Capitolo 3

Prototipo per esecuzione efficiente di codice Wasm con tecniche di Image Processing

Nel capitolo precedente sono state esaminate le differenze sostanziali tra un'approccio basato su Rust in combinazione con WebAssembly e uno basato su Node.js. In questo capitolo si presenterà il prototipo sviluppato con l'obiettivo di comprendere l'impatto di tali differenze in un'applicazione pratica.

3.1 Descrizione dell'applicazione

Il prototipo sviluppato è un'applicazione dedicata all'elaborazione digitale di immagini, concepita per simulare un contesto realistico in cui le operazioni richiedono una considerevole quantità di elaborazioni da parte della CPU.
Dato il limitato tempo disponibile, non è stato possibile esplorare in dettaglio il vasto campo del *digital image processing*. Pertanto, sono state impiegate librerie preesistenti in entrambi i linguaggi, evitando di immergersi eccessivamente nella programmazione di basso livello.
L'architettura dell'applicazione segue un modello client-server in entrambe le implementazioni. Il cliente ha il compito di fornire i file da elaborare insieme alle relative

specifiche sulle modifiche da apportare. Il server eseguirà le modifiche richieste e restituirà al cliente il percorso della nuova immagine, pronta per il download.
Nel processo di selezione delle possibili modifiche da apportare, è stato essenziale individuare due librerie nei rispettivi linguaggi utilizzati. Successivamente, per garantire uniformità nelle opzioni di modifica disponibili, sono state estratte le seguenti funzionalità comuni:

- ridimensionamento;
- rotazione di 90°;
- ribaltamento in orizzantale;
- conversione in bianco e nero;
- regolazione del contrasto;
- modifica della luminosità;

Tali operazioni sono state selezionate poiché rappresentano funzionalità frequentemente utilizzate anche da utenti comuni, oltre a caratterizzarsi per la loro eterogeneità. Alcune di queste coinvolgono esclusivamente la manipolazione dei pixel, come ad esempio la rotazione e il ribaltamento, mentre altre, come la conversione in scala di grigi o la modifica del contrasto/luminosità, comportano modifiche dirette ai pixel stessi.

Figura 3.1: Esempio di immagine a cui sono state apportate tutte le elaborazioni specificate.

3.2 Implementazione in Rust e Wasm/WASI

Per quanto riguarda l'implementazione si è deciso di partire dal prototipo sviluppato in Rust poiché rappresentava l'aspetto più innovativo e richiedeva un considerevole impegno in termini di tempo.
Si è resa inoltre necessaria la ricerca di un framework che consentisse la creazione di un web server per la gestione delle richieste utente.
La scelta è ricaduta su actix-web, un web framework potente ed estremamente veloce per Rust.
Il client è costituito da una semplice pagina HTML contenente un form per il caricamento delle immagini e due riquadri che mostrano l'immagine pre e post modifiche. Tale pagina eseguirà una richiesta AJAX al server, inviando il file da elaborare e le relative specifiche sulle modifiche.
Terminata l'elaborazione il server restituirà il percorso della nuova immagine pronta per essere scaricata.

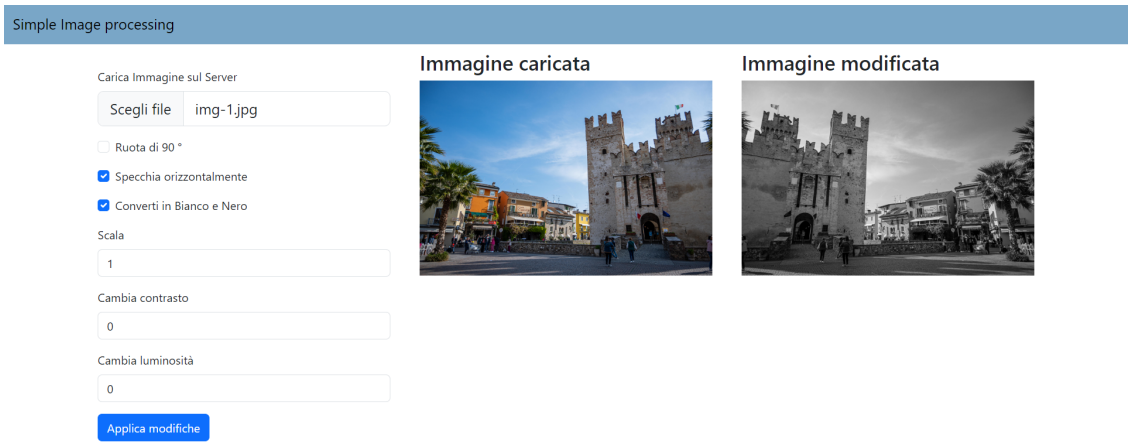

Figura 3.2: View nel browser del client

3.2.1 Actix-Web

Lo studio del funzionamento del framework Actix-Web ha costituito un punto focale durante la fase iniziale dell'implementazione.

