JPEG Encoder on GPU

State of the art Review and Critical Requirement Analysis

Document ID	State of the art Review and Critical Requirement Analysis
Issue	1.0
Date	27/04/2020
Copyright	

Prepared by	**Authorized by**
Francesco de Gioia	Luca Fanucci

Indice:

Indice delle figure

Indice delle tabelle

1 Introduzione

Il seguente documento descrive le attività compiute per rispondere alla richiesta di Alkeria S.r.l. di analizzare algoritmi, librerie e piattaforme per accelerare la compressione JPEG su GPU.

In questo studio sono state considerate librerie commerciali e librerie opensource con licenza libera (Custom BSD-like).

Come richiesto da Alkeria, le piattaforme utilizzate nello studio sono state:

- Nvidia Jetson TX2
- Palit Nvidia GeForce GTX 1650

Ulteriori prove sono state effettuate su Nvidia Jetson Nano.

In questo studio sono state utilizzate due piattaforme di sviluppo, di cui una usata per sviluppo in ambiente Linux e una per sviluppo ambiente Windows.

Come piattaforma di sviluppo Linux è stato utilizzato un server con processore AMD EPYC 7301 16-Core 2.7 GHz con GPU Nvidia Tesla T4. Su questa piattaforma erano preinstallate le librerie CUDA Toolkit 10.2. Questa piattaforma è stata utilizzata per il solo sviluppo del codice del programma di benchmark, il tempo di esecuzione della compressione JPEG non è stato incluso in questa analisi in quanto questa tipologia di GPU non corrisponde al target richiesto da Alkeria.

Come piattaforma di sviluppo in ambiente Windows 10 è stato utilizzato un tower PC con processore Intel i5 e scheda video Palit Nvidia GeForce GTX 1650. Sulla piattaforma sono stati installati i driver della GPU e le librerie CUDA Toolkit 10.2.

Per misurare le prestazioni delle librerie e delle GPU, è stato realizzato un software di benchmark eseguibile sia su Linux che su Windows. L'analisi delle metriche considerate nel benchmark ed uno studio comparato dell'esecuzione delle librerie ha contribuito a definire alcuni elementi dell'architettura di riferimento riportata nel documento *Reference Architecture and development plan*.

Come integrazione all'analisi delle librerie che rappresentano lo *State of the Art* per applicazioni commerciali, sono stati revisionati una serie di articoli presenti in letteratura. Si è analizzata la fattibilità delle implementazioni proposte sulla base della complessità di sviluppo e dell'effettivo incremento delle prestazioni.

Il resto del documento è organizzato come segue. In Requisiti di Progetto vengono presentati i requisiti funzionali e non funzionali presentati da Alkeria, adottati come guida nella definizione delle metriche di benchmark e per delineare le decisioni implementative proposte nell'architettura finale. In Librerie Software e Piattaforme Hardware vengono descritte nel dettaglio rispettivamente le librerie e le piattaforme utilizzate in questo progetto con particolare attenzione alle differenti caratteristiche architetturali delle GPU e all'effetto che queste differenze hanno sulle prestazioni finali. In Benchmark viene presentata la descrizione del programma di benchmark, le metriche analizzate ed i risultati ottenuti sulle varie piattaforme con relativa interpretazione e discussione. In Articoli e Pubblicazioni vengono revisionati alcuni articoli significativi che descrivono tecniche ed algoritmi finalizzati a migliorare le prestazioni di compressione JPEG su GPU. Infine, in Conclusioni si riassumono le caratteristiche osservate delle librerie e piattaforme esaminate e si riportano possibili soluzioni implementative da includere nell'architettura di riferimento e relative difficoltà di realizzazione.

2 Compressione JPEG

Lo standard JPEG descritto in [1], definisce un algoritmo di compressione immagini basato su trasformata discreta del coseno (DCT) e codifica entropica (Entropy encoding), lo standard non definisce né richiede un colorspace specifico, non specifica o codifica informazioni sul pixel aspect ratio e non include caratteristiche di acquisizione dell'immagine. La definizione di questi elementi viene riportata nel documento che descrive il JPEG File Interchange Format (JFIF) [2] che rappresenta il file format standard per immagini compresse con JPEG.

In [1], vengono proposti quattro *Modes of Operation* per l'econding JPEG:

- Sequential Encoding: Nell'encoding sequenziale ogni componente (canale) dell'immagine viene codificato procedendo da sinistra a destra e dall'alto in basso.
- Progressive Encoding: In questa modalità l'immagine viene compressa a livelli di definizione incrementali (da un'immagine poco definita ad una più definita) ed è utile in caso di lunghi tempi di trasmissione.
- Lossless Encoding: In questa modalità la compressione è lossless, nel senso che è possibile recuperare esattamente le varie componenti dell'immagine originale.
- Hierarchical encoding: L'immagine viene codificata a risoluzioni diverse in modo da permetterne la visualizzazione senza dover essere prima decodificata a dimensioni originali.

Per ogni /mode of operation/ lo standard JPEG definisce codec per immagini a 8 bit e 12 bit.

Tra le codifiche basate sulla trasformata discreta del coseno (Discrete Cosine Transform, DCT), la codifica JPEG Baseline è quella più supportata e sufficiente per la maggior parte delle applicazioni.

La codfica di immagini a più componenti può essere considerata come un'estensione del processo di codifica di una componente ad ogni componente. Le codifiche ottenute dovranno essere poi ricongiunte a formare la *datastream* tramite metodi di interlacciamento (interleaving) o concatenazione.

3 FDCT e IDCT

1. La FDCT (Forward Discrete Cosine Transform) e la IDCT (Inverse Discrete Cosine Transform), rappresentano la trasformata discreta del coseno e la sua trasformata inversa, e costituiscono le operazioni che permettono di ottenere una notevole compressione delle dimensioni (in byte) dell'immmagine senza compromettere eccessivamente la qualità. In modo analogo alla trasformata di Fourier, la DCT permette di rappresentare il segnale in ingresso come una combinazione lineare di componenti a differenti frequenze. L'assunzione che permette di raggiungere alti livelli di compressione e al contempo di evitare la degradazione dell'immagine è che nelle immagini naturali le componenti a frequenze più basse sono maggiormente presenti rispetto alle componenti a frequenze più alte. Spesso questa assunzione è corretta – si pensi ad immagini con aree a colore uniforme –, ma quando questa assunzione viene violata, nell'immagine codificata sono visibili delle distorsioni. Teoricamente la ricostruzione dell'immagine originale è possibile solo per una DCT a precisione infinita, ma in pratica anche DCT a precisione finita possono essere utilizzate per ricostruire l'immagine originale senza perdita di dettaglio. Inoltre, come nel caso della trasformata di Fourier, anche per la DCT sono state realizzate delle implementazioni ottimizzate per varie tipologie di sistemi come processori General Purpose, DSP, FPGA e ASIC.

Prima di applicare la FDCT, l'immagine in ingresso viene suddivisa in blocchi d 8x8 *samples* (pixel) convertiti da unsigned integer su P bit (con input range [0, 2^(P-1)]) a signed integer su P bit (con range [-2^(P-1), 2^(P-1)-1]). Applicata la FDCT ad un blocco di 8x8 samples, si ottengono 8x8 coefficienti relativi alle componenti frequenziali 2D. Il primo dei 64 coefficienti è la componente DC,

mentre i rimanenti 63 coefficienti sono le componenti AC. La trasformata DCT permette di concentrare la maggior parte dell'informazione del segnale in ingresso nelle componenti frequenziali a bassa frequenza che si vanno a collocare nella prima parte superiore della matrice 8x8 di uscita, mentre le componenti frequenziali a più alta frequenza hanno coefficienti vicini a zero e possono dunque essere ignorati nella codifica senza perdita di informazione.

3.1 Quantizzazione

Dopo la trasformata DCT, viene effettuata la quantizzazione (dequantizzazione) dei 64 coefficienti della DCT utilizzando la tabella di quantizzazione Q. Formalmente siano Q la tabella di quantizzazione e F output della FDCT, allora:

$$F_Q(u,v) = floor(\frac{F(u,v)}{Q(u,v)}),$$

con u e v coordinate frequenziali, cioè ogni elemento viene diviso per il suo corrispondente.

La fase di quantizzazione risulta essere l'elemento *lossy* della compressione JPEG, in quanto la funzione non è invertibile.

Lo standard JPEG specifica che un decoder può memorizzare al massimo quattro tabelle di quantizzazione diverse. In un comune file JPEG (file JFIF) vengono associate alla componente Luma (Y) e Chroma (Cb e Cr) una coppia di tabelle rispettivamente per il coefficiente DC e per i coefficienti AC. In totale in un file JFIF le quattro tabelle presenti sono: Y_DC, Y_AC, CbCr_DC e CbCr_AC. In [3] vengono riportate le tabelle di quantizzazione consigliate determinate in modo sperimentale per le immagini e gli schermi CCIR-601.

Dopo la quantizzazione, al coefficiente DC di ogni blocco 8x8 viene sottratto il coefficiente del blocco precedente (il primo blocco ha il coefficiente DC precedente uguale a zero), questa operazione è utile per sfruttare la correlazione tra blocchi adiacenti e agevolare l'entropy coding. I coefficienti AC vengono invece ordinati secondo una sequenza a "zig-zag" sempre per facilitare la fase successiva di entropy coding (si veda Figura 1). La maggior parte delle implementazioni utilizzano una lookup table per implementare questo tipo di ordinamento.

