Alma Mater Studiorum · Università di Bologna

SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea in Ingegneria Informatica

Porting di un algoritmo per la stima del flusso ottico su smartphone Android

Relatore:
Prof. Stefano Mattoccia

Candidato:
Guglielmo Palaferri

Appello II
Anno Accademico 2020-2021

Indice

Capitolo 1

Introduzione

Il monitoraggio costante della velocità di fiumi e correnti d'acqua può assumere notevole importanza sia nello studio di fenomeni idrologici puramente naturali, sia nella progettazione di opere ingegneristiche strettamente legate ad un particolare flusso d'acqua. Ad esempio, può aiutare ad analizzare e rilevare fenomeni come le inondazioni (specie gli avvenimenti improvvisi, che destano particolare attenzione), così come anche il trasporto di sedimenti o l'erosione delle rocce.

Come evidenziato in [1], molte delle tecniche tradizionali utilizzate per l'osservazione di un flusso idrico, tuttavia, non garantiscono grande efficienza e presentano costi elevati: spesso è richiesta la presenza di personale specializzato per la manutenzione di dispositivi complessi.
Una soluzione che preveda invece l'installazione di apparecchi ottici, e basi quindi il monitoraggio sull'elaborazione di immagini, può consentire di abbattere notevolmente i costi e di distribuire il sistema di osservazione ottenendo quindi maggiore resistenza ai guasti.

È proprio questo un caso di utilizzo di **OTV** (*Optical Tracking Velocimetry*), una tecnica che fa uso di particolari algoritmi di computer vision (in particolare l'algoritmo di *Lucas-Kanade*, utilizzato per la stima del flusso ottico) per tracciare le traiettorie e le velocità del flusso d'acqua a partire da una serie di immagini. Il tracciamento viene svolto grazie al riconoscimento di particelle quali detriti e altri residui e al confronto di fotogrammi consecutivi.

Figura 1.1: Esempio di installazione di un dispositivo embedded basato sull'elaborazione ottica

Il metodo OTV è pensato per essere applicato a dispositivi di elaborazione a basso costo e di dimensioni contenute: questi sarebbero posizionati lungo corsi d'acqua in aree geografiche remote. I dati poi raccolti da questi dispositivi potranno essere spediti (tramite meccanismi semplici come l'invio di SMS) ad un sistema di raccolta dati centralizzato. Va da sé dunque che l'ottimizzazione dei consumi energetici dei dispositivi costituisca un punto cruciale per la realizzabilità di un tale sistema di monitoraggio. Questo tema verrà preso in considerazione e rappresenta uno dei punti principali degli studi finora condotti sull'argomento.

L'algoritmo è stato inizialmente testato su dispositivi della famiglia *Raspberry*, per via delle loro dimensioni molto contenute e in generale per le funzionalità da essi offerte, molto coerenti con i requisiti del progetto. Le analisi [2] hanno riportato ottimi risultati dal punto di vista dei consumi energetici in particolare dei modelli Raspberry Pi 3B e 4.

Altri dispositivi con buone potenzialità e con un profilo che si presti bene al contesto di utilizzo sono gli **smartphone**, con particolare riferimento a quelli basati su sistema operativo **Android**. L'utilizzo di tali dispositivi richiede ovviamente una seppur minima quantità di modifiche rispetto al deployment effettuato su Raspberry, ed è

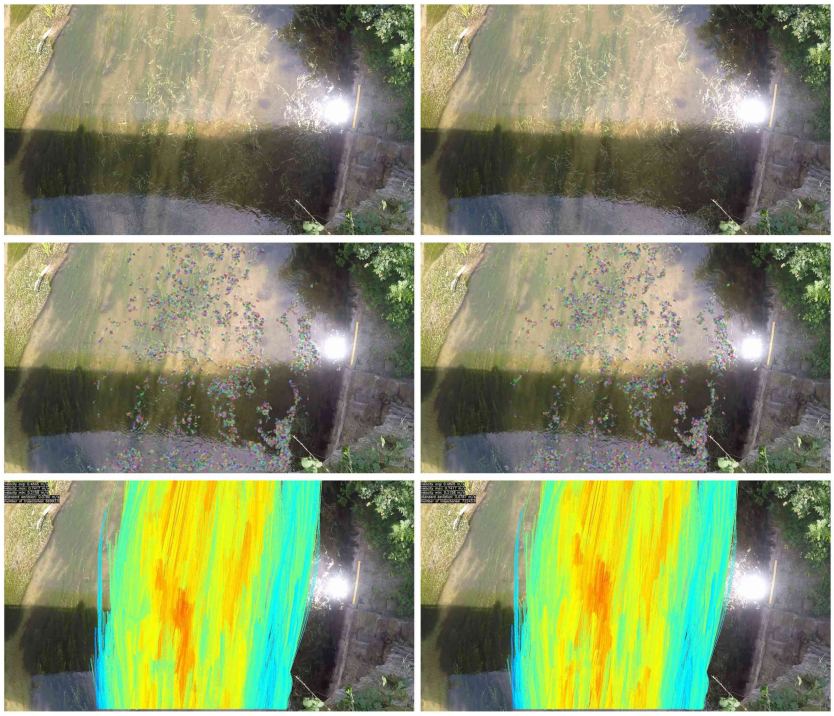

Figura 1.2: Sequenze video ottenute con OTV: le linee tracciate rappresentano le traiettorie riconosciute, colorazioni più calde indicano velocità maggiori

proprio questo il tema centrale del seguente documento.

Nei prossimi capitoli si procede quindi — dopo aver introdotto qualche informazione necessaria su OTV — a descrivere la realizzazione di un'applicazione per smartphone Android che adatti l'implementazione in C++ di OTV (disponibile su GitHub al link [3]) e i risultati in termini di prestazioni e consumi energetici che ne sono conseguiti.

Capitolo 2

OTV

2.1 Contesto di utilizzo

Come già brevemente descritto, OTV prevede un deployment su dispositivi di dimensioni ridotte e autosufficienti dal punto di vista energetico. In particolare, la configurazione testata su Raspberry [2] introduceva i seguenti componenti:

- Raspberry Pi 3B/4 per l'elaborazione
- Panello solare 6 W (PiJuice Solar Panel) per sostenere i consumi energetici
- Batteria esterna (PiJuice Hat) per fornire alimentazione

Una simile configurazione verrebbe usata con smartphone, salvo ovviamente l'utilizzo di una batteria aggiuntiva.