Tale framework si è dimostrato estremamente flessibile ed adatto per lo sviluppo del prototipo in questione. Grazie all'impiego di **extractors**, di **handlers** e di altre funzionalità integrate, la gestione di richieste HTTP è risultata intuitiva e diretta.

Al fine di garantire un codice più ordinato e manutenibile è stata presa la decisione di strutturare l'applicazione in diversi file. Tra questi, il file primario è **main.rs**, che svolge un ruolo centrale nell'architettura complessiva.

Configurazione server

```
#[actix_rt::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(move || {
        App::new()
            .route("/", web::get().to(server::handlers::index))
            .route("/upload", web::post().to(server::handlers::upload))
```

```
7               .service(fs::Files::new("/script", "./src/static/"))
8               .service(fs::Files::new("/img", "./img/"))
9               .app_data(state.clone())
10      })
11      .bind("127.0.0.1:8080")?
12      .run()
13      .await
14  }
```

Listato 3.1: Porzione del file main.rs

Nel frammento di codice fornito, viene presentata la creazione di un HttpServer e un'istanza di App tramite il framework Actix-Web. In Actix-Web, ogni server è costruito attorno a un'istanza di App, che consente di configurare le "regole di routing" per risorse di vario tipo, registrare servizi HTTP e gestire stato di livello applicazione.
Nel caso specifico, vengono configurate due regole di routing:

- La prima regola gestisce le richieste dirette alla **home** del sito: in questo caso verrà semplicemente restituita la pagina index.html presente nella directory "static".
- La seconda regola gestisce richieste all'endpoint **upload**, per le quali è necessario ricevere un'immagine e le relative modifiche da apportare all'interno del modulo WebAssembly. Successivamente, verrà restituito il percorso della nuova immagine elaborata.

Si sottolinea che entrambe le regole di routing mappano le richieste a funzioni presenti all'interno del file **handlers.rs** nel modulo server.
Successivamente sono stati registrati due servizi HTTP per garantire l'accesso a risorse statiche: script Javascript e immagini elaborate dall'applicazione.
Infine, dopo aver configurato il metodo di dispatching delle richieste ed aver messo a disposizione le risorse necessarie ai client, si è passati alla scrittura del codice nel file **handlers.rs**. Tale file, come suggerito dal nome, contiene gli handler delle richieste HTTP, con la conseguente esecuzione di codice WebAssembly per richieste di modifica di immagini.

```rust
#[derive(MultipartForm)]
pub struct ImageUpload {
    image: TempFile,
    scala: Text<f32>,
    contrasto: Text<f32>,
    luminosita: Text<i32>,
    ruota: Text<bool>,
    specchia: Text<bool>,
    bw: Text<bool>
}
#[derive(Serialize, Deserialize)]
pub struct Editings{
    scala: f32,
    ...
}
pub async fn index() -> HttpResponse {
    HttpResponse::Ok().content_type("text/html")
        .body(include_str!("../static/index.html"))
}

pub async fn upload(form: MultipartForm<ImageUpload>) -> HttpResponse {
    let time = SystemTime::now().duration_since(SystemTime::UNIX_EPOCH);
    let filepath = format!("img/uploaded/{:?}_{}", time,
        form.0.file_name.as_str());
    match form.0.image.file.persist(filepath) {
        Ok(_) => {
            let editings = Editings{
                scala : form.0.scala.0,
                ...
            };
            edit(editings)
        },
```

```
33          Err(e) => {
34              HttpResponse::InternalServerError().finish()
35          },
36      }
37  }
```

