encoding e streaming. Vengono presi in considerazione solamente gli altri due sistemi.

**Test 9: Video file, Risoluzione: 426 x 240, Color scale: 8 bpp (gray), Blur: on, Intrusion detection: off**

## Osservazioni sui test effettuati:

Dai test effettuati è possibile osservare che l’Intrusion detection risulta essere l’operazione più pesante, a prescindere dal tipo di sorgente di ingresso utilizzata. Partendo da immagini HD, le operazioni di detection risultano particolarmente onerose poi in termini di risorse.

Il Jetson Nano è risultato come preventivato, il sistema più svantaggiato dei 3; ma grazie alla sua efficienza energetica (è sufficiente solamente un’alimentazione a 5V, 2.5 Ampere; utilizzata nei test) lo rendono ottimo per essere utilizzato in contesti dove lo spazio e l’energia elettrica sono limitati, a fronte di una minore fluidità nello streaming e maggiori tempi di risposta.

## Profiling funzionalità CUDA e ARM su Nvidia Jetson Nano:

Per effettuare il profiling delle funzionalità che utilizzano l’accelerazione CUDA è stato preso in esame il seguente caso d’uso:

**Camera, Resize: 426 x 240, Color scale: 8 bpp (gray), Blur: on, Intrusion Detection: on**

Di seguito vengono riportati i risultati dei tempi medi ottenuti dal Jetson nano mettendo a confronto le performance delle versioni CUDA di Resize + Filtering e Intrusion Detection rispetto alla versione ARM.

**CUDA speedup: 1.89 CUDA speedup: 1.40**

Dai test effettuati è risultato vantaggioso l’utilizzo dell’accelerazione CUDA (nonostante l’overhead pagato per i trasferimenti delle immagini nella memoria GPU) sia per l’Intrusion Detection che per il Resize e Filtering. Ottenendo uno speedup considerevole nell’esecuzione di entrambe le funzionalità.

Nei seguenti due grafici è possibile vedere come variano le proporzioni dei tempi d’esecuzione di tutte le operazioni necessarie a produrre un singolo frame, nelle due diverse modalità di esecuzione ARM e CUDA.

**Total Frame Time: 0.44313 s Total Frame Time: 0.33402 s**

**AVG Frames Per Second: 2.26 AVG Frames Per Second: 3.00**

I tempi per l’acquisizione di un frame, per l’esecuzione dello streaming e il resto delle operazioni necessarie (Others), vengono mantenuti invariati.

L’operazione più lunga nell’elaborazione di un frame è risultata essere l’Intrusion Detection, che è stata anche la funzionalità ad aver tratto più beneficio dall’impiego di CUDA. Nel grafico relativo alla versione CUDA si può notare un leggero ingrandimento della porzione di “Resize and blur”, che pur avendo anch’essa tratto beneficio dall’utilizzo di CUDA non è migliorata come l’ID. Nella seconda versione si riesce a distinguere poi tra le operazioni effettuate anche il tempo dedicato all’acquisizione di un singolo frame.

Infine, sono risultate ottime le performance delle funzionalità di codifica e di streaming attuate da Ffmpeg, che in entrambe le versioni dell’applicazione non hanno impattato in modo significativo sulle performance globali del sistema.

# Conclusioni:

Questo progetto mi ha consetito di acquisire più familiarità con il mondo dei dispositivi embedded ad alte prestazioni e degli acceleratori grafici; dovendo pensare non solo a soluzioni efficaci in termini di funzionalità ma anche efficienti.

Aver progettato un’applicazione generica a cui poter applicare accelerazioni tramite CUDA mi ha aiutato oltre che a prendere più dimestichezza con questa tecnologia e le sue librerie, anche a capire quali funzionalità del progetto fossero più facilmente parallelizzabili rispetto a quelle più fortemente sequenziali.

## Ottimizzazioni e miglioramenti futuri:

In futuro l’applicazione potrebbe essere migliorata aggiungendo un array circolare che possa funzionare come una sorta di buffer in modo da poter lavorare su più frame contemporaneamente ed evitare di applicare le operazioni di manipolazione, detection, encoding e stream in modo sequenziale su un frame alla volta. Questo aumenterebbe sicuramente il throughput dei frame incrementando di conseguenza il numero di frame per secondo visualizzati dal Client, a fronte dell’introduzione di un certo ritardo, dipendente dalla grandezza dell’array circolare.