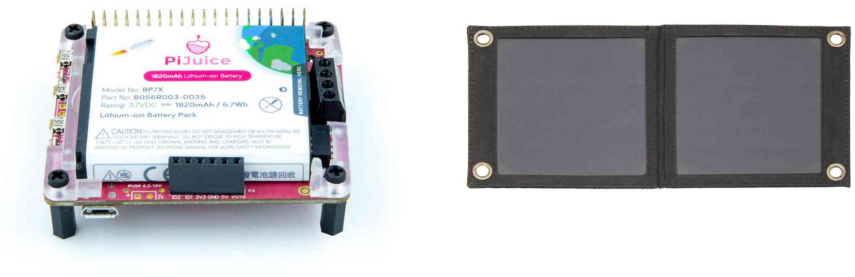

Figura 2.1: Esempio di deployment del dispositivo su Raspberry Pi

2.2 Ciclo di funzionamento

Il dispositivo Android così composto, una volta accuratamente posizionato ed avviato, dovrebbe eseguire *quattro* misurazioni della velocità dell'acqua ogni ora, risultando quindi a regime in un ciclo di funzionamento periodico della durata di 15 minuti.

Sebbene la misurazione mediante l'algoritmo OTV sia svolta sul momento, non viene effettuata sulle immagini direttamente ricevute e lette in input dalla telecamera: il video acquisito necessita di una fase preliminare che prepari le immagini per essere elaborate. Questo viene fatto, tra le altre cose, per consentire di scegliere un settaggio particolare (ad esempio, selezionare una risoluzione diversa rispetto al video originale), utile successivamente al fine di ottimizzare l'elaborazione.

Il ciclo di funzionamento si articola quindi in questo modo:

1. Fase di **acquisizione**: le immagini vengono acquisite dalla telecamera. Questa fase ha una durata fissa e dipende dalla lunghezza del video che si vuole analizzare: tipicamente 20 secondi.
2. Fase di **estrazione** dei frame: a partire dal video acquisito, si estraggono i fotogrammi che lo compongono a seconda della configurazione scelta, in particolare è possibile specificare la risoluzione desiderata tra:
   - Full Resolution (**F**): Risoluzione originale
   - Half Resolution (**H**): Risoluzione dimezzata
   - Quarter Resolution (**Q**): Risoluzione 1/4 dell'originale
3. Fase di **elaborazione** (OTV): a questo punto le immagini estratte vengono effettivamente elaborate utilizzando OTV. Questa fase è cruciale dal punto di vista dei consumi in quanto è quella che può variare maggiormente a seconda della configurazione usata e delle ottimizzazioni implementate. È bene quindi analizzarla di conseguenza.
4. Fase di **idle**: una volta conclusa l'elaborazione (ed eventualmente spediti i dati rilevati) segue un periodo di stand-by, in cui si attende il tempo necessario prima della prossima rilevazione. Anche questa fase è molto importante per

determinare i consumi energetici del processo: se il dispositivo dovesse disporre di una modalità di risparmio energetico, l'energia utilizzata potrebbe diminuire drasticamente.

Le fasi su cui è possibile effettivamente lavorare per ottenere risultati migliori sono quelle di elaborazione (in modo particolare) e di idle.

Prima di introdurre i vari livelli di ottimizzazione, va intanto fatto notare che la specifica implementazione di OTV presa in caso è basata sulla libreria open-source di computer vision **OpenCV**.
OpenCV fornisce un framework per la creazione di applicazioni legate alla computer vision e implementa una vasta gamma di algoritmi, tra cui l'algoritmo di Lucas-Kanade utilizzato da OTV già menzionato.
L'utilizzo di OpenCV prescrive una serie di passaggi di installazione che variano in base all'ambiente di sviluppo e che — nel caso specifico di Android — verranno analizzati nel successivo capitolo.

2.3 Ottimizzazioni

Le ottimizzazioni applicabili ad OTV analizzate in [4] consistono in una serie di tecniche e meccanismi che possano contribuire ad aumentare l'efficienza energetica del sistema. Tra queste, possiamo distinguere quelle legate al **software** e quelle invece a livello **hardware** (ad esempio, l'utilizzo di istruzioni particolari).

Le ottimizzazioni software consistono essenzialmente nella configurazione del dispositivo in modo ad esempio da disattivare le opzioni software che risultino superflue (Wi-Fi, Bluetooth ecc.). Si parla di ottimizzazioni derivanti dal sistema operativo utilizzato e dunque dipendenti dal dispositivo in questione. Si vedranno ottimizzazioni di questo tipo esclusive al sistema Android.

Per quanto riguarda le ottimizzazioni software, una di queste è risultata particolarmente efficiente nei test effettuati su Raspberry mostrati in [2]: l'elaborazione in **scala di grigi** (o **monocromatica**). Non si tratta di una configurazione a livello di sistema operativo bensì di una piccola modifica al codice di OTV: i fotogrammi precedentemente estratti vengono acquisiti — mediante le API di OpenCV — come immagini monocromatiche invece che a colori. Questo passaggio non comporta

risultati particolarmente diversi (in termini di velocità e numero di traiettorie rilevate) ma consente di ottenere un notevole guadagno in termini di prestazioni. Questo deriva dal fatto che le immagini in scala di grigi sono composte da un singolo canale, a differenza delle immagini a colori rappresentate invece da tre canali (Rosso, Verde, Blu).

2.3.1 Ottimizzazioni hardware

Nel caso delle ottimizzazioni hardware, si parla di particolari metodi che introducono differenti modelli di esecuzione a livello di processore, facendo leva specialmente sulla parallelizzazione delle istruzioni. Questo, oltre agli ovvi vantaggi in termini di performance, può portare ad una maggiore efficienza in termini di consumi. Si delineano tre possibilità principali, eventualmente sovrapponibili, focalizzate su aspetti e modalità diverse di parallelizzazione:

- Esecuzione multi-core mediante **OpenMP** o **TBB**.
- Esecuzione di istruzioni **SIMD** tramite **NEON**
- Esecuzione su **GPU** mediante la libreria **OpenCL**