Listato 3.2: Operazioni principali presenti nel file handlers.rs

Nel codice quì presentato, si possono facilmente notare i due endpoint del prototipo: la funzione **index()** per richieste alla pagina home e la funzione **upload()** per richieste di elaborazione di immagini.
Per quanto riguarda la prima tipologia si risponde ai client semplicemente fornendo il file HTML statico "index.html".
Al contrario la gestione delle richieste di upload è notevolmente più articolata e per questo motivo coinvolge una funzione di supporto denominata **edit()**. Sarà questa funzione che si occuperà dell'istanziazione del modulo WebAssembly/WASI e della sua esecuzione.
Si sottolinea anche il parametro ricevuto dalla funzione "upload": un **MultipartForm<ImageUpload>**. Il framework Actix-Web, con il supporto del crate actix-multipart semplifica notevolmente la gestione di richieste provenienti da form html, anche in presenza di campi di input di tipo "file".
Ogni campo definito nella struttura "ImageUpload" corrisponde a un parametro proveniente dalla richiesta HTML e viene automaticamente popolato prima dell'invocazione dell'handler, consentendo un accesso immediato ai valori inviati al server.
In sintesi la funzione **upload()** rende persistente il file temporaneo ricevuto ed inoltre crea un ulteriore *struct* di supporto (**Editings**) che verrà sfruttata dalla funzione **edit()** per velocizzare il successivo accesso alle elaborazioni da effettuare.
Va sottolineato che "edit()" viene invocata solo se il file è stato reso persistente senza errori. Infatti, tale funzione si occupa dell'istanziazione del modulo WebAssembly e non ha senso tentare di eseguire modifiche su un file che non è stato salvato correttamente.

3.2.2 Integrazione modulo Wasm/WASI

Come precedentemente indicato, per l'esecuzione di codice WebAssembly, si è deciso di utilizzare il runtime environment **Wasmtime**. La sua integrazione all'interno di un'applicazione Rust è resa possibile primariamente grazie ai crate **wasmtime**, **wasmtime-wasi** e **wasi-common**.

Scambio di dati

Prima ancora di iniziare l'implementazione è stato fondamentale trovare un modo per **scambiare dati** tra il modulo WebAssembly e l'host.
Per soddisfare i requesti imposti in questa tesi, è necessario che il modulo WebAssembly riceva tutte le elaborazioni da effettuare su un'immagine e il nome del file da modificare, ma come introdotto in sezione 2.1.3, una funzione WebAssembly può ricevere solamente valori di **tipo numerico**.
Analizzando le API WebAssembly e WASI sono emerse diverse soluzioni a questo problema. Tra queste l'utilizzo della memoria lineare, la scrittura su file, la modifica di variabili d'ambiente o la comunicazione attraverso standard input. Tuttavia, alcuni di questi approcci potrebbero risultare complessi da implementare e considerando che per l'applicazione sviluppata sarebbe stato sufficiente scambiare una stringa per ottenere tutte le informazioni necessarie, si è scelto di adottare la comunicazione tramite **standard input**.
Nello specifico, per consentire lo scambio di dati tra host e guest, verrà implementato un protocollo composto dalla seguente sequenza di operazioni:

- Serializzazione in formato JSON della struct "Editing" contenente tutte le elaborazioni e il nome del file da modificare;
- Inserimento sullo standard input del modulo Wasm, della stringa ottenuta dalla serializzazione;
- Lettura della stringa da standard input all'interno del modulo, sfruttando le API messe a disposizione da WebAssembly System Interface;
- Deserializzazione in una Struct Editing equivalente a quella di partenza;

Si noti che, se necessario, questo protocollo potrebbe essere applicato anche per ottenere strutture dati in output dal modulo WebAssembly.

Condizioni necessarie per l'esecuzione

Prima di poter eseguire un modulo tramite Wasmtime, sono necessarie diverse operazioni preliminari.
Dopo aver serializzato la struttura dati e predisposto una **ReadPipe** (crate wasi-common) per mettere a disposizione la stringa serializzata su standard input, la prima operazione necessaria è la creazione dell'**Engine** Wasmtime.
Esso rappresenta un contesto globale per la compilazione e l'esecuzione di moduli Wasm, che nel nostro specifico caso adotterà la configurazione di default.
Si procederà poi con l'istanziazione di un **wasmtime::Linker**. Il linker faciliterà l'istanziazione del modulo Wasm, risolvendo le diverse import (tra cui quelle per le syscall WASI).
Non bisogna poi dimenticare, che a causa dell'architettura di WASI, sarà possibile accedere ed utilizzare i file presenti in una certa directory, solamente se al programma sono state fornite le **capabilities** necessarie.
Per ottenere le capabilities per operare sulle immagini caricate dagli utenti, è necessario aprire la cartella "img" prima dell'istanziazione del modulo WASI.

```
pub fn edit(editing : Editings) -> HttpResponse {
    let serialized_input = serde_json::to_string(&editing);
    let stdin = ReadPipe::from(serialized_input);

    let engine = Engine::default();

    let mut linker: Linker<WasiCtx> = Linker::new(&engine);
    wasmtime_wasi::add_to_linker(&mut linker, |s| s);
    let  image_directory = Dir::open_ambient_dir("img",
    ambient_authority());
    ...
}
```

Listato 3.3: File handlers.rs: operazioni preliminari

A questo punto è possibile creare e configurare un contesto WASI tramite la struct **wasmtime_wasi::WasiCtxBuilder**.