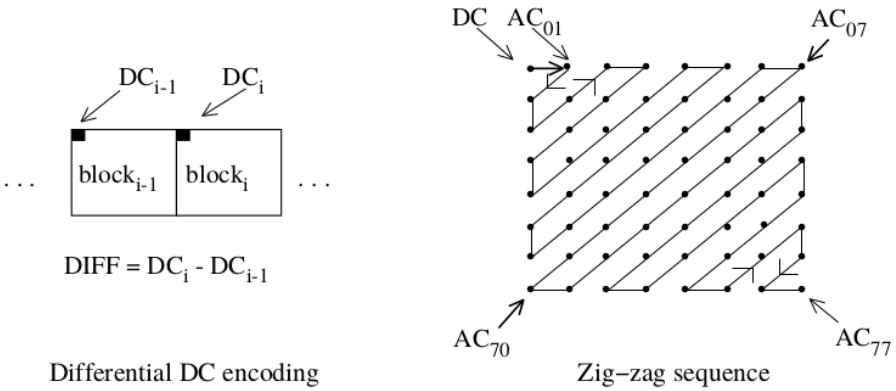

Figura 1 Preparazione dei samples alla fase di entropy coding. Fonte: [1].

3.2 Entropy Coding

Lo standard JPEG specifica due metodi di entropy coding lossless: Huffman coding e Arithmetic coding. L'Huffman coding necessita di tabelle di codici di Huffman che devono essere usate sia in fase di codifica che di decodifica. Queste tabelle possono essere tabelle predifinite o specifiche per l'immagine di input. In quest'ultimo caso la compressione risulta essere più efficiente a prezzo di tempi di esecuzione più lunghi legati alla costruzione della tabella.

L'arithmetic coding non richiede tabelle perché ricava misure statistiche dall'immagine. Risulta essere più complesso da implementare della codifica di Huffman e quindi poco supportato.

Il codec Baseline sequential utilizza l'Huffman coding con input a 8 bit. È utile dividere l'entropy coding in due fasi: nella prima fase si convertono le sequenze di coefficienti in una sequenza intermedia di simboli; nella seconda fase la sequenza di simboli viene convertita in un datastream binario di simboli contigui usando le tabelle di Huffman fornite in input al codec.

Le tabelle di Huffman, relative alle componenti Luma e Chroma, utilizzate per la compressione JPEG di un'immagine multicanale sono incluse nel file JFIF.

Per immagini JPEG multicanale la datastream può contenere la codifica interlacciata o sequenziale (non-interleaved) delle componenti. Alcuni decoder JPEG non supportano immagini non-interleaved a meno di utilizzare un Progressive Encoding come mode of operation.

Si definiscono due tabelle di Huffman: una per le componenti AC e una per le componenti DC. Di conseguenza, due componenti devono condividere una tabella. Per le componenti noninterleaved, questo non si applica perché, visto che le componenti sono separate nel file, terminata la decodifica di una componente possono essere caricate nuove tabelle relative alla componente da decodificare.

Le immagini JPEG a colori possono essere interleaved o non-interleaved. In un JPEG interleaved le componenti vengono salvate in successione, mentre in un JPEG non-interleaved le componenti sono salvate separatamente. Alcuni decoder JPEG non supportano immagini non-interleaved a meno di utilizzare una codifica progressiva (progressive encoding, invece di sequential encoding).

La codifica sequenziale de coefficienti ottenuti dalla FDCT avviene in due fasi: nella prima fase i coefficienti, ordinati secondo l'ordinamento a zig-zag, vengono convertiti in simboli intermedi; nella seconda fase i simboli vengono convertiti in datastream utilizzando le apposite tabelle di Huffman. Per la conversione dei simboli intermedi in datastream e' necessario fornire come input la tabella di Huffman.

Nel caso di immagini a componenti interleaved, ognuna delle Ci componenti è partizionata (divisa) in regioni di dimensione HixVi *data units* ordinate da sinistra a destra, dall'alto in basso, all'interno di ogni regione le *data units* sono anch'esse ordinate da sinistra a destra e dall'alto in basso. Come rappresentato in Figura 2. JPEG definisce il termine Mininum Coded Unit (MCU) come il minimo gruppo di *data unit* interleaved.

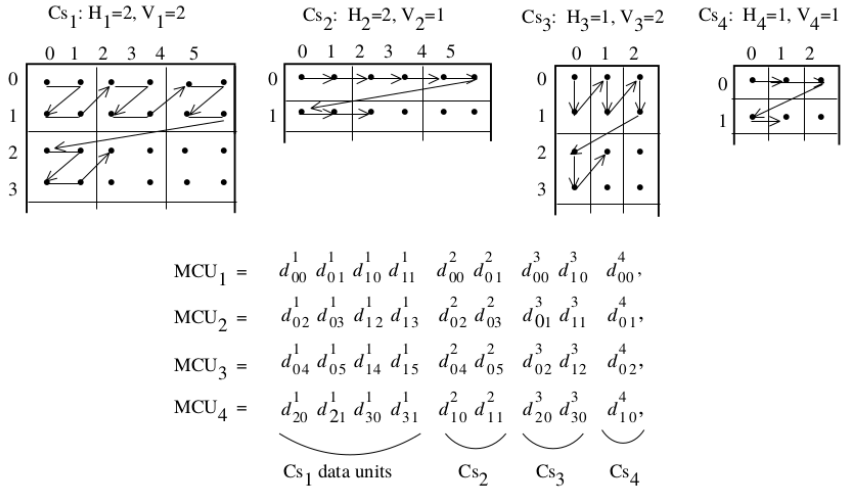

Figura 2 Esempio di ordinamento per formati diversi di chroma subsampling.

4 Requisiti di Progetto

I requisiti dell'encoder jpeg oggetto dello studio sono stati individuati nelle fasi preliminari di questa attività e comprendono:

- Compatibilità con i formati supportati dalle API di Alkeria.
- Portabilità dell'encoder jpeg in ambiente Linux e Windows.
- Supporto dell'encoding jpeg baseline (come specificato in ISO/IEC 10918-1) con tabelle di quantizzazione e tabelle di Huffman definibili dall'utente.
- Tempo di codifica inferiore a 3 volte il tempo di codifica ottenuto dalla libreria Fastvideo (formalmente, *encoding_time* < *3 * fastvideo_encoding_time*).

La misura di *encoding_time* comprende:

- la conversione dal formato in ingresso a YCbCr (4:2:0 e 4:4:4), standard per i file JFIF;
- il tempo di codifica dell'immagine in formato YCbCr in bytestream.

Si considera come *transfer_time* il tempo di trasferimento bidirezionale dei dati tra la memoria host e la memoria device. In particolare, si definisce come *h2d_transfer_time* la latenza di una memcopy sincrona da host memory a device memory e, analogalmente, si definisce come *d2h_transfer_time* la latenza di una memcopy sincrona da device memory a host memory.

Il codec deve poter supportare i seguenti formati in ingresso:

- RGB 24 bit
- BGR 24 bit
- YUV 4:2:2 16 bit
- Grayscale 8 bit
- Grayscale 16 bit[1].

Il colorspace usato internamente nel file JFIF è YCbCr, per cui è prevista una conversione del colorspace in ingresso a YCbCr. Il codec supporta i campionamenti nei formati 4:2:0 e 4:4:4. Le immagini Grayscale vengono codificate sfruttando la componente Y e specificando nel file JFIF l'assenza delle altre componenti, come previsto dallo standard.

Nelle prove da realizzare nel benchmark devono essere fornite: per l'immagine in ingresso, le dimensioni, il formato (colorspace) e la risoluzione; per l'encoder il parametro di compressione *quality factor*.

[1] Lo Standard JPEG definisce un enconding a 8 bit (o meno supportato, 12 bit). Può essere necessaria una conversine a 8 bit.

5 Librerie Software

Le librerie software utilizzate in questa analisi si dividono in librerie che eseguono l'algoritmo di compressione JPEG sfruttando la CPU e librerie che effettuano la compressione delegandone l'esecuzione alla GPU. Per quest'ultime, prima di poter eseguire la compressione, è necessario effettuare una copia dei dati dell'immagine in ingresso dalla memoria host in un'area preventivamente allocata nella memoria device. Al termine della compressione è necessario eseguire un ulteriore trasferimento dalla memoria device alla memoria host. Nonostante la latenza introdotta dal trasferimento bidirezionale dei dati, la capacità della GPU di parallelizzare le istruzioni, permette di raggiungere elevate prestazioni in termini di tempo di esecuzione complessivo.

L'efficacia dell'implementazione di un algoritmo su GPU è legata alla possibilità di parallelizzarne le operazioni (di conseguenza riducendo il numero di operazioni seriali). Nel caso della compressione JPEG, tuttavia, alcune operazioni non sono direttamente parallelizzabili e costituiscono le cause principali della degradazione delle prestazioni.

La seconda fonte di latenze di un'implementazione su GPU è l'accesso alla memoria device. In modo analogo alla cache di una CPU, anche la memoria della GPU è organizzata in modo gerarchico, con memorie accessibili con latenze minori di altre. Anche la modalità di esecuzione cambia, in particolare nell'esecuzione delle istruzioni di branching. Tutte questi aspetti devono essere tenuti in considerazione per ottenere il massimo delle prestazioni.

La maggior parte degli sforzi compiuti dagli sviluppatori di librerie di compressione JPEG eseguibili su GPU sono perciò diretti alla riduzione delle operazioni seriali e all'organizzazione della memoria device al fine di sfruttarne le caratteristiche di accesso in modo da ridurre la latenza di lettura/scrittura (*memory coalescing*).

Di seguito verranno presentate le librerie di compressione su CPU e GPU utilizzate durante questa attività. Le librerie di compressione CPU sono state considerate per fornire un livello minimo con cui comparare le librerie GPU-accelerated e per identificare le dimensioni minime del file di ingresso che giustificano il trasferimento verso la GPU.