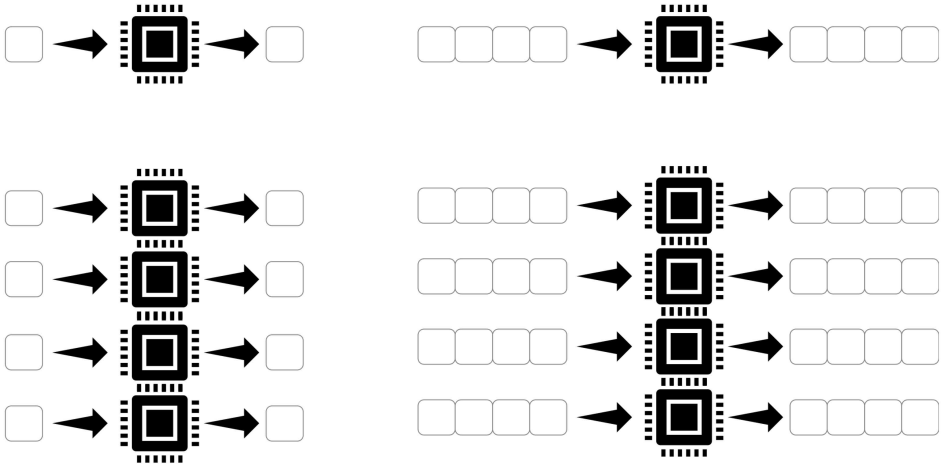

Figura 2.2: Confronto tra le varie ottimizzazioni hardware: Baseline (in alto a sinistra), Multicore (in basso a sinistra), SIMD (in alto a destra), SIMD Multicore (in basso a destra)

OpenMP e TBB

OpenMP e TBB sono due librerie che in fase di sperimentazione sono state utilizzate per testare il parallelismo *thread-level* in modo da sfruttare i multipli core disponibili nelle moderne CPU. Le due librerie — poiché forniscono lo stesso tipo di funzionalità — sono da utilizzare in modo mutuamente esclusivo. La scelta della libreria da utilizzare cadrà dunque su quella che garantisca le migliori prestazioni.

SIMD

SIMD (*Single Instruction Multiple Data*) è una classe di istruzioni che consente di ottenere parallelismo su un singolo core. Il modello prevede l'esecuzione della stessa operazione su una molteplicità di dati mediante l'utilizzo di una singola istruzione. All'interno di un'istruzione vengono quindi inglobati diversi dati e, ovviamente, l'operazione da svolgersi.

I processori ARM, montati sulla stragrande maggioranza di dispositivi Android così come anche sui modelli di Raspberry Pi già testati, dispongono di un'architettura SIMD avanzata chiamata NEON.
Poiché la libreria OpenCV fornisce nativamente supporto per le istruzioni NEON, risulta piuttosto immediato testare l'utilizzo di tali istruzioni nell'applicazione OTV.

L'utilizzo delle istruzioni SIMD può beneficiare specialmente i casi in cui una singola operazione debba essere ripetuta molte volte su un insieme di dati anche grande. È il caso dell'elaborazione di immagini, in cui spesso è richiesto operare su dati sotto forma di matrici ed eseguire operazioni anche semplici ma molto ripetitive.

GPU e OpenCL

L'utilizzo della potenza di calcolo di una GPU è giustamente considerato tra le ottimizzazioni da testare: qui il parallelismo è intrinseco al tipo di processore, che è progettato appositamente per svolgere operazioni semplici ma ripetitive su un grande insieme di dati, in particolare nei casi di elaborazione di immagini. Tuttavia, come evidenziato in [4] il potenziale guadagno in efficienza che questa soluzione garantirebbe non è facilmente stimabile e dipende da una varietà di fattori.

La libreria OpenCL permette di sfruttare le funzionalità della GPU, qualora supportata dal dispositivo in questione.

Capitolo 3

Processo di sviluppo

Parlando ora dell'effettiva operazione di porting di OTV su dispositivo Android, si espongono gli strumenti utilizzati e le complessità riscontrate durante il processo di sviluppo. Verranno trattati inoltre i dettagli tecnici dell'applicazione e i meccanismi utilizzati propri del sistema Android.

3.1 OpenCV

Come brevemente spiegato nel capitolo precedente, il codice disponibile di OTV è basato su OpenCV. Nel caso di Android, i moduli di OpenCV devono essere necessariamente incapsulati all'interno dell'applicazione, non essendo possibile installarli tra le librerie di sistema. Chiaramente, i moduli dovranno essere compilati con la toolchain di Android per poter ottenere binari effettivamente compatibili con il sistema operativo.

I binari di OpenCV sono reperibili sul sito ufficiale oppure compilabili manualmente tramite una **cross-compilazione**: questa seconda opzione è preferibile in quanto consente di configurare a proprio piacimento l'installazione di OpenCV, aggiungendo o rimuovendo selettivamente le ottimizzazioni introdotte in sezione 2.3.

Tutte le ottimizzazioni hardware discusse, infatti, sono disponibili su OpenCV. Non avrebbe senso invece implementarle nel codice OTV poiché tutte le operazioni che facciano uso intensivo della CPU e che richiedano tempi di esecuzione piuttosto

lunghi sono interne alla libreria OpenCV. L'unica ottimizzazione che effettivamente viene implementata nel codice di OTV è l'elaborazione delle immagini in scala di grigi (non essendo a livello hardware): è sufficiente aggiungere una riga di codice nel momento in cui i frame estratti vengono letti dall'applicazione.

3.1.1 Cross-compilazione

Dovendo testare numerose configurazioni diverse, per la cross-compilazione di OpenCV è stato preparato un semplice script in bash che automatizzasse in minima parte il processo e fornisse inoltre uno scheletro da seguire nel caso in cui si volesse modificare la configurazione di OpenCV, in particolare aggiungendo o rimuovendo ottimizzazioni hardware. Si riporta di seguito il passaggio principale dello script:

```
for ABI in "armeabi-v7a" "arm64-v8a"; do
    mkdir -p build_$ABI
    cd build_$ABI
    rm CMakeCache.txt
    cmake \
    -DCMAKE_TOOLCHAIN_FILE=$ANDROID_TOOLCHAIN_PATH \
    -DANDROID_ABI=$ABI \
    -DANDROID_NATIVE_API_LEVEL=$ANDROID_API_LEVEL \
    -DANDROID_STL=c++_shared \
    -GNinja -DCMAKE_BUILD_TYPE:STRING=Release \
    -DBUILD_FAT_JAVA_LIB=OFF \
    -DBUILD_JAVA=OFF \
    -DBUILD_SHARED_LIBS=ON \
    -DWITH_TBB=ON -DBUILD_TBB=ON \
    -DENABLE_NEON=OFF \
    -DWITH_OPENMP=OFF \
    -DWITH_OPENCL=OFF \
    -DWITH_CAROTENE=OFF \
    -DWITH_PTHREADS_PF=OFF \
    -DANDROID_SDK_ROOT=$ANDROID_SDK_ROOT \
    -DANDROID_NDK=$ANDROID_NDK \
    -DOPENCV_ENABLE_NONFREE=ON \
    -DBUILD_TESTS=FALSE -DBUILD_PERF_TESTS=FALSE \
    -DBUILD_ANDROID_EXAMPLES=FALSE \
    -DBUILD_DOCS:BOOL=OFF -DWITH_IPP=OFF -DWITH_MSMF=OFF \
```

```
        ..

        ninja
        ninja install
        cd ..
    done
```