Tramite tale oggetto si specifica che lo standard input sarà prelevato dall'oggetto contenente la serializzazione, lo standard output e lo standard error saranno ereditati dalla macchina host ed infine si fornisce una directory precedentemente aperta, che sarà disponibile al percorso "img".
Sfruttando il contesto WASI è ora possibile la creazione dello **Store** Wasmtime, l'oggetto designato per l'effettiva istanziazione del modulo WebAssembly e che successivamente conterrà tutte le funzioni, la memoria, le tabelle e lo stato interno del programma.

```
1    let builder = WasiCtxBuilder::new()
2    .stdin(Box::new(stdin.clone()))
3    .inherit_stdout()
4    .inherit_stderr()
5    .preopened_dir(image_directory,"img");
6    let wasi = builder.build();
7
8    let mut store = Store::new(&engine, wasi);
```

Listato 3.4: File handlers.rs: creazione di contesto WASI e Store

Ora risulta possibile la creazione del **Module** WebAssembly e il suo collegamento con il Linker per poi ottenere, tramite quest'ultimo, un'**istanza** relativa al Module e allo Store specificati nel metodo *linker.instantiate()*.

```
1    let module = Module::from_file(&engine,
     "src/server/image_proc_module.wasm");
2    linker.module(&mut store, "", &module)
3    let instance = linker.instantiate(&mut store, &module);
```

Listato 3.5: File handlers.rs: istanziazione modulo Wasm

Esecuzione del modulo WebAssembly

Avendo configurato Store e contesto WASI, il programma possiede già tutti gli argomenti necessari per il corretto funzionamento e l'unico passo rimanente consiste nell'effettiva esecuzione della funzione del modulo WebAssembly.
Per fare ciò è necessario ottenere un'istanza di **wasmtime::Func** tramite l'operazione

get_typed_func() sull'istanza WebAssembly. A questo punto l'invocazione della funzione richiesta è finalmente possibile grazie al metodo *Func::call()*.
Ad esecuzione terminata verrà eliminato lo store dalla memoria e restituito il percorso dell'immagine modificata al cliente.

```
1    let instance_main = instance.get_typed_func::<(), ()>(&mut store,
     "_start");
2    instance_main.call(&mut store, ());
3    drop(store);
4    HttpResponse::Ok()
5    .content_type("text/plain")
6    .body(e.modified_file_path)
```

Listato 3.6: File handlers.rs: invocazione funzione _start presente nel modulo Wasm

Si noti che in questi esempi di codice non è presente alcuna gestione degli errori. Tuttavia nel prototipo sviluppato, l'utente finale otterrà il nuovo percorso del file, solo nel caso in cui ciascuna delle operazioni illustrate sarà andata a buon fine.

3.2.3 Modulo WebAssembly/WASI

Il modulo WebAssembly risulta a questo punto piuttosto semplice.
Esso si occupa infatti della lettura dei parametri da **standard input** e della loro deserializzazione in una struct Editings.
Successivamente vengono utilizzati i metodi forniti dal crate **image** di Rust per aprire l'immagine ricevuta, modificarla secondo le specifiche dell'utente e salvarla nel percorso specificato.[17]

```rust
#[derive(Serialize, Deserialize)]
pub struct Editings{
    scala: f32,
    ...
}
fn main() {
    let mut serialized_params = String::new();
    std::io::stdin().read_to_string(&mut serialized_params);
    let editings : Editings = serde_json::from_str(&serialized_params);
    let mut img = image::open(editings.filepath)
    if editings.scala != 0.0 {
        let new_width = (img.width() as f32) * editings.scala;
        let new_heigth = (img.height() as f32) * editings.scala;
        img = img.resize(new_width as u32, new_heigth as u32,
    image::imageops::FilterType::Nearest);
    }
    if editings.ruota {
        img = img.rotate90();
    }
    ...
    img.save(editings.modified_filepath);
}
```

Listato 3.7: Codice Rust che successivamente verrà compilato in WebAssembly

Terminata l'esecuzione del modulo, il controllo ritornerà all'host che si occuperà dell'invio di una risposta adeguata al client:

- Se durante l'esecuzione del modulo Wasm tutte le operazioni sono terminate correttamente verrà restituito il percorso della nuova immagine generata;
- Altrimenti verrà restituito un messaggio di errore;

Terminata la scrittura del codice, è necessaria la compilazione per ottenere un modulo WebAssembly utilizzabile da wasmtime.
Ciò si può fare agovelmente grazie al package manager **cargo**, ed in particolare grazie al seguente comando:

```
cargo build --release --target wasm32-wasi
        Compiling image_proc_module v0.1.0 (.\image_proc_module)
            Finished release [optimized] target(s) in 1.21s
```

In questo caso risulta pressochè obbligatoria la presenza del flag **- - release**, in quanto diverse funzioni del crate image, sono estremamente lente se utilizzate in debug mode.
Infine si sottolinea che avendo svolto tutte le operazioni necessarie nel main, nel momento in cui l'host invocherà il metodo *get_typed_func()*, sarà necessario specificare la funzione denominata **_start**.