5.1 Libjpeg (CPU)

Si tratta dell'implementazione standard di riferimento per un codec JPEG implementata in linguaggio C e distribuita come free software con licenza permissive BSD-like.

La maggior parte dei programmi di visualizzazione e manipolazione immagini richiamano le funzioni presenti in questa libreria o nella sua variante libjpeg-turbo (imagemagick, gstreamer, ffmpeg, gimp, etc.).

Sulle piattaforme Jetson Nano e Jetson TX2, come parte della L4T Multimedia API Reference è presente una versione modificata di *libjpeg* con l'estensione TEGRA ACCELERATE che accelera le operazioni di encoding richiamando i metodi della classe *NvJpegEncoder,* malgrado al momento supporti solo YCbCr 420.

5.2 Libjpeg-turbo (CPU)

Librerie compressione CPU libjpeg e variante libjpeg-turbo con istruzioni Single Instruction Multiple Data (SIMD). Mantiene la ABI retrocompatibilità con libjpeg-v6b. Utilizzata come libreria di default dalle distribuzioni GNU/Linux: Fedora, Debian, Ubuntu, openSUSE, ecc.

5.3 Nvidia nvjpeg-10.2 (GPU)

Sebbene presentata sul sito Nvidia come una libreria di JPEG decoding, nvjpeg supporta funzioni di encoding JPEG ibrido CPU/GPU, cioè con alcune funzioni implementate dalla GPU e altre dalla CPU.

La libreria è parte di CUDA Toolkit dalla versione 10.0 e in questo progetto è stata utilizzata la versione presente in CUDA Toolkit 10.2 compatibile con l'installazione GTX 1650. *È stato osservato*

che su Jetson Nano e Jetson TX2 la versione di nvjpeg è in realtà un wrapper della L4T Multimedia API.

Nvjpeg supporta un encoding JPEG baseline e progressive, conversione da RGB, BGR, RGBI, BGRI, Grayscale e YUV, tabelle di quantizzazione a 8 e a 16 bit e i formati di chroma subsampling: 4:4:4, 4:2:2, 4:2:0, 4:4:0, 4:1:1 e 4:1:0 per le componenti YCbCr.

Comprende varie funzioni di manipolazione dell'immagine (*translation*, *zoom*, *scale*, *crop*, *flip*, ecc.) e funzioni per gestire batch di immagini.

È possibile controllare il processo di encoding usando i parametri previsti dalle funzioni di API, tra le quali anche funzioni per impostare un memory allocator *definibile dall'utente per allocare buffer temporanei usati durante il processo di encoding.*

5.4 Nvidia NPP (GPU)

Nvidia NPP (Nvidia Performance Primitives) è una libreria di image, video e signal processing con funzioni ottimizzate per esecuzione su GPU. Comprende funzioni per la conversione di colorspace, funzioni di compressione e altre funzioni di filtraggio utili per l'elaborazione di immagini. Ampiamente usata in varie applicazioni, ha un'ampia (sebbene a volte concisa) documentazione ed un buon supporto dalla *community*. Estremamente ricca di funzionalità e flessibile, permette di sviluppare in tempi ridotti ed analizzare possibili soluzioni prima di effettuare il tuning di particolari sezioni del programma di cui si vogliono aumentare le prestazioni. Essendo, tuttavia, una libreria ottimizzata per GPU è particolarmente complesso ottenere degli incrementi rilevanti delle prestazioni.

Nvidia NPP è stata utilizzata in questo progetto per implementare le varie fasi della compressione jpeg. Al contrario di nvjpeg che racchiude le fasi di codifica in una sola funzione, NPP permette di suddividere il processo di codifica in modo da analizzare le latenze ed identificare i principali *bottleneck*.

5.5 Fastvideo SDK (GPU)

Fastvideo SDK è una libreria commerciale per image e video processing. Per testarne le potenzialità vengono forniti delle applicazioni demo per Windows e Linux. Le applicazioni demo per Windows vengono distribuite come precompilate e non sono perciò modificabili, mentre le applicazioni demo per Linux vengono distribuite con il codice sorgente relativo all'applicazione demo, un Makefile e le librerie sdk da linkare.

Studiando il codice sorgente dell'applicazione demo, si nota come Fastvideo allochi i buffer temporanei e le strutture dati relative all'encoder in una fase di inizializzazione e deallocati in una fase di *cleanup* le cui latenze non sono considerate dai timer che Fastvideo utilizza per misurare le performance della libreria. Inoltre, come anche riportato dagli sviluppatori di Fastvideo, le latenze riportate nel benchmark non comprendono i tempi di trasferimento dalla memoria host alla memoria device e viceversa.

Time and performance measurements for Fastvideo SDK modules on NVIDIA GPUs for grayscale and color images don't take into account host I/O latency (image loading to RAM from HDD/SSD and saving back). We have presented timings for computations on GPU only. As soon as any image processing pipeline consists of series of such algorithms (SDK modules), it's a reasonable approach to measure only computation time, assuming that initial and final images reside in GPU memory.

[Fonte: [4]]

Fastvideo implementa lo standard di compressione JPEG baseline per immagini grayscale a 8 bit e immagini a colori a 24 bit. Fastvideo supporta formati standard di ingresso PGM, YUV, PPM e BMP, con una modifica del codice sorgente dell'applicazione demo per Linux da noi apportata per questo progetto, è anche possible fornire in ingresso i dati raw decompressi e supportare, quindi, altri formati.

Fastvideo supporta i formati di chroma subsampling 4:4:4, 4:2:2 e 4:2:0. È compatibile con ffmpeg e può essere integrata con OpenGL.

Come riportato da Fastvideo, le operazioni compiute dalla libreria durante una compressione JPEG comprendono le operazioni definite dallo standard JPEG baseline: Input data parsing, Color Transform, 2D DCT, Quantization, Zig-zag, AC/DC, DPCM, RLE, Huffman coding, byte stuffing, JFIF formatting. Gli sviluppatori sostengono di essere riusciti a parallelizzare tutte le operazioni di codifica incluso l'entropy encoding che viene spesso implementato in modo seriale su CPU. L'implementazione parallela della codifica di Huffman non è chiaramente disponibile perciò, in questa attività, si è provato ad analizzare un possibile approccio per rendere parallela la codifica di Huffman sfruttando i Restart Markers previsti dallo standard JPEG. Riteniamo che questa soluzione possa però avere dei problemi di compatibilità con i decoder più semplici che non supportano i Restart Markers. In questa attività le prove effettuate sono state effettuate utilizzando come decoder libjpeg che supporta i Restart Markers e si è dimostrato che, almeno in linea di principio, l'utilizzo dei Restart Markers possa essere la soluzione per implementare in modo parallelo e completamente in GPU l'ultima fase di codifica in bytestream.
Le misure riportate da Fastvideo per i benchmark di encoding JPEG riguardano i tempi di esecuzione della sola compressione JPEG su Nvidia GeForce GTX 1080 TI (Nvidia Pascal 3584 Cuda Core), Nvidia Quadro P6000 (Nvidia Pascal 3840 Cuda Core), Jetson Nano e Jetson TX2. In questa attività abbiamo eseguito le applicazioni demo.

6 Piattaforme Hardware

6.1 Nvidia GeForce GTX 1650

Architettura

Turing TU117, 896 core, 14 Streaming Multiprocessors (64 Streaming Processors per SM), Base Clock 1485 MHz (Boosted Clock 1665 Mhz).

Memoria

4GB GDDR5 8Gbps bus 128bit double slot PCIe, L1 Cache 64 KB (per SM) L2 Cache 1024 KB.

Performance

Pixel Rate 53.28 Gpixel/s, FP16 5.967 TFLOPS, FP32 (float) 2.984 TFLOPS.

Supporto Librerie

CUDA 7.5, DirectX 12, OpenGL 4.6, OpenCL 1.2, Vulkan 1.2.131.

Descrizione architettura Turing

L'architettura Turing è l'evoluzione dell'architettura Pascal. Ha migliori prestazioni rispetto a Pascal [5], pur mantenendo la retrocompatibilità del codice. La *major version* dell'architettura dello Streaming Multiprocessor (SM) Turing è la stessa di Volta. Nell'architettura Turing, ogni SM comprende 4 warp scheduler che gestiscono un insieme statico di warp assegnando l'esecuzione delle istruzioni a specifiche unità di calcolo. Istruzioni indipendenti vengono emesse ad ogni ciclo di clock, mentre sono necessari due cicli di clock per l'emissione di istruzioni dipendenti. Diversamente dall'architettura Pascal, in cui unità FMA (Fused Multiply and Add) completano l'esecuzione di un'istruzione in 6 cicli di clock, per le FMA dell'architettura Turing sono sufficienti 4 cicli di clock. In un SM Turing sono presenti: 64 core FP32 (Floating Point a 32 bit), 2 core FP64, 64 core INT32 e 8 Tensor Cores. Essendoci unità FP32 e INT32 indipendenti, è perciò possibile l'esecuzione contemporanea di istruzioni FP32 e INT32.

Su architetture Turing è possibile l'esecuzione concorrente di un massimo di 32 warps per SM, rispetto ai 64 possibili su Volta e un massimo di 16 thread blocks per SM.

L'architettura Turing utilizza l'Independent Thread Scheduling introdotto dall'architettura Volta per abilitare la sincronizzazione tra warp. Questo tipo di scheduling, può richiedere delle modifiche del codice se questo è stato scritto facendo delle assuzioni sulla sincronizzazione dei warp validi per achitetture hardware antecedenti Turing. In particolare, le applicazioni che utilizzano l'istruzione *__synchthreads()* devono garantire che tutti i thread non-exited eseguano l'istruzione. Questa ed altre problematiche relative alla sincronizzazione di thread nel warp possono essere identificate e corrette dai tool di cuda-memcheck *racecheck* e *synccheck*.