In questo caso viene utilizzato **CMake** per configurare la compilazione (scelta obbligata, poiché il file CMake è fornito nella distribuzione di OpenCV) e **ninja** come *build system*. CMake infatti non esegue la vera e propria compilazione, ma prepara l'ambiente e crea i file di compilazione che verranno poi usati dal build system per effettivamente compilare il progetto. In alternativa a ninja era possibile utilizzare **Make**, la scelta fra i due non comporta nessuna differenza nel risultato ma può influire leggermente sui tempi di compilazione.

Tra le diverse opzioni di CMake si nota intanto `ANDROID_ABI`, che specifica l'ABI per il quale compilare OpenCV. Un ABI (*Application Binary Interface*) rappresenta, tra le altre cose, l'insieme di set di istruzioni supportati da un dispositivo, come spiegato in [5]. Si limita la scelta ad `armeabi-v7a` e `arm64-v8a`, rispettivamente le ABI delle architetture ARM a 32 e 64 bit. Architetture differenti da ARM non verrebbero comunque testate su Android.

Altre opzioni interessanti sono quelle riguardanti le ottimizzazioni hardware: ad esempio `WITH_TBB`, `ENABLE_NEON` o `WITH_OPENCL`. Per abilitare le ottimizzazioni è sufficiente settare a `ON` l'opzione di interesse.

3.1.2 Installazione

Una volta ottenuti i binari di tutti i moduli necessari di OpenCV (tramite cross-compilazione o download), è possibile procedere con l'inserimento all'interno del progetto di Android Studio: come già detto, queste librerie dovranno essere parte integrante dell'applicazione.

L'inserimento è molto meccanico e avviene con la creazione di due cartelle nel progetto: una per contenere gli header di OpenCV, l'altra per contenere le librerie dinamiche contenenti il codice (file `.so`). Sia gli header che le librerie vengono in-

clusi nel progetto aggiungendo istruzioni ad hoc all'interno del file `CMakeLists.txt` generato automaticamente da Android Studio per compilare la componente nativa del progetto che comunichi con gli entry point Java. Segue un esempio con le principali istruzioni CMake utilizzate per aggiungere le librerie di OpenCV:

```
include_directories(${CMAKE_SOURCE_DIR}/../jniIncludes)

add_library( libopencv_core SHARED IMPORTED )
set_target_properties( libopencv_core PROPERTIES IMPORTED_LOCATION
        ${CMAKE_SOURCE_DIR}/../cmakeLibs/${ANDROID_ABI}/libopencv_core.so)

#...

target_link_libraries(
        # Specifies the target library.
            native-lib

        # Links the target library to the log library
        # included in the NDK.
        ${log-lib}

        libopencv_core
        #...
)

#...
```

3.2 Sviluppo dell'applicazione Android

Il deployment e l'installazione di applicazioni su dispositivi Android avviene mediante l'uso di file .APK. I file APK sono sostanzialmente degli archivi compressi in formato .ZIP il cui contenuto è quindi visualizzabile con un qualsiasi gestore di archivi. Contengono tutti i file necessari per una singola applicazione Android: librerie native, classi Java compilate, ma anche risorse e immagini.

La creazione dei file APK a partire dai file sorgente e dal progetto iniziale spetta solitamente a sistemi di compilazione. Per facilitare questa operazione, viene di

norma utilizzato un ambiente di sviluppo e quindi un IDE specializzato.
Nel caso di Android l'IDE consigliato è Android Studio, che fornisce un bundle di installazione contenente tutti gli strumenti e le toolchain necessarie, facilmente configurabili.

Il linguaggio principale supportato da Android è **Java** (ultimamente anche Kotlin sta avendo largo impiego): ogni applicazione richiede anche solo una minima parte scritta in Java o Kotlin per poter utilizzare le API di sistema e gestire i cosiddetti *Entry Point* dell'applicazione, ovvero i meccanismi tramite i quali l'utente può avviare ed interagire con l'applicazione, l'interfaccia utente è infatti manipolabile esclusivamente con l'utilizzo di classi Java/Kotlin.

È comunque disponibile il supporto per codice nativo C/C++ grazie all'NDK (*Native Development Kit*): un insieme di strumenti per la compilazione e il debug integrabili con Android Studio. Lo strumento di configurazione di default per codice nativo è CMake, tra i più diffusi per progetti C e C++: un esempio di configurazione tramite CMake è quello giusto esposto nella sezione di cui sopra, in cui si aggiungono le librerie OpenCV al progetto.

Per configurare infine il progetto nella sua interezza, compresi i dettagli di deployment, Android Studio richiede l'utilizzo del plugin **Gradle**, che può essere integrato con CMake (in modo molto automatizzato) qualora sia prevista la presenza di codice nativo.

3.2.1 Architettura dell'applicazione

Il codice Java e quello nativo, all'interno di un'applicazione Android, comunicano grazie al framework **JNI** (*Java Native Interface*). Questo framework consente in modo particolare l'invocazione di funzioni scritte in codice nativo da parte di classi Java.

Nonostante il funzionamento di JNI sia piuttosto complesso (si veda la documentazione in [6]), l'utilizzo di esso in un progetto Android Studio avviene in modo piuttosto guidato e trasparente nei confronti dello sviluppatore: tutti i dettagli interni di collegamento e configurazione tra i vari componenti vengono gestiti in maniera automatizzata. Inoltre, come si vedrà, ai fini di questo progetto è stato sufficiente fare uso delle funzioni più basilari di JNI.

Dopo un'attenta analisi, si è deciso di strutturare l'applicazione come segue:

- Entry point Java (`MainActivity.java`);
- Classe Java che implementi un *thread* secondario per l'invocazione del codice OTV (`OTVThread.java`);
- File C++ che faccia da wrapper per le funzionalità di OTV, interfacciandosi con il codice Java (file `native-lib.cpp`);
- Codice di OTV, pressoché inalterato (cartella `otv/src/`).