3.3 Implementazione in Node.js

Dopo aver terminato l'implementazione del prototipo mediante l'utilizzo di Rust e WebAssembly, si è proceduto con l'implementazione di un'applicazione dotata delle medesime funzionalità, questa volta utilizzando il runtime environment **Node.js**.
La seconda implementazione è risultata notevolmente più immediata. In particolare è stato possibile riutilizzare il client sviluppato precedentemente, senza necessità di un'analisi approfondita per l'integrazione di un modulo WebAssembly. Come verrà sottolineato in seguito, ciò ha permesso di ottenere un **codice** decisamente più **pulito** e di lunghezza proporzionata alla semplicità dell'applicazione sviluppata.
La ricerca si è dunque focalizzata sulla selezione di un moduli adeguati per la creazione di un'applicazione web in grado di supportare l'upload e l'elaborazione di immagini. La scelta è ricaduta sul web framework **express.js**, in combinazione con il middleware **multer** e la libreria **Jimp**.

3.3.1 Express.js

Express.js è un **web framework** che semplifica lo sviluppo di applicazioni web robuste e scalabili. Esso è stato riconosciuto come lo **standard de facto** in quest'ambito.
Il design di Express è relativamente minimale, tuttavia, grazie all'impiego di **plugin** e **middleware** come Multer, è in grado di fornire una vasta gamma di funzionalità.
Il framework inoltre rende più semplice il routing, gestendo richieste e risposte HTTP in modo semplice e diretto.
Per quanto concerne la nostra applicazione, dopo aver importato i moduli richiesti, è essenziale effettuare una configurazione breve ma precisa, per la gestione delle richieste necessarie e per la configurazione dell'upload dei file.

```
1    const express = require('express');
2    const app = express()
3    const port = 3000
4
5    app.use(express.static('./static'))
6    app.use(express.static('./img/modified'))
7
8    app.post('/upload', upload.single('image'), (req, res) => {
```

```
9       try{
10      ...
11      }
12      } catch (error){
13        console.log(error);
14        res.status(500).send('Error processing the file');
15      }
16    })
17
18    app.listen(port, () => {
19      console.log(`Server listening on port ${port}`)
20    })
```

Listato 3.8: Configurazione Express.js

Nello specifico viene creata un'applicazione Express tramite l'omonimo metodo *express()*. Successivamente tramite i metodi *express.use(express.static(...))* vengono rese accessibili ai client le directory "static"(contenente il file HTML della home e gli script correlati) e "img/modified" (necessaria per il download delle immagini elaborate dal server).
Il server viene poi messo in ascolto sulla porta 3000 e configurato per gestire richieste di tipo POST dirette all'endpoint "/upload".
A questo punto, per la corretta gestione del caricamento di immagini, risulta necessario configurare l'applicazione affinché sia in grado di ricevere richieste con Content-Type "multipart/form-data".

3.3.2 Multer

Come introdotto inizialmente, per la gestione di richieste provenienti da un form contenente campi di input di tipo file, è stato adottato il middleware **multer**.

```
1 const multer  = require('multer');
2 const storage = multer.diskStorage({
3     destination: 'img/uploaded/',
4     filename: (req, file, cb) => {
5     cb(null, file.originalname);
6     }
7 });
```

```
8  const upload = multer({ storage });
9  app.post('/upload', upload.single('image'), (req, res) => {
10     ...
11 })
```

Listato 3.9: Configurazione upload immagine

Una volta importato questo modulo, è necessario impostare, tramite il metodo *multer.diskStorage()*, la destinazione (per coerenza viene mantenuta la directory img/uploaded) e il nome del file appena caricato (viene utilizzato lo stesso nome del file fornito dall'utente).
In seguito attraverso l'utilizzo del metodo *multer()* otteniamo un'istanza, tramite la quale è possibile specificare, nei parametri della callback function per richieste all'endpoint upload, che un singolo file dovrà essere reso persistente, come precedentemente specificato nella variabile storage.