Su architettura Turing è possibile eseguire operazioni di multiply and accumulate su matrici sfruttando apposite unità di calcolo (tensor core). Le API di CUDA 10 riflettono questa opzione, esponendo operazioni specifiche di load, store e multiply-accumulate per matrici. Per utilizzare queste istruzioni si suddivide una matrice di dimensioni arbitrarie in sottomatrici (matrix fragment) di specifiche dimensioni variabili a seconda del tipo di dati di input.

Nell'architettura Turing viene riproposto il modello Unified Shared Memory L1 Texture Cache già introdotto da Volta. La Unified Shared Memory combina le funzionalità della cache L1 e della Texture cache in un modello unico che mantiene una porzione di memoria contigua pre-caricata prima di renderla accessibile ai thread del warp. La Unified Shared Memory è di 96KB di cui una porzione può essere dedicata ad uso cache L1 (*carveout*). Turing supporta due configurazioni: 64KB di shared memory e 32KB di cache L1, oppure 32KB di shared memory e 64KB di cache L1. In Turing un singolo thread può accede all'intera shared memory da 64KB, ma sono possibili allocazioni statiche di massimo 48KB per garantire retrocompatibilità e viene introdotta una funzione per abilitare

l'allocazione dinamica di aree di maggiori dimensioni.

6.2 Jetson TX2 (Chipset Tegra X2)

Architettura

Big.little Dual-core NVIDIA Denver2 e Quad-core ARM Cortex-A57, Pascal 256 core CUDA 6.1.

Memoria

8GB LPDDR4 bus 128bit PCIe 2.0, 32GB eMMC flash memory.

Periferiche

HDMI 2.0, 802.11a/b/g/n/ac 2×2 867Mbps WiFi, Bluetooth 4.1, USB3, USB2, 10/100/1000 BASE-T Ethernet, 12 lanes MIPI CSI 2.0, 2.5 Gb/sec per lane, PCIe 2.0, SATA, SDcard, dual CAN bus, UART, SPI, I2C, I2S, GPIO.

Performance

Pixel Rate 53.28 Gpixel/s, FP16 5.967 TFLOPS, FP32 (float) 2.984 TFLOPS.

Supporto Librerie

JetPack 4.2.2, Linux4Tegra R32.2.1 (L4T), CUDA Toolkit 10.0.326, L4T Multimedia API (Argus 0.97), GStreamer 1.14.1, cuDNN 7.5.0, TensorRT 5.1.6, TensorFlow 1.14.0, OpenCV 3.3.1, OpenGL 4.6 / OpenGL ES 3.2.5, Vulkan 1.1.1.

6.3 Jetson Nano (Chipset Tegra X1)

Architettura

Quad-core ARM Cortex-A57, Architettura Maxwell 128-core CUDA 5.3

Memoria

4GB LPDDR4, 16GB eMMC

Performance

H.264/H.265 encoder & decoder, Dual ISPs (Image Service Processors).

Periferiche

HDMI 2.0, 802.11ac WiFi, Bluetooth 4.0, USB3, USB2, Gigabit Ethernet, 12 lanes MIPI CSI 2.0, 4 lanes PCIe gen 2.0, SATA, 2x SDcard, 3x UART, 3x SPI, 4x I2C.

Supporto Librerie

JetPack 4.2.2, Linux4Tegra R32.2.1 (L4T), CUDA Toolkit 10.0.326, cuDNN 7.5.0, TensorRT 5.1.6, TensorFlow 1.14.0, VisionWorks 1.6, OpenCV 3.3.1, OpenGL 4.6 / OpenGL ES 3.2.5, Vulkan 1.1.1, L4T Multimedia API (Argus 0.97), GStreamer 1.14.1.

7 Articoli e Pubblicazioni Rilevanti

In questa sezione vengono proposti alcuni articoli rilevanti per l'attività.

7.1 Accelerating the pre-processing stages of JPEG encoder on a heterogenous system using OpenCL

In [6] gli autori descrivono un'implementazione della conversione di colorspace e chroma subsampling per OpenCL (Open Computing Language) su un sistema ibrido CPU multicore - GPU. Gli autori riportano una riduzione dei tempi di esecuzione di un fattore 8.78 rispetto ad

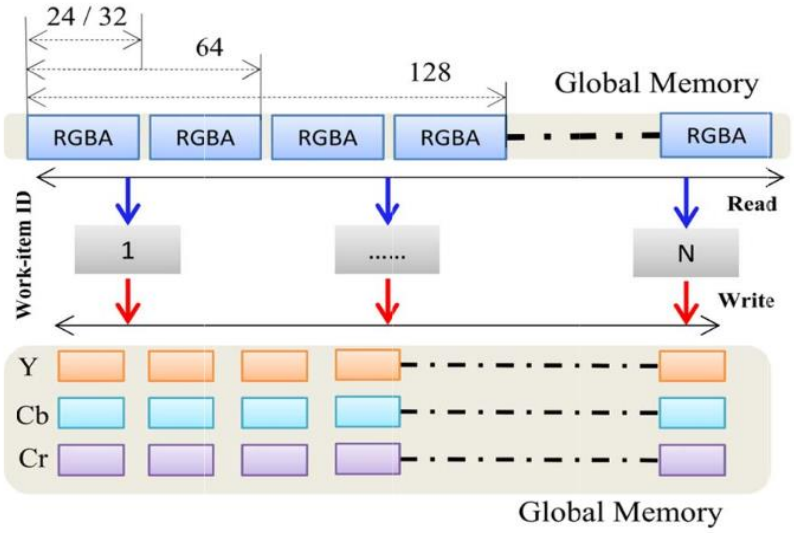

Figura 3 Organizzazione della memoria e mappatura dei work-item per conversione colorspace RGBA a YCbCr. Si evidenziano i diversi tipi di allineamento a 32, 64 e 128 bit. Fonte: [6].

un'implementazione seriale su CPU. Nell'articolo viene studiato il problema di ridurre la latenza della conversione di colorspace da RGB a YCbCr e del chroma subsampling dal formato 4:4:4 ai formati 4:2:0 e 4:2:2. Complessivamente queste fasi di pre-processing costituiscono circa il 40% del tempo di esecuzione totale di una compressione JPEG. Gli autori descrivono un'implementazione per OpenCL in cui l'accesso in memoria viene ottimizzato assegnando un numero opportuno di pixel da elaborare ad ogni thread. Nella fase di conversione del colorspace ogni pixel può essere elaborato in modo indipendente, ma invece di assegnare un thread ad ogni singolo pixel, risulta più efficiente elaborare più pixel per thread in modo da avere accessi in memoria allineati (si veda Figura 3).

In CUDA già con l'introduzione di architetture con CUDA compute capability 2.0, l'effetto di accessi non allineati viene mitigato dalla presenza di una cache L1 con line cache da 128 bit. In questo tipo di architetture gli accessi sequenziali sono combinati (*coalesced*). Il problema di accessi non ottimizzati in memoria resta per gli accessi *strided* tipici dell'indirizzamento delle diverse righe dell'immagine.

7.2 Parallel program design for JPEG compression encoding

In [7] viene presentata un'architettura software per l'encoding JPEG parallelo per API Nvidia CUDA. Gli autori riportano un miglioramento sostanziale (20-24x) delle prestazioni in confronto ad un'implementazione seriale su CPU single core.

L'algoritmo presentato assegna alla CPU il compito di eseguire la conversione di colorspace e di chroma subsampling. La GPU viene utilizzata per la DCT e la quantizzazione dei blocchi 8x8, assegnando 64 thread (8x8) per ogni blocco organizzati in una CUDA grid di dimensioni *image_width*/8 *image_height*/8 CUDA blocks. L'enconding delle componenti DC e AC vengono eseguite dalla CPU. Per mascherare la latenza dei trasferimenti in memoria, mentre la GPU è occupata

nell'esecuzione della DCT sulle componenti Cb e Cr, la CPU può iniziare l'encoding delle componenti DC e AC. Nell'articolo vengono alcune possibili ottimizzazioni: sostituire gli operatori ordinari (+, *, /) con le rispettive funzioni built-in di CUDA (*fadd*, *fmul* e *fdiv*); utilizzare la *constant memory* per le matrici di quantizzazione; sfruttare aree di memoria allocate con *cudaMallocHost()* per garantire la presenza della memoria allocata in memoria fisica (non rilocabile) e assicurarsi di avere accessi allineati ottenibili con *cudaMallocPitch()* e *cudaMemvpy2D()*.

Con il supporto della libreria NPP, l'implementazione proposta nell'articolo può essere estesa per includere tra le operazioni eseguite dalla GPU anche la conversione di colorspace e l'encoding di Huffman.

7.3 A Performance Model of Fast 2D-DCT Parallel JPEG Encoding Using CUDA GPU and SMP-Architecture

In [8] vengono comparate le prestazioni di un'implementazione di un codec JPEG su un sistema multiprocessore (Symmetric Multiprocessor, SMP) e su GPU NVIDIA GeForce

GT430 con 96 CUDA core. Gli autori dimostrano che un sistema SMP ha prestazioni migliori di un'implementazione su GPU, sia per quanto riguarda il fattore di utilizzo dei core, sia per i tempi di esecuzione. L'algoritmo di compressione presentato sfrutta l'implementazione della fast DCT presente anche in libjpeg e l'encoding di Huffman. Nell'analisi, gli autori considerano i tempi di trasferimento da e verso la GPU, non considerano un'implementazione che sfrutti la shared memory della GPU e includono i tempi di inizializzazione dell'ambiente CUDA. L'implementazione JPEG parallela presentata nell'articolo è applicabile ad architetture parallele di tipo SMP che condividono la stessa memoria, al contrario di un'implementazione che sfrutti la GPU come co-processore per la compressione incorrendo nei tempi di trasferimento sul bus PCI. Architetture multiprocessore possono beneficiare del parallelismo di alcune delle fasi della compressione JPEG e di minori latenze da e verso la memoria, impiegando però risorse computazionali sfruttabili per altri task.