Il motivo per cui si è scelto di introdurre un thread apposito per l'invocazione di OTV è legato alle modalità con cui la classe MainActivity gestisce l'interazione con l'utente. Ogni *Activity* ha un ciclo di vita — descritto in [7] — basato su funzioni di callback invocate dal sistema Android a seguito di determinate azioni compiute dall'utente, ad esempio il metodo `onCreate`, invocato non appena l'utente apre l'applicazione.

Poiché OTV presenta le caratteristiche di un demone (ha un ciclo di esecuzione che idealmente non termina mai), inserirlo all'interno di un metodo di callback significherebbe deviare dal normale ciclo di esecuzione dell'attività. La scelta di un thread che esegua OTV in background è quasi obbligata in quanto è l'unico modo di garantire un'esecuzione (quasi) disaccoppiata dagli eventi della UI ma soprattutto che avvenga in modo asincrono rispetto alla creazione degli elementi dell'interfaccia grafica: se si invocasse OTV all'interno del metodo `onCreate`, l'applicazione risulterebbe bloccata sulla creazione dell'interfaccia.

Nel metodo `onCreate` della classe `MainActivity` risulterà quindi:

```
protected void onCreate(Bundle savedInstanceState) {
    //...
    OTVThread otvThread = new OTVThread(updateUIHandler);
    otvThread.start();

    //casella di testo nella UI
    tv.setText("OTV was launched in background...");
}
```

Nella classe `OTVThread`:

```
public void run() {
    String otv_results = runOTV();
    //...
}

// metodo nativo invocabile tramite JNI
public native String runOTV();
```

Infine, nel wrapper C++ (`native-lib.cpp`):

```
extern "C" JNIEXPORT jstring JNICALL
Java_gullp_androidotv_OTVThread_runOTV( JNIEnv* env, jobject) {

    //costruzione dei parametri richiesti
    char* argv[6];
    argv[0] = "OTV_AndroidInstance";
    argv[1] = PATH_INFO_VIDEO;
    argv[2] = PATH_FILE_PARAMS;
    argv[3] = PATH_LK_FILE;
    argv[4] = PATH_OUTPUT;
    argv[5] = PATH_MASK;

    LOGD("Running OTV...");

    //effettiva invocazione del codice OTV nativo
    return env->NewStringUTF(run(6, argv).c_str());
}
```

I file di configurazione richiesti da OTV (contenenti i parametri di calibrazione per l'algoritmo di Lucas Kanade) sono stati inseriti nella cartella `/data/data/{nome_package}` del dispositivo, che rappresenta la porzione di storage interno dedicata all'applicazione.

Capitolo 4

Test e risultati

Completato lo sviluppo dell'applicazione, ha avuto luogo la fase finale nonché probabilmente la più importante: il testing di varie configurazioni e la conseguente raccolta ed analisi dei risultati, al fine di valutare i vantaggi e gli svantaggi di un porting sul sistema Android.

A differenza del testing su Raspberry Pi, i cui modelli sono molto standardizzati e distribuiti dallo stesso produttore, per gli smartphone Android i risultati ottenuti in termini di performance e consumi possono variare anche molto al variare del dispositivo, a seconda delle caratteristiche tecniche di quest'ultimo.

Per quanto riguarda il seguente progetto, i test sono stati svolti su un dispositivo Android basato sul SoC *Qualcomm Snapdragon 730* (sul quale è presente una CPU octa-core).

4.1 Testing

Ai fini del testing sono stati utilizzati fotogrammi precedentemente estratti: riprodurre l'intero ciclo di esecuzione OTV su Android, comprese le fasi di acquisizione ed estrazione introdurrebbe diverse complessità, costringendo a fare i conti con diversi problemi (ad esempio, la mancanza di *ffmpeg* su Android). Difatti, nel testare l'applicazione si è considerata esclusivamente la fase di esecuzione, tralasciando le

altre tre fasi presentate in sezione 2.2 principalmente per motivi di tempo. L'esecuzione di OTV rappresenta comunque la fase più importante e con il maggior numero di variabili che possano influenzare le prestazioni e i consumi: le analisi che ne conseguono sono sicuramente indicative delle potenzialità di un dispositivo.

Il testing è stato eseguito mediante l'ausilio degli strumenti forniti dalla repository GitHub di OTV ([3]), in particolare gli script Bash adibiti alla creazione dei file di configurazione richiesti per il funzionamento del programma. Lo script che implementa l'estrazione dei frame a partire da un file video fa uso del programma *ffmpeg* e salva i fotogrammi estratti all'interno di una cartella `frames` il cui percorso va specificato.

La cartella dei frames e gli altri file necessari sono stati quindi prima generati sulla macchina utilizzata per lo sviluppo dell'applicazione, poi trasferiti sul dispositivo Android mediante l'interfaccia grafica fornita da Android Studio. Collegando infatti lo smartphone al PC su cui esegue Android Studio è possibile accedere ad una sezione dedicata al file system del dispositivo.

Come già menzionato, tutti i file di servizio sono stati inseriti all'interno della cartella `/data/data/{package_applicazione}` poiché questa fa parte del cosiddetto storage *interno* dell'applicazione, in cui gli accessi non sono vincolati da permessi richiesti all'utente. Questa cartella inoltre non è accessibile dall'utente se non tramite strumenti esterni al dispositivo (come il debugger Android) e può quindi essere utile ad evitare cancellazioni e/o modifiche accidentali dei file necessari.

Particolare attenzione va posta al file `info_video.txt`, generato in seguito all'estrazione dei frame. Il file si presenta nel seguente modo:

```
FPS 25
Width 715
Height 540
FileCount 500
DirectoryFrame /data/data/gullp.androidotv/frames
Jump 2
```

dove il parametro DirectoryFrame rappresenta la directory in cui sono presenti i frame: il file viene letto dall'applicazione in fase di inizializzazione, quindi è im-

portante che il percorso di tale directory sia accurato. Tuttavia, essendo il file `info_video.txt` generato da uno script eseguito su una macchina differente, è necessario andare a modificare tale parametro specificando la corrispettiva cartella nel dispositivo Android.