3.3.3 Jimp

Una volta terminata la configurazione del server tramite express.js, è possibile procedere con la modifica del file ricevuto.

```
1  app.post('/upload', upload.single('image'), (req, res) => {
2    try{
3      const uploadededFilePath =
           'img/uploaded/' + req.file.originalname
4      const newFileName =  Date.now() + req.file.originalname
5      const modifiedFilePath = 'img/modified/' + newFileName;
6
7      Jimp.read(uploadededFilePath, (err, img) => {
8        if (err) throw err
9        else{
10         img.scale(req.body.scala)
11            .rotate(req.body.ruota)
12            .mirror(req.body.specchia,false)
13            .contrast(req.body.contrasto/100)
14            .brightness(req.body.luminosita/100, function(){
15               if(req.body.bw) img.grayscale()
```

```
16                img.write(modifiedFilePath, function(){
17                  res.status(200).send(newFileName);
18                });
19            });
20        }
21      });
22
23    } catch (error){
24      res.status(500).send('Error processing the file');
25    }
26  }
```

Listato 3.10: Elaborazione immagine grazie ai metodi della libreria Jimp

Il file è aperto mediante l'impego del metodo **Jimp.read()**, il quale fornisce un'istanza utilizzabile per moodificare l'immagine secondo specifiche. Quest'ultime possono essere acquisite tramite l'oggetto req.body, passato alla callback function *app.post('/upload',...)*. Dopo aver apportato le opportune modifiche all'immagine, essa viene salvata nella cartella "img/modified" con un nuovo nome al fine di garantire univocità tra i vari file.
Infine, se tutte le operazioni precedenti vengono eseguite correttamente, verrà restituito all'utente il percorso dell'immagine modificata. In caso contrario, verrà restituito un messaggio di errore.

3.4 Metodologia di test

Per valutare le performance di ciascuna implementazione presentata, si è deciso di eseguire lo stesso **set di test** su entrambe.
In particolare è stata scelta un'elaborazione specifica tra quelle disponibili e durante la sua esecuzione sono stati misurati diversi parametri.
Per scegliere la tipologia di operazione si è empiricamente tentato di capire quale fosse, tra le elaborazioni implementate, quella più computazionalmente complessa. Eseguendo vari test su una stessa immagine e verificando la latenza è stato scelto il **resize** di un'immagine ed in particolare il resize del 10% (x0.9), al fine di manipolare un elevato numero di pixel e di ottenere un'immagine di dimensioni simili a quella iniziale.
In seguito sono state scelte le immagini di test e i parametri da misurare. Per quanto riguarda le immagini, si è scelto di utilizzare tre file con dimensione crescente:

- 1114 x 742 (0.82 Megapixel, 925 KiloByte);
- 2869 x 1912 (5.48 Megapixel, 4.63 MegaByte);
- 5578 x 3712 (20.7 Megapixel, 17.4 MegaByte);

Come parametri da misurare si è deciso di focalizzarsi su:

- Latenza per ogni richiesta;
- Utilizzo di CPU;
- Consumo di memoria;

3.5 Setup sperimentale

Per eseguire i test presentati è stato utilizzato un laptop con processore Intel Core i5-10210 (1.6 - 2.10 GHz), 8 GigaByte di Ram e sistema operativo Windows 11. Per quanto riguarda i software utilizzati, sono state scelte le ultime versioni al momento disponibili:

- Rust 1.71
- Wasmtime 11.0
- Node 18.17.1 (LTS)

Il seguente setup, pur non essendo propriamente tipico di un server, ha comunque permesso di osservare differenze notevoli tra i due apprrocci esaminati.

3.6 Valutazione delle prestazioni

Per ottenere un campione di risultati affidabile sono state eseguite 10 misurazioni per ogni immagine. In seguito sono stato calcolati il **valore medio** del campione, ottenuto come media aritmetica e la **deviazione standard**:

$$E[X] = \frac{1}{n}\sum_{i=0}^{n} x_i$$

$$\sigma_X = \sqrt{\frac{1}{n}\sum_{i=0}^{n}(x_i - E[x])^2}$$

Successivamente verranno presentati i grafici per ciascuno dei parametri scelti, utilizzando rispettivamente Rust in combinazione con WebAssembly e Node.js. Ogni grafico conterrà i risultati per ognuna delle tre immagini utilizzati, consentendo così anche una facile analisi visiva.