7.4 Fast approximate DCT with GPU implementation for image compression

In [9], gli autori descrivono un metodo di approssimazione della DCT su un blocco 8x8 implementabile con sole 17 operazioni di addizione. Con l'approssimazione proposta, denominata PADCT (Proposed Approximate DCT), si possono ridurre i tempi di esecuzione della DCT senza degradare eccessivamente l'immagine di output.

Nell'articolo viene riportata una possibile implementazione su GPU che sfrutta le API Nvidia CUDA analizzata di seguito. L'analisi comparativa descritta nell'articolo presenta le maggiori differenze in termini di latenza e di qualità di immagine di output tra metodi di approssimazione simili derivanti dalla Signed DCT.

La PADCT è formalmente definita dalla trasformazione

$$F = (D_p T_p) X (T_p D_p)^T = D_p (T_p X T_p^T) D_p^T,$$

con X matrice 8x8 di input, T_p matrice con elementi 0, +1 e -1 e D_p matrice diagonale usata come coefficiente di normalizzazione. La matrice F descrive una serie di somme e differenze tra solo alcuni degli elementi del blocco 8x8 e costituisce le 17 operazioni indipendenti della PADCT.

Nell'articolo, l'effetto della sostituzione della DCT con la PADCT è stato analizzato su immagini 256x256 grayscale a 8-bit e riportato graficamente in Figura 4. Essendo la PADCT un'approssimazione della DCT può essere sostituita in un codec JPEG senza incorrere in un'incorretta decodifica, tuttavia l'approssimazione introduce un errore in alcuni casi non trascurabile per particolari applicazioni. Ad esempio, in Figura 4 si possono notare alcuni artefatti (spalle e oggetti sullo sfondo) che possono risultare indesiderati in particolari applicazioni.

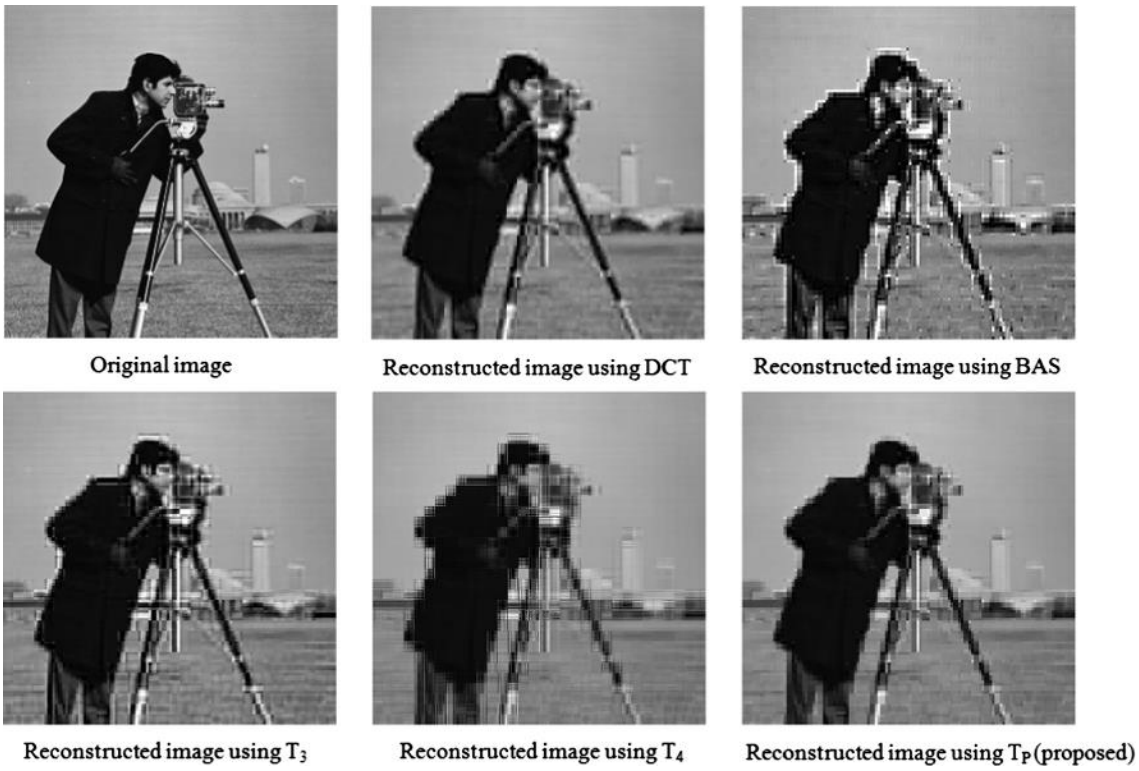

Figura 4 Comparazione di differenti approssimazioni della DCT per un'immagine grayscale 256x256. Fonte: [9].

I tempi di esecuzione della trasformata PADCT sono stati misurati su CPU (Intel i5-4200M 2.50 GHz) e su GPU (GeForce GT 720M) con CUDA Toolkit 6 per immagini grayscale di dimensioni 256x256, 512x512, 1024x1024, 2048x2048 e 4096x4096. I risultati sono riportati in Tabella 1. Si nota come la sostituzione della PADCT alla DCT permetta di ridurre i tempi di esecuzione dell'80% su CPU e del 90% su GPU.

Running times (ms) for DCT and PADCT on CPU and GPU.

Image sizes	DCT on CPU	DCT on GPU	PADCT on CPU	PADCT on GPU
4096 × 4096	3322.949	167.002	715.571	17.990
2048 × 2048	828.826	46.591	186.111	4.571
1024 × 1024	256.378	11.854	44.542	1.264
512 × 512	53.265	3.094	10.553	0.322
256 × 256	12.996	0.875	2.485	0.086

Tabella 1 Tempi di esecuzione per DCT e PADCT con implementazione per CPU e GPU.

L'implementazione per GPU proposta nell'articolo prevede la divisione dell'immagine in input in blocchi di dimensione 8x8 processati in maniera indipendente dai core della GPU in modo da sfruttarne il parallelismo. La trasformata 2D viene eseguita da 8 thread che applicano una trasformata 1D prima lungo le colonne della matrice e poi lungo le righe (sfruttando l'associatività della moltiplicazione tra matrici). Come suggerito nel documento [10], per ottenere le massime prestazioni dalla GPU, è consigliabile selezionare il numero di thread per blocco come un multiplo della dimensione del wrap (32 nel caso di architettura GeForce 8x), pertanto sono stati assegnati 32 thread (1 wrap) per ogni CUDA blocks. Di conseguenza ogni CUDA block (insieme di thread) è responsabile dell'esecuzione della trasformata su 4 matrici 8x8 dell'immagine in ingresso (si veda Figura 5).

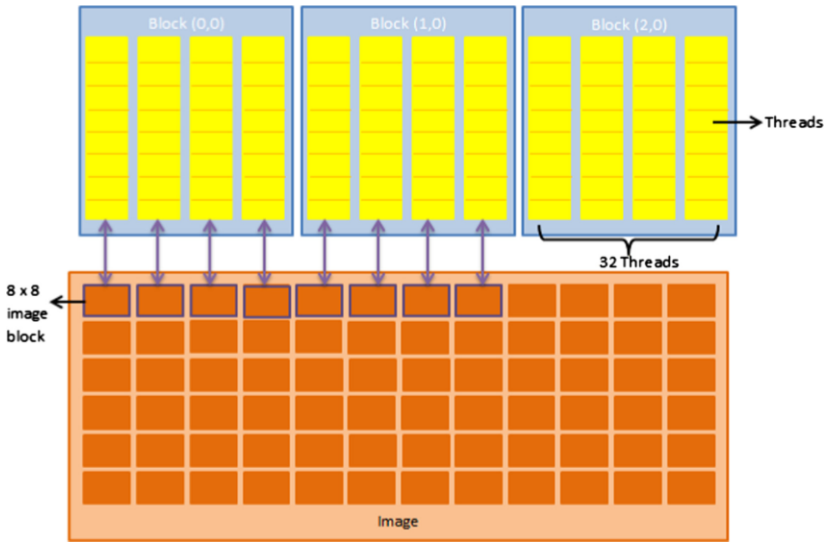

Figura 5 Mappatura dei CUDA thread per l'esecuzione della DCT su image block 8x8. Fonte: [9].

7.5 Comparing CNNs and JPEG for Real-Time Multi-view Streaming in Tele-Immersive Scenarios

Abbiamo deciso di includere questo articolo per presentare un approccio alternativo alla codifica JPEG classica basato su Auto Encoder implementati come Convolutional Neural Network (CNN), Recurrent Neural Network (RNN) o Generative Adversarial Network (GAN). Un Auto Encoder viene allenato su un dataset per realizzare una funzione di compressione da uno spazio con un numero elevato di dimensioni (un'immagine in RGB, ad esempio) ad uno spazio di variabili latenti con dimensioni di molto inferiori alle dimensioni di input. L'assunzione di questi modelli è quella di considerare le immagini presenti in input come vettori in un sottospazio dell'intero spazio di combinazioni possibili. In particolare, si assume che immagini appartenenti a categorie simili (ad esempio immagini di volti umani) abbiano caratteristiche comuni e identificare queste caratteristiche (*features*) permette di comprimere informazioni ridondanti e raggiungere alti fattori di compressione.