Il processo di testing è avvenuto seguendo un'iterazione di questo tipo:

1. Copia delle librerie dinamiche precompilate di OpenCV all'interno del progetto, ottenute con la cross-compilazione e nelle quali è attivo uno specifico set di ottimizzazioni;
2. Esecuzione dell'applicazione e trascrizione dei risultati ottenuti
3. Ripetizione del punto 2 fino ad ottenere una quantità soddisfacente di risultati
4. Reiterazione a partire dal punto 1 utilizzando una nuova configurazione di OpenCV

I risultati così ottenuti sono poi stati sottoposti ad un'analisi finale per stabilire i vantaggi di ciascuna configurazione.

4.2 Stima dei consumi energetici

Data l'influenza che i consumi energetici hanno sulla realizzabilità e sulla valutazione di un dispositivo embedded di questo tipo, si è deciso di svolgere — anche se non approfonditamente — una stima dell'energia consumata da esso. È importante notare che si tratta di una stima, poiché gli strumenti utilizzati consentono un livello di precisione piuttosto basso nella misurazione dei consumi su periodi piuttosto brevi di tempo.

Per stimare i consumi energetici dell'esecuzione di OTV è stato utilizzato uno strumento messo a disposizione per gli sviluppatori Android: ***Battery Historian***. Si tratta di un tool web il cui utilizzo molto semplice è descritto in [8]. In sostanza, analizza i file di log generati dal dispositivo e ne visualizza gli eventi correlati al consumo di batteria, fornendone una presentazione ad alto livello tramite grafici e tabelle. Tra i parametri analizzati da Battery Historian figurano il livello di carica

della batteria (in mAh) e la tensione generata da essa (in mV). L'analisi viene svolta su tutto l'arco temporale coperto dai log: è bene quindi resettare i log relativi all'uso della batteria.

I file di log necessari all'utilizzo di Battery Historian possono essere acquisiti mediante il debugger Android (`adb`): la procedura guidata è disponibile in [9].

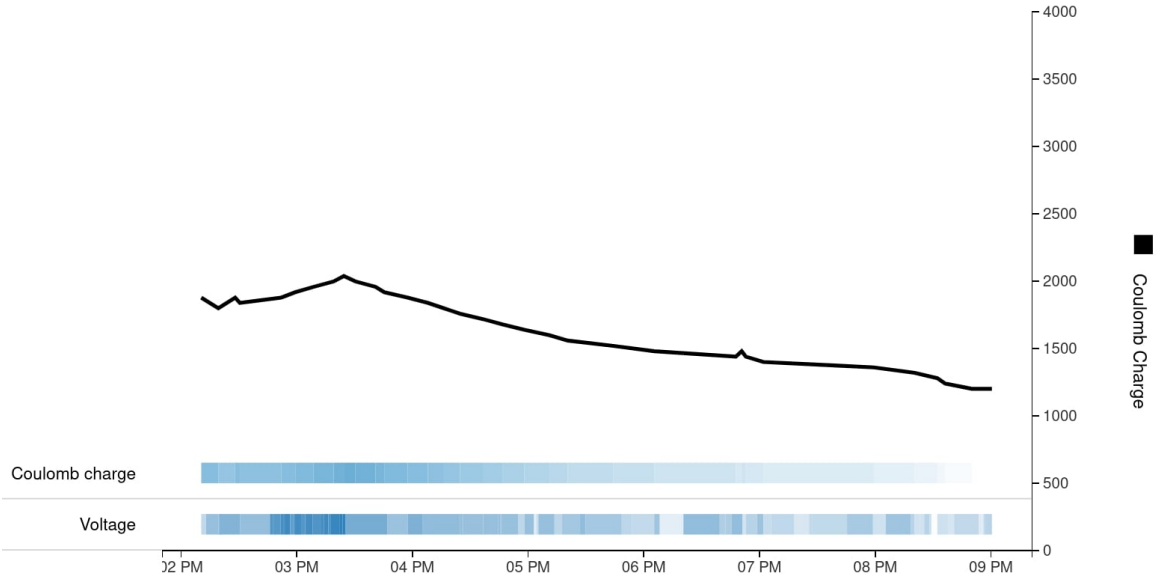

Figura 4.1: Andamento nel tempo del livello di carica della batteria in mAh

L'immagine di cui sopra rappresenta uno dei grafici ottenibili con Battery Historian di maggiore interesse al fine di determinare il consumo energetico: l'andamento del livello di carica. Passando il cursore lungo il grafico è possibile ottenere informazioni aggiuntive particolarmente utili.

Come si può in parte notare dalla figura, l'andamento è discretizzato con dei campionamenti del livello di carica ad intervalli di tempo non regolari.

Il dettaglio più interessante tra quelli forniti è il *Discharge rate* espresso in mA: è il rapporto tra la differenza di carica ΔQ e la lunghezza dell'intervallo Δt e rappresenta quindi la media della **corrente in uscita** dalla batteria sull'intervallo di tempo:

$$I_i = \frac{\Delta Q_i}{\Delta t_i}$$

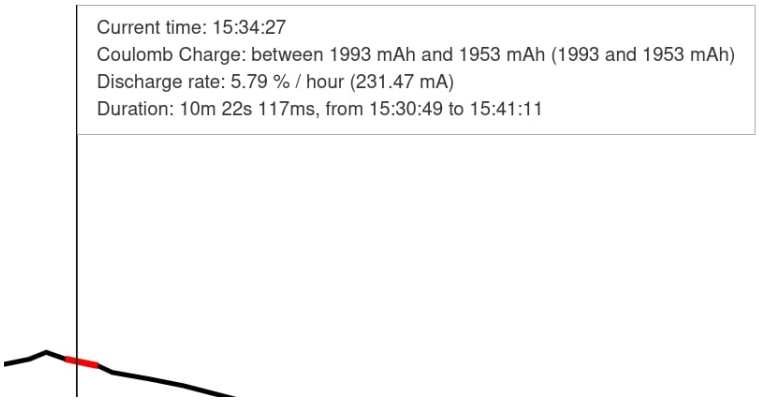

Figura 4.2: Informazioni dettagliate relative ad uno specifico intervallo di tempo

È possibile comunque applicare la medesima espressione conoscendo la carica in mAh sapendo che 1 $mAh = 3.6$ C.