3.6.1 Latenza

Il primo parametro misurato è stato quello della latenza.
Nello specifico si è analizzato il tempo intercorso tra l'invio di una richiesta di elaborazione e la successiva risposta contenente il percorso della nuova immagine elaborata.
Per quanto riguarda Rust si sono ottenuti i seguenti risultati sperimentali:

$$E[X]_{small} = 850ms, \qquad \sigma_X = 154.94ms$$
$$E[X]_{medium} = 1487ms, \qquad \sigma_X = 144.99ms$$
$$E[X]_{large} = 4212ms, \qquad \sigma_X = 169.95ms$$

Relativemente a Node.js invece:

$$E[X]_{small} = 704ms, \qquad \sigma_X = 79.19ms$$
$$E[X]_{medium} = 3081ms, \qquad \sigma_X = 240.58ms$$
$$E[X]_{large} = 11335ms, \qquad \sigma_X = 282.73ms$$

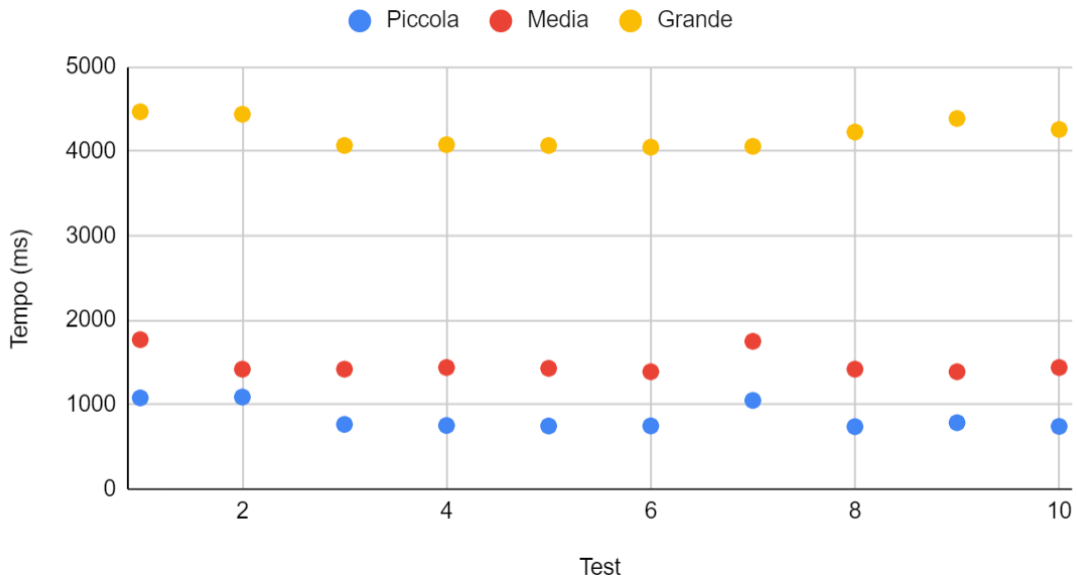

Figura 3.3: Latenza misurata utilizzando Rust in combinazione con WebAssembly.

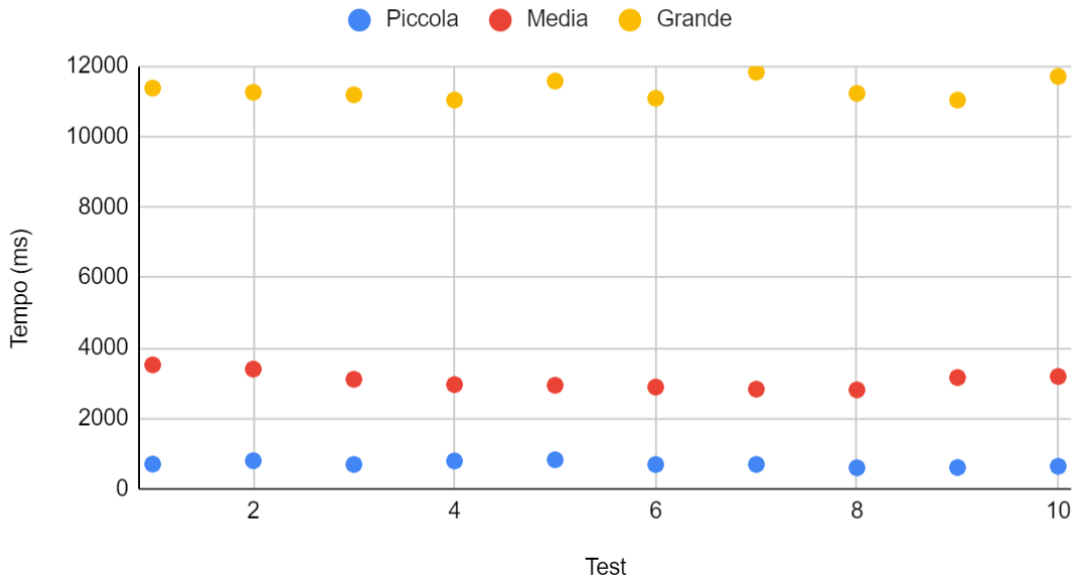

Figura 3.4: Latenza misurata utilizzando Node.js

Come si può notare il tempo di latenza risulta comparabile solamente durante l'elaborazione dell'immagine con dimensione minore. Durante l'elaborazione delle rimanenti due immagini, WebAssembly ha ottenuto performance notevolmente migliori, richiedendo circa un terzo del tempo necessario a Node.js per l'elaborazione