In [11] vengono comparate le curve di compressione-distorsione (*rate-distortion*) e i tempi di esecuzione per gli algoritmi di compressione classici (JPEG e JPEG2000) e algoritmi di compressione basati su reti neurali. Nonostante questi ultimi abbiano prestazioni migliori in termini di fattore di compressione, hanno tempi di esecuzione molto maggiori degli algoritmi di compressione classici (10x del tempo di esecuzione di compressione JPEG su CPU). Questo tipo di approccio non è pertanto consigliato per applicazioni a bassa latenza e alto throughput. Il vantaggio principale nell'utilizzo di approcci basati su reti neurali è la possibiltà di estenderli per realizzare funzioni ad ora non supportate da algoritmi classici come mutli-*scale prediction* per streaming adattativo.

8 Benchmark

Il benchmark è stato realizzato sulla base delle osservazioni compiute sulla libreria Fastvideo.

In particolare, viene considerata come metrica comparativa con Fastvideo il solo tempo di encoding escludendo il tempo di trasferimento dalla memoria host alla memoria device e viceversa.

I tempi di allocazione, inizializzazione e deallocazione della memoria vengono esclusi dall'analsi.

In modo analogo all'implementazione dei programmi di test di Fastvideo, è prevista l'allocazione di un buffer destinato a contenere l'immagine di dimensioni massime dell'insieme di immagini in ingresso, tale buffer verrà utilizzato anche per tutte le altre immagini di dimensioni inferiori o uguali.

Per quest'attività sono stati sviluppati i programmi *npp_jpgenc* e *nvjpeg_jpgenc* che richiamano rispettivamente le funzioni della libreria Nvidia NPP e Nvidia NVJPEG. Su Jetson Nano e Jetson TX2 la libreria NVJPEG, presente nella Jetson Multimedia API, ha un'interfaccia diversa dalla libreria con lo stesso nome disponibile in CUDA Toolkit 10 e le due librerie non sono compatibili.

Come da requisiti, il programma *npp_jpgenc* supporta i seguenti formati per le immagini in ingresso:

- RGB 24 bit
- BGR 24 bit
- YUV 4:2:2 16 bit
- Grayscale 8 bit

Mentre *nvjpeg_jpegenc* supporta solo RGB 24 bit e Grayscale a 8 bit.

I programmi di benchmark disponibili come applicazioni demo da Fastvideo sono stati lasciati inalterati. Poiché queste applicazioni non supportano tutti i formati richiesti da Alkeria, si propone la comparazione dei tempi di esecuzione per il solo formato RGB.

I tempi di esecuzione degli altri formati sono invece riportati per *npp_jpgenc* e *nvjpeg_jpegenc*.

Con *npp_jpgenc* è stato possibile misurare anche i tempi di esecuzione delle varie fasi della codifica JPEG, in particolare sono stati misurati i tempi della conversione in colorspace YCbCr, dell'esecuzione della DCT quantizzata e della codifica di Huffman. Le altre librerie non forniscono la possibilità di analizzare in modo isolato queste fasi dell'algoritmo di codifica, percui per queste librerie viene riportato il tempo di encoding.

Nelle prove realizzate nel benchmark sono state specificate per l'immagine in ingresso: le dimensioni, il formato (colorspace) e la risoluzione, vengono anche riportate le dimensioni del file non compresso e del file compresso; mentre per l'encoder viene specificato come parametro di compressione il *quality factor*. Come ulteriore analisi viene riportato l'effetto del *jpeg quality factor* sull'*encoding_time.*

Al fine di fornire un livello minimo di prestazioni vengono considerati i tempi di esecuzione dell'encoding JPEG su CPU dalla libreria *libjpeg* (o la variante SIMD-accelerated *libjpeg-turbo*) richiamata dal programma *imagemagick.*

Come immagine in ingresso viene fornita un'immagine random generata da una distribuzione uniforme nell'intervallo [0, 255] sui tre canali con seed definito al fine di permettere la riproducibilità dei risultati, generabile dal programma *imgen.py* incluso nei tool del benchmark. Si osserva che un'immagine random RGB è caratterizzata da bassa correlazione e può essere considerata come un limite superiore alle performance effettive dell'algoritmo di compressione che assume, invece, un'alta correlazione tra pixel spazialmente vicini. In Figura 6 viene riportato un grafico che evidenzia i differenti tempi di codifica per un'immagine naturale (non sintetica) ed un'immagine sintetica al

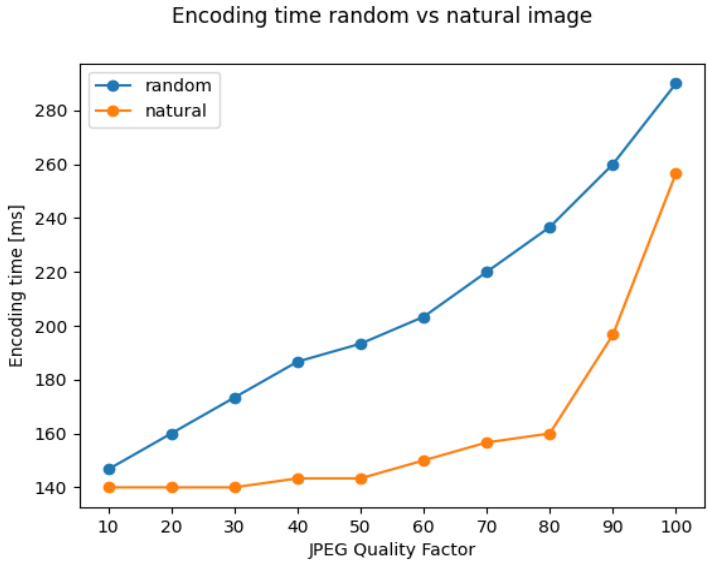

Figura 6 Tempi di esecuzione di libjpeg-turbo per immagini sintetiche random ed immagini non sintetiche. Le immagini sintetiche random hanno tempi di esecuzione più alti a causa della minor correlazione spaziale.

variare del JPEG quality factor. Dal grafico si nota che i tempi di codifica dell'immagine random sono sempre maggiori rispetto ai tempi di codifica di un'immagine non sintetica. Pertanto, è valida l'assunzione di considerare i tempi di compressione di un'immagine sintetica come limite superiore ai tempi di codifica di un'immagine non sintetica.

Nel grafico in Figura 7 sono stati riportati, per le varie librerie, i tempi di esecuzione del processo di encoding compreso il tempo di esecuzione necessario per la conversione a colorspace YCbCr di un'immagine random di 5Megapixel con risoluzione 2560x1960. Dal grafico si nota come le librerie

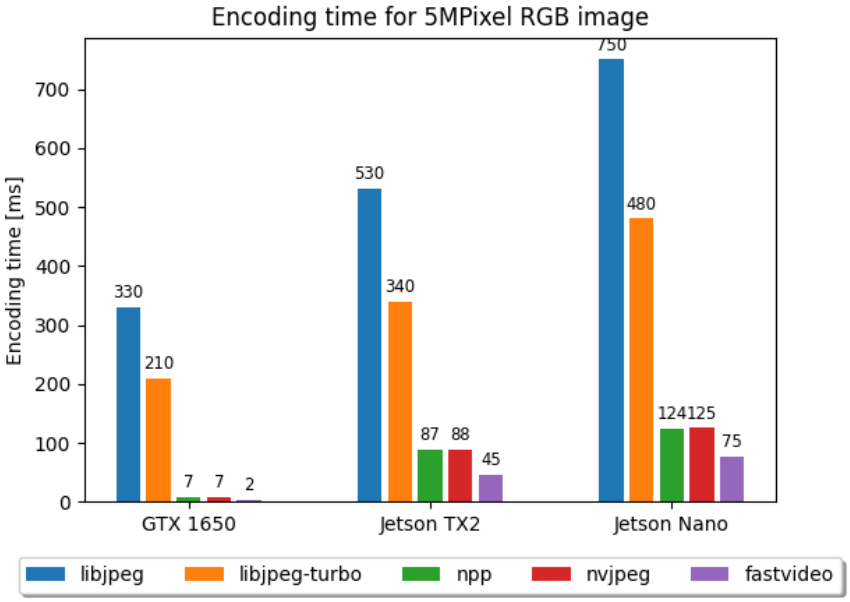

Figura 7 Tempi di esecuzione media dell'encoding JPEG relativi alle librerie analizzate per immagini RGB a 5MPixel con risoluzione 2560x1960.

che sfruttano la GPU (npp, nvjpeg e fastvideo) per accelerare il processo di compressione hanno prestazioni migliori di librerie implementate per CPU. I valori riportati le librerie implementate per GPU devono però essere opportunamente corretti per includere anche il tempo di trasferimento

trasferimento di un'immagine da host a device e il tempo di trasferimento dello bitstream compressa da device a host.

Il tempo di trasferimento di un'immagine di 5Megapixel su un bus PCI Express v2 (500 MB/s per lane) da host a device è di ~7.5 ms; mentre, per via del processo di compressione che produce una bitstream di minori dimensioni, il tempo di trasferimento da device a host della bitstream è inferiore. Per una stima delle dimensioni della bitstream, si possono utilizzare come rifemento i modelli risultanti dall'analisi dell'andamento del compression factor al variare del JPEG quality factor.

Nonostante l'inclusione dei tempi di trasferimento, una compressione JPEG su GPU risulta essere comunque più efficiente di una compressione su CPU, anche se confrontata con librerie che sfruttano istruzioni SIMD (libjpeg-turbo).