A questo punto, per calcolare la potenza dissipata dallo smartphone durante l'esecuzione di OTV e dunque ottenere l'energia consumata, è necessario avere a disposizione altre due informazioni:

- I tempi di inizio e fine dell'esecuzione di OTV
- Il valore della tensione durante l'esecuzione

Entrambe le informazioni sono reperibili immediatamente grazie alle tabelle fornite da Battery Historian. Analizzando i grafici della tensione si nota che essa non è costante ma soggetta a variazioni molto frequenti, tende tuttavia ad assumere valori piuttosto omogenei con una bassa varianza ed è quindi approssimabile ad un valore medio con una discreta precisione, specie se l'intervallo di tempo analizzato è relativamente piccolo (60 secondi con la configurazione peggiore di OTV).
Supponendo quindi di approssimare la tensione ad un valore V e di avere a disposizione N intervalli di calcolo della corrente durante l'esecuzione di OTV, è possibile calcolare l'energia consumata basandosi sulla potenza media dissipata come segue:

$$E = P \cdot \Delta t = \sum_{i=1}^{N} VI_i \cdot \Delta t_i$$

Dove Δt è la durata dell'esecuzione di OTV e Δt_i è la lunghezza dell'intervallo di tempo in cui si verifica la corrente i-esima I_i.
Poiché, come fatto notare poco sopra, l'esecuzione di OTV termina in un lasso di tempo piuttosto breve, sono spesso osservabili tensioni e correnti costanti lungo tale periodo.

4.3 Analisi dei risultati

Al termine del progetto ha avuto luogo la fase di analisi dei risultati, sia dal punto di vista delle prestazioni che dei consumi. Tutti i test sono stati eseguiti sullo stesso campione video della durata di 20 secondi. La maggior parte dei test è stata eseguita con fotogrammi ad *Half Resolution* in quanto, alla luce dei risultati ottenuti in [2], questa configurazione si è dimostrata il migliore compromesso tra affidabilità dei risultati e velocità di esecuzione. Per lo stesso motivo, nessun test è stato svolto con una configurazione a *Quarter Resolution*.

L'ottimizzazione relativa all'utilizzo della GPU è stata esclusa dai test poiché, dopo alcuni tentativi, è stato impossibile abilitarla correttamente: analizzando i log del sistema Android, si riscontrano errori di permessi e di letture relativi alla GPU del dispositivo, probabilmente dovuti ad un mancato supporto di OpenCL.

Si riportano di seguito i dati ottenuti in termini di tempo di esecuzione, evidenziando le differenze tra le varie configurazioni. I tempi riportati corrispondono al valore medio del tempo di esecuzione osservato tra tutti i test svolti, la varianza è risultata tale da essere trascurabile.

Come si può notare in Figura 4.3, l'elaborazione in scala di grigi influisce notevolmente sulle prestazioni, riducendo almeno del 50% i tempi in tutti i test svolti. Sorprendentemente, l'abilitazione di istruzioni SIMD NEON non ha influito in alcun modo sull'esecuzione, risultando anzi in un lieve peggioramento delle prestazioni (pochi decimi di secondo). Il motivo di ciò non è ben chiaro e potrebbe essere riconducibile a diverse cause, ad esempio un banale errore di configurazione.

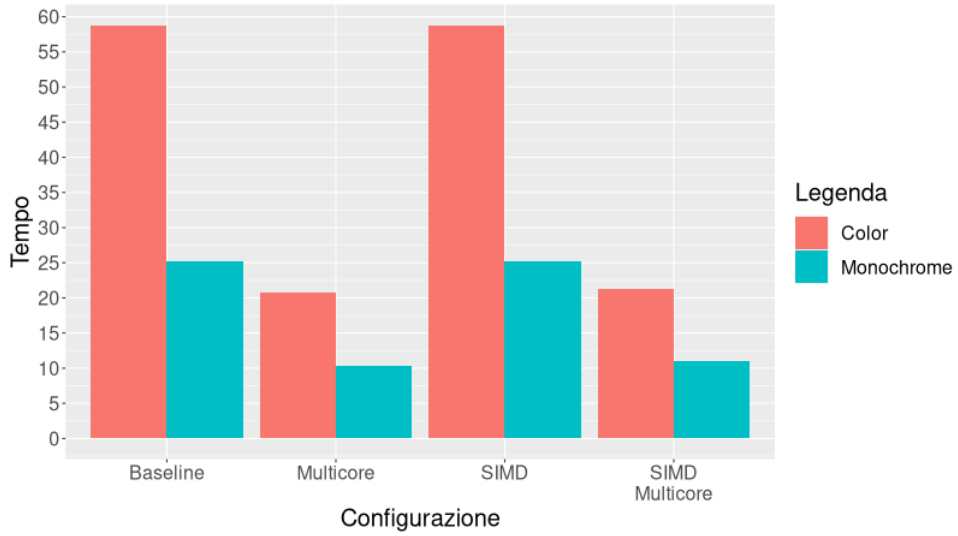

Figura 4.3: Tempo di esecuzione [s] (**Half Resolution**)

Nel testare invece i consumi energetici sono state abilitate tutte le possibili ottimizzazioni software fornite dal sistema Android: disabilitazione di connessione dati, Wi-Fi, Bluetooth, attivazione della **modalità risparmio energetico** e luminosità dello schermo impostata al minimo, avendo cura di mantenere attive le funzionalità basilari necessarie alla comunicazione (es. SMS). Nessuna di queste ottimizzazioni ha influito negativamente sulle prestazioni dell'applicazione.

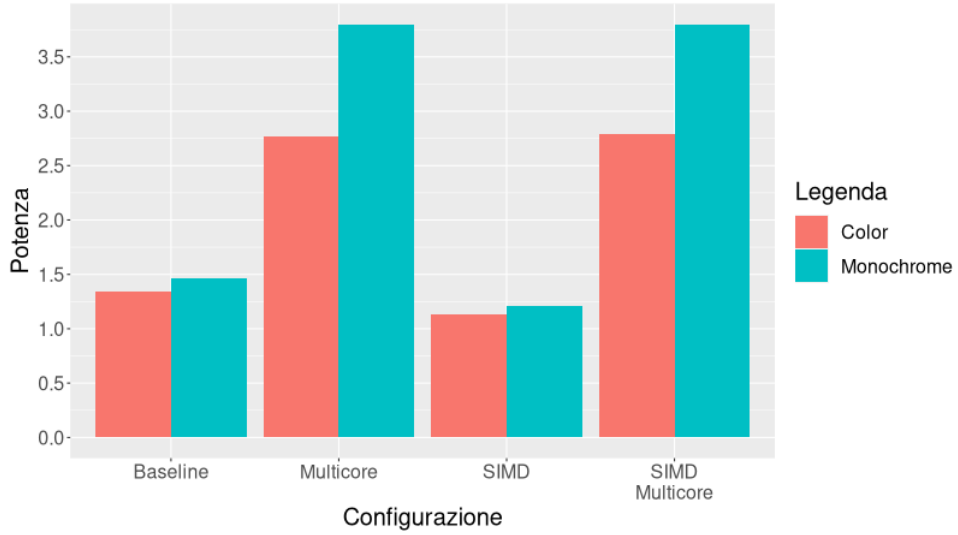

Figura 4.4: Potenza dissipata [W] (**Half Resolution**)