dell'immagine più grande.
È plausibile ipotizzare che, durante la modifica di immagini di piccole dimensioni, WebAssembly non riesca a mettere in evidenza i suoi vantaggi, soprattutto a causa dell'overhead introdotto dalla creazione di un'istanza del modulo e dalle operazioni correlate.
Tuttavia, quando il numero di operazioni a carico della CPU aumenta, emergono sia la tanto citata "velocità di esecuzione pari a quella del codice nativo" di WebAssembly che il bottleneck dovuto all'approccio event-driven di Node.js

3.6.2 Utilizzo CPU

In seguito è stato misurato l'utilizzo della CPU per ciascuno dei due approcci durante l'esecuzione di una richiesta, utilizzando le medesime immagini e impostazioni precedentemente indicate.
Anche in questo secondo test si è partiti dall'approccio basato Rust e sono stati ottenuti i seguenti risultati sperimentali:

$$E[X]_{small} = 16.09\%, \qquad \sigma_X = 3.32\%$$
$$E[X]_{medium} = 55.35\%, \qquad \sigma_X = 3.64\%$$
$$E[X]_{large} = 82.44\%, \qquad \sigma_X = 1.93\%$$

Relativemente a Node.js, invece:

$$E[X]_{small} = 20.87\%, \qquad \sigma_X = 6.89\%$$
$$E[X]_{medium} = 16.33\%, \qquad \sigma_X = 3.81\%$$
$$E[X]_{large} = 18.96\%, \qquad \sigma_X = 2.67\%$$

Come si può notare, in questo secondo test sono stati ottenuti risultati sostanzialmente differenti dal preceddente.
Nell'approccio basato su WebAssembly si può notare un utilizzo di CPU direttamente proporzionale alla dimensione dell'immagine elaborata, mentre utilizzando Node.js l'utilizzo rimane pressochè costante per tutte e tre le immagini esaminate.
Questi risultati mettono in luce come Rust, combinato con WebAssembly, riesca a sfruttare appieno le capacità computazionali della macchina quando le circostanze lo richiedono.

Al contrario, Node.js dimostra limitazioni nell'impiego ottimale della CPU, traducendosi in tempi di latenza maggiori per richieste ad alta intensità computazionale.

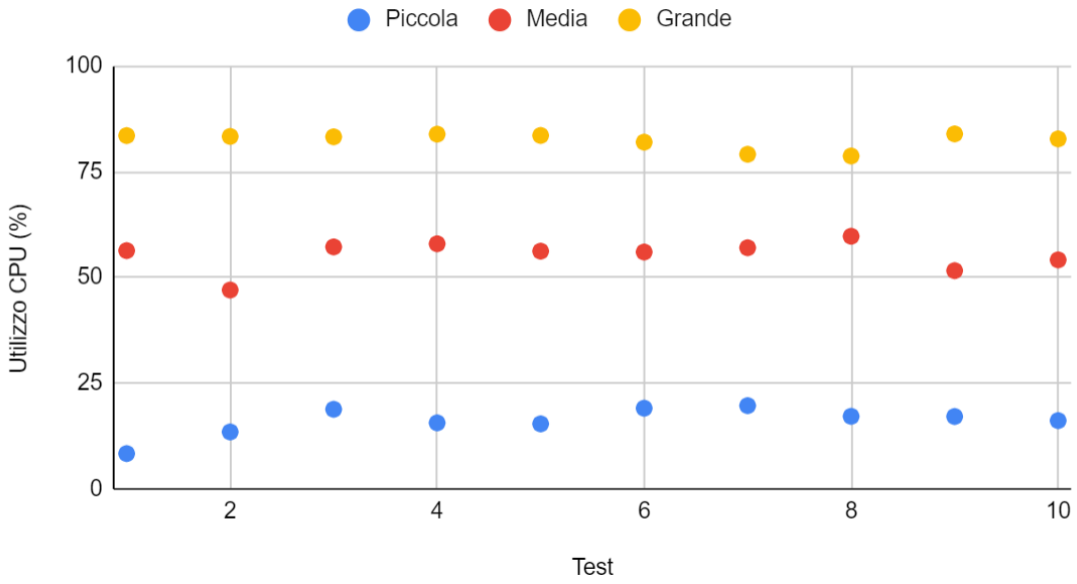

Figura 3.5: Utilizzo CPU misurato utilizzando Rust in combinazione con WebAssembly.