Si osserva che il tempo di enconding di fastvideo su GTX 1650 non sembra essere confrontabile con gli altri valori riportati dai test di fastvideo su Jetson TX2 e Jetson Nano. Poiché il codice del programma di benchmark non è disponibile, riteniamo che il valore riportato in output da fastvideo debba essere ulteriormente vagliato.

In Figura 8 si riportano i grafici relativi all'effetto del JPEG *quality factor* sul tempo di encoding e sul compression ratio (definito come il rapporto tra le dimensioni del file compresso e le dimensioni del file originale). Dai grafici presentati si può notare come il tempo di encoding su GPU vari molto meno del tempo di encoding seriale su CPU relativamente al JPEG quality factor. Anche nel caso di

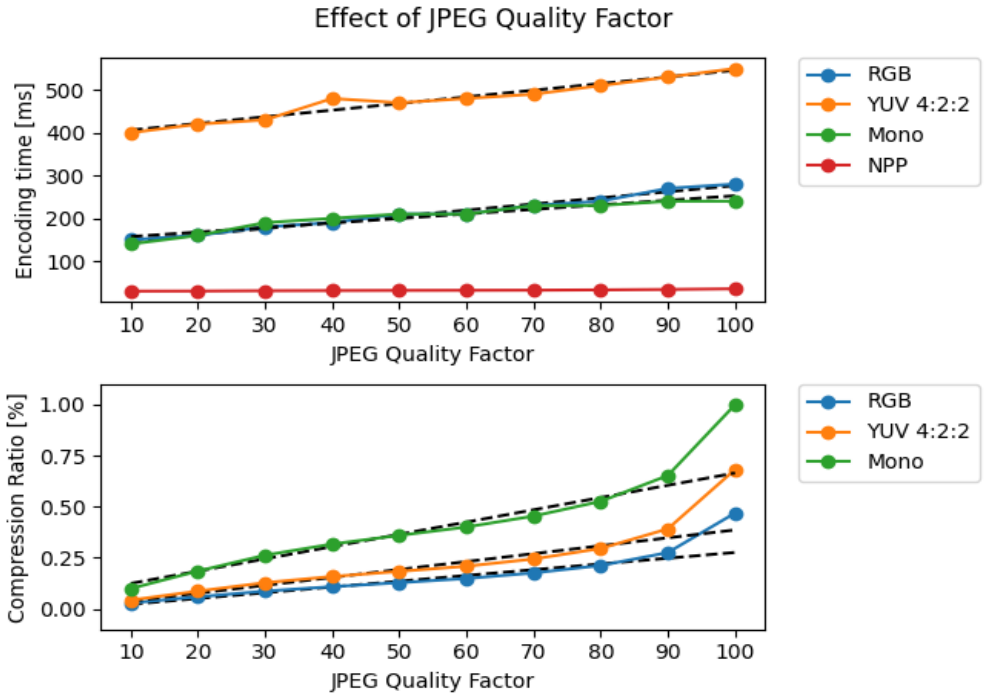

Figura 8 Tempi di compressione e fattore di compressione al variare del JPEG quality factor per diversi formati di input.

compressione su GPU è comunque presente una relazione lineare con il JPEG quality factor che risulta poco visibile dal grafico in figura. Per evidenziare la relazione lineare è possibile isolare le latenze delle varie fasi di compressione tramite l'esecuzione del programma *npp_jpgenc*.

I valori derivati dall'esecuzione del programma di test sono riportati in Figura 9. Come si può notare dal nuovo grafico, il tempo di esecuzione delle fasi di colorspace conversion, chroma subsampling e DCT è praticamente costante su GPU per via della parallelizzazione delle operazioni, mentre la codifica di Huffman ha una dipendenza approssimativamente lineare rispetto al JPEG quality factor. Questa relazione di dipendenza deriva dalla minor lunghezza dei codici di Huffman (e quindi un numero inferiore di scritture in memoria) per bassi JPEG quality factor.

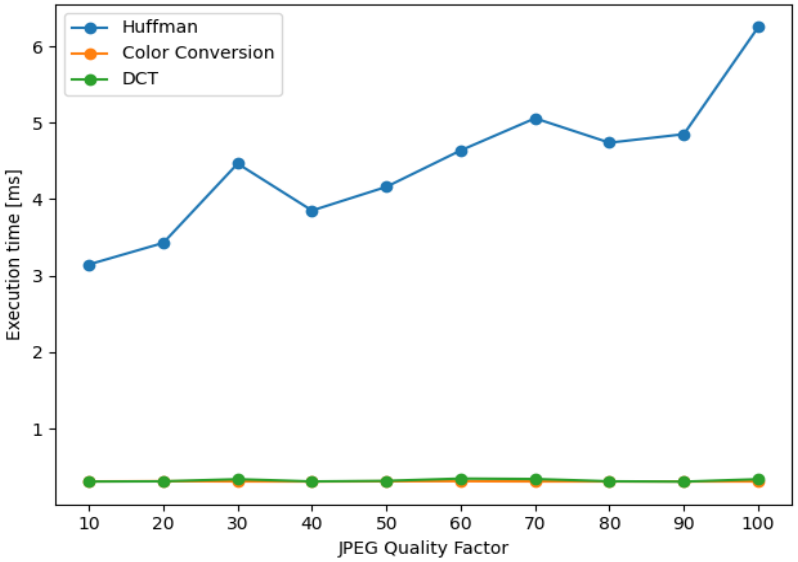

Figura 9 Dettaglio dei tempi di esecuzione per le varie fasi di codifica JPEG al variare del JPEG quality factor. È evidente il costo costo computazionale della codifica di Huffman che rappresenta la fase non parallelizzabile della codifica JPEG.

Nelle prove effettuate con la libreria NPP sono stati notati tempi di encoding molto più alti per la prima esecuzione che per le successive. Per evidenziare il fenomeno di *warmup* della libreria, sono state eseguite 40 compressioni per immagini RGB generate randomicamente con distribuzione uniforme in [0, 255] su tre canali con diverso seed (al fine di escludere l'effetto della cache dall'analisi). I valori ottenuti sono riportati nel grafico in Figura 10, dove la linea rossa rappresenta il valor medio del tempo di encoding escludendo la prima iterazione.

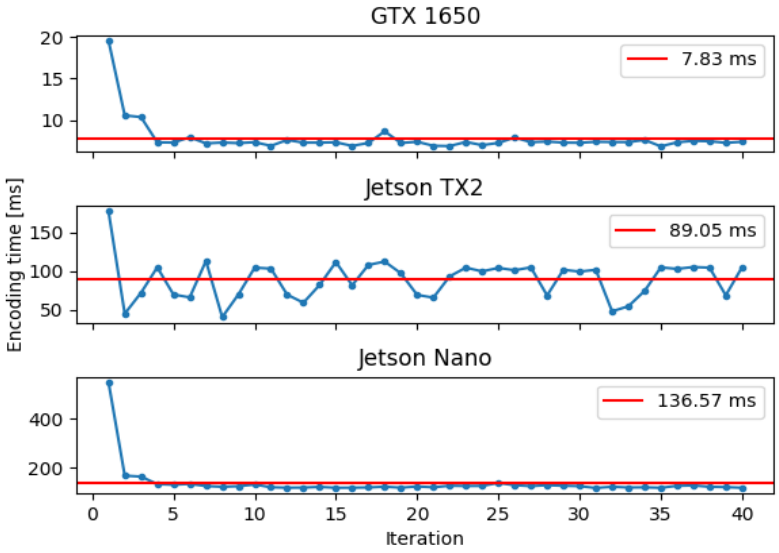

Figura 10 Tempi di encoding di NPP per le piattaforme analizzate. Gli alti valori per la prima iterazione sono dovuti al caricamento delle librerie dinamiche e all'inizializzazione dell'ambiente CUDA.

Il modello relativo al compression ratio è stato utilizzato per stimare le dimensioni della bytestream di output per calcolare il tempo di trasferimento da device a host noti il quality factor e il formato di

ingresso. I modelli per i vari formati sono riportati in Tabella 2.

	Compression Ratio
RGB	2.82e-03*quality-6.37e-03
YUV 4:2:2	3.87e-03*quality-1.58e-03
Mono	6.01e-03*quality+6.39e-02

Tabella 2 Modelli di compressione per diversi formati di immagini di input.

Utilizzando la libreria Nvidia NPP è stato possibile analizzare le latenze delle varie fasi della compressione JPEG. Nel grafico in Figura 11, sono riportate le latenze relative (in termini percentuali) rispetto al tempo di esecuzione totale delle varie fasi di compressione per ogni piattaforma analizzata. Si nota come la maggior parte della latenza è dovuta alla codifica di Huffman che rappresenta la sezione non parallelizzabile della codifica JPEG. Si osserva inoltre un valore di latenza rilevante per la DCT su Jetson TX2 che potrebbe essere dovuto ad una diversa implementazione della trasformata (di default su Jetson TX2 è installata versione 9.0 di CUDA Toolkit), oppure ad accessi non allineati in memoria.

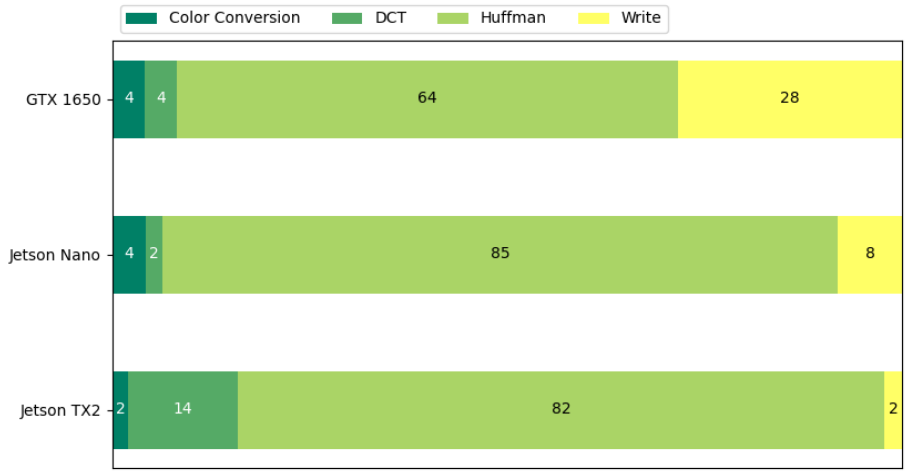

Figura 11 Tempi di esecuzione relativi (in termini percentuali) delle varie fasi di codifica JPEG per le varie piattaforme analizzate.

In Tabella 3 vengono riportati i tempi di esecuzione (in ms) dell'encoding JPEG ottenuti sulle piattaforme analizzate in quest'attività, per immagini random con diversi formati di ingresso. Viene mantenuto costante il *jpeg quality factor* per l'encoder a 90%. I tempi di esecuzione sono relativi rispettivamente alle librerie Nvidia NPP, Nvidia NVJPEG e Fastvideo.

Formato Immagine Input	GTX 1650	Jetson TX2	Jetson Nano
Mono 2K (2048×1080)	2.17 / 3.12 / 0.77	25.61 / 26.56 / 17.71	58.23 / 58.63 / 20.94
RGB 2K (2048×1080)	4.29 / 3.78 / 0.93	50.62 / 51.57 / 34.55	67.12 / 68.35 / 24.41
Mono 5Mpixel (2560×1960)	5.43 / 5.72 / 1.60	64.06 / 64.36 / 43.12	110.30 / 112.30 / 49.67
RGB 5Mpixel (2560×1960)	7.45 / 7.21 / 2.07	87.89 / 88.19 / 45.00	124.20 / 125.00 / 75.40
Mono 4K (3840x2160)	7.55 / 9.41 / 2.24	89.10 / 89.4 / 59.89	165.71 / 163.84 / 57.94
RGB 4K (3840x2160)	13.07 / 11.30 / 2.84	154.23 / 154.53 / 103.54	206.58 / 203.75 / 75.73

Tabella 3 Tabella riassuntiva dei tempi di esecuzione delle librerie NPP, NVJPEG e Fastvideo per diversi formati di immagini di input e piattaforme.

Come si nota dalla Tabella 3, Fastvideo ha prestazioni migliori rispetto alle altre due librerie, in quanto gli sviluppatori sono riusciti a parallelizzare la fase di codifica di Huffman che è il principale collo di bottiglia per le altre due librerie. Tale parallelizzazione è possibile grazie all'introduzione dei Restart Marker previsti per codec di tipo progressive ed in genere utilizzati durante la fase di decodifica per supportare il recupero dell'esecuzione durante il processo in caso di errore. L'uso dei Restart Marker permette di rendere indipendenti le codifiche dei coefficienti DC per un certo numero di blocchi (definibile dal parametro *restart interval*) e, di conseguenza, di parallelizzarne la codifica e migliorarne i tempi di esecuzione. Sebbene libjpeg – e quindi molti programmi che internamente utilizzano questa libreria – supportino file JFIF con Restart Marker, non è garantito che altre librerie che implementano solo codec baseline riescano a decodificare questi file. Per una misura quantitativa del miglioramento sui tempi di esecuzione introdotto da questa ottimizzazione, è necessaria un'analisi specifica che prevedrebbe di estendere la libreria NPP in modo da supportare i Restart Marker, al momento non supportati nativamente.

I valori rilevati in questa analisi risultano di molto superiori ai valori riportati nel documento [4] per le piattaforme Jetson Nano (TX1) e Jetson TX2. Si evidenzia, in particolare un fattore x10 di differenza che potrebbe essere dovuto all'utilizzo di immagini random (a bassa correlazione spaziale). In mancanza di informazioni sulla fonte delle immagini utilizzate per rilevare le latenze di compressione del benchmark di Fastvideo, si ritiene che i valori riportati da Fastvideo siano relativi ad immagini non sintetiche a maggiore correlazione spaziale.

In Tabella 4 riportiamo i tempi di esecuzione (in ms) dell'encoding per immagini naturali del dataset Kodak [12]. Il dataset è costituito da immagini RGB a risoluzione 3072x2048 verticali ed orizzontali. Il formato di encoding di output JPEG è YCbCr 420. Come previsto, i tempi di esecuzione sono inferiori rispetto all'encoding di immagini sintetiche random (fattore di circa 0.7 per NPP e NVJPEG e 0.4 per Fastvideo). Si nota, inoltre, che il margine tra i tempi di esecuzione delle librerie NPP e NVJPEG e Fastvideo incrementi da ~3.4 a ~5.5 – Fastvideo per immagini naturali è circa 5.5 volte più veloce di NPP e NVJPEG – un comportamento spiegabile per la maggior incidenza della codifica di Huffman e, quindi, una maggior rilevanza dell'implementazione ottimizzata di Fastvideo sui tempi di esecuzione.

	GTX 1650	Jetson TX2	Jetson Nano
Kodak RGB (3072x2048)	7.89 / 7.58 / 1.25	128.59 / 126.64 / 25.20	138.30 / 130.1 / 27.10
Random RGB 4K (3840x2160)	13.07 / 11.30 / 2.84	154.23 / 154.53 / 56.54	206.58 / 203.75 / 75.73
Variazione relativa	*0.60 / 0.67 / 0.44*	*0.83 / 0.82 / 0.45*	*0.67 / 0.64 / 0.36*

Tabella 4 Tempi di esecuzione (in ms) delle librerie NPP, NVJPEG e Fastivideo per le piattaforme analizzate relativi al dataset Kodak, comparati con i tempi di esecuzione per immagini random.

9 Conclusioni

In questa attività sono state analizzate e comparate alcune possibili soluzioni esistenti per implementare la codifica JPEG su CPU e GPU. Si è osservato che un'architettura che includa la GPU nel processo di compressione possa ridurne i tempi di esecuzione nonostante vengano introdotte delle latenze dovute alla trasmissione dei dati sul bus PCI. Sono state comparate le prestazioni delle librerie NPP, NVJPEG e Fastvideo, che implementano la compressione JPEG su GPU per le piattaforme Nvidia GeForce GTX 1650 in ambiente Windows 10, Nvidia Jetson Nano e Nvidia Jetson TX2 in ambiente Linux4Tegra. Dall'analisi risulta che Fastvideo è la libreria con prestazioni migliori, ma è una libreria commerciale difficilmente modificabile. Al contrario, la libreria NPP ha il vantaggio di essere estremamente flessibile in quanto presenta un'API che isola le operazioni necessarie per la codifica JPEG e ne permette una libera riorganizzazione. Per immagini sintetiche, i tempi di esecuzione della libreria NPP sono vicini alle specifiche richieste da Alkeria, mentre per immagini naturali Fastvideo risulta essere circa 5.5 volte più veloce di NPP. Si ritiene che tale differenza possa essere spiegata dall'implementazione parallela dell'encoding di Huffman di Fastvideo. Quantificare la variazione delle performance includendo una codifica di Huffman parallela, richiede l'implementazione della stessa e rimandata ad un'analisi successiva. Nel caso in cui tale analisi desse risultati positivi, i tempi di esecuzione di NPP potrebbero essere ulteriormente migliorati adottando un approccio simile a quello utilizzato da Fastvideo per la parallelizzazione della codifica di Huffman impiegando i Restart Marker.

Dai dati raccolti in questa analisi, si identifica la libreria NPP come l'implementazione di riferimento sulla quale verranno adattate le modifiche richieste da Alkeria ed incluse nella Reference Architecture definita nel documento *Reference Architecture and development plan*.

Bibliografia

[1] G. K. Wallace, *The JPEG still picture compression standard*, IEEE Transactions on Consumer Electronics

[2] Eric Hamilton, *JPEG File Interchange Format Version 1.02*

[3] DIS, ISO, *10918-1. Digital Compression and Coding of Continuous-tone Still Images (JPEG)*,

[4] Fastvideo, *Fastvideo Benchmark*

[5] Nvidia, *NVIDIA Turing GPU Architecture*

[6] N. Alqudami and S. Kim, *Accelerating the pre-processing stages of JPEG encoder on a heterogenous system using OpenCL*, 2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)

[7] D. Liu and X. Y. Fan, *Parallel program design for JPEG compression encoding*, 2012 9th International Conference on Fuzzy Systems and Knowledge Discovery

[8] M. K. A. Shatnawi and H. A. Shatnawi, *A performance model of fast 2D-DCT parallel JPEG encoding using CUDA GPU and SMP-architecture*, 2014 IEEE High Performance Extreme Computing Conference (HPEC)

[9] Haweel, Reem T. and El-Kilani, Wail S. and Ramadan, Hassan H., *Fast Approximate DCT with GPU Implementation for Image Compression*, J. Vis. Comun. Image Represent.

[10] Nvidia, *Cuda Best Practices Guide*

[11] K. Konstantoudakis, E. Christakis, P. Drakoulis, A. Doumanoglou, N. Zioulis, D. Zarpalas and P. Daras, *Comparing CNNs and JPEG for Real-Time Multi-view Streaming in Tele-Immersive Scenarios*, 2018 14th International Conference on Signal-Image Technology Internet-Based Systems (SITIS)

[12] https://www.math.purdue.edu/~lucier/PHOTO_CD/BMP_IMAGES/