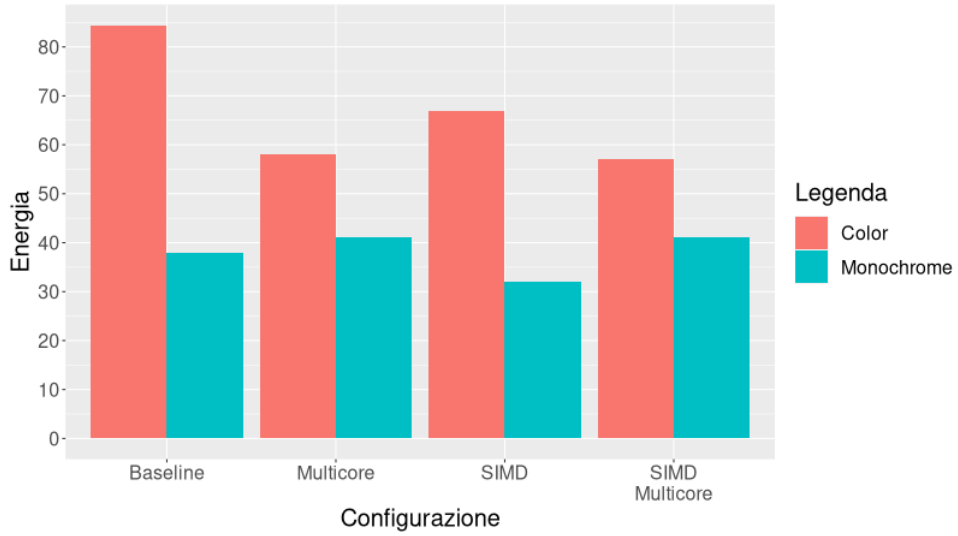

Figura 4.5: Energia consumata [J] (**Half Resolution**)

Contrariamente a quanto riscontrato con i test su Raspberry Pi, la configurazione che pare garantire il minor consumo di energia sembra essere quella che fa uso del minor numero di ottimizzazioni hardware. Si ricorda che i grafici di potenza ed energia qui riportati sono una stima dei consumi energetici ottenuti con il metodo descritto in sezione 4.2, il consumo reale potrebbe discostarsi da quello stimato. Risultano in effetti insoliti i picchi di potenza in Figura 4.4 registrati con configurazioni *Multicore* in elaborazioni monocromatiche.

Confrontando queste stime con i consumi dei dispositivi Raspberry Pi riportati in [2], si nota un netto miglioramento dei consumi energetici stimati. Questa analisi va comunque corredata da considerazioni più complete circa il ciclo di esecuzione dell'applicazione Android finale.

Conclusioni

Il porting di OTV su sistema Android ha prodotto risultati decisamente promettenti, evidenziando performance e consumi energetici anche parzialmente migliori rispetto a quanto ottenuto su Raspberry Pi.

Per fornire una valutazione completa e confrontare sotto ogni aspetto le due soluzioni, è necessario condurre un'analisi più approfondita sui consumi energetici, specie considerando il ciclo di esecuzione completo di OTV che prevede le fasi aggiuntive di acquisizione, estrazione ed infine di idle.
In particolare, la fase di idle risulta particolarmente critica nei sistemi Android, poiché essi — al contrario dei Raspberry — non dispongono di una modalità di stand-by ma devono fare affidamento alle varie modalità di sistema meno sospensive quali il blocco dello schermo.

Qualora si decidesse di approfondire il presente lavoro, il primo passo sarebbe quindi quello di completare l'applicazione inserendo componenti dedicate all'estrazione, acquisizione, e soprattutto un'implementazione ad hoc della fase di idle.

L'utilizzo di dispositivi Android nell'ambito di applicazione discusso potrebbe — se confermato da ulteriori analisi — portare a prestazioni migliori, a fronte di costi comunque piuttosto contenuti e una configurazione molto semplificata.

Bibliografia

[1] F. Tauro, F. Tosi, S. Mattoccia, E. Toth, R. Piscopia, e S. Grimaldi, "Optical tracking velocimetry (OTV): Leveraging optical flow and trajectory-based filtering for surface streamflow observations," *Remote Sensing*, vol. 10, no. 12, 2018. [Online]. Disponibile: https://www.mdpi.com/2072-4292/10/12/2010

[2] A.-H. Livoroi, A. Conti, L. Foianesi, F. Tosi, F. Aleotti, M. Poggi, F. Tauro, E. Toth, S. Grimaldi, e S. Mattoccia, "On the deployment of out-of-the-box embedded devices for self-powered river surface flow velocity monitoring at the edge," *Applied Sciences*, vol. 11, no. 15, 2021. [Online]. Disponibile: https://www.mdpi.com/2076-3417/11/15/7027

[3] F. Tosi, "Optical tracking velocimetry." [Online]. Disponibile: https://github.com/fabiotosi92/Optical-Tracking-Velocimetry

[4] F. Tosi, M. Rocca, F. Aleotti, M. Poggi, S. Mattoccia, F. Tauro, E. Toth, e S. Grimaldi, "Enabling image-based streamflow monitoring at the edge," *Remote Sensing*, vol. 12, no. 12, 2020. [Online]. Disponibile: https://www.mdpi.com/2072-4292/12/12/2047

[5] "Android ABIs." [Online]. Disponibile: https://developer.android.com/ndk/guides/abis

[6] "Java Native Interface Specification," Nov 2002. [Online]. Disponibile: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html

[7] "Understand the Activity Lifecycle." [Online]. Disponibile: https://developer.android.com/guide/components/activities/activity-lifecycle

[8] “Analyze power use with Battery Historian.” [Online]. Disponibile: https://developer.android.com/topic/performance/power/battery-historian

[9] “Profile battery usage with Batterystats and Battery Historian.” [Online]. Disponibile: https://developer.android.com/topic/performance/power/setup-battery-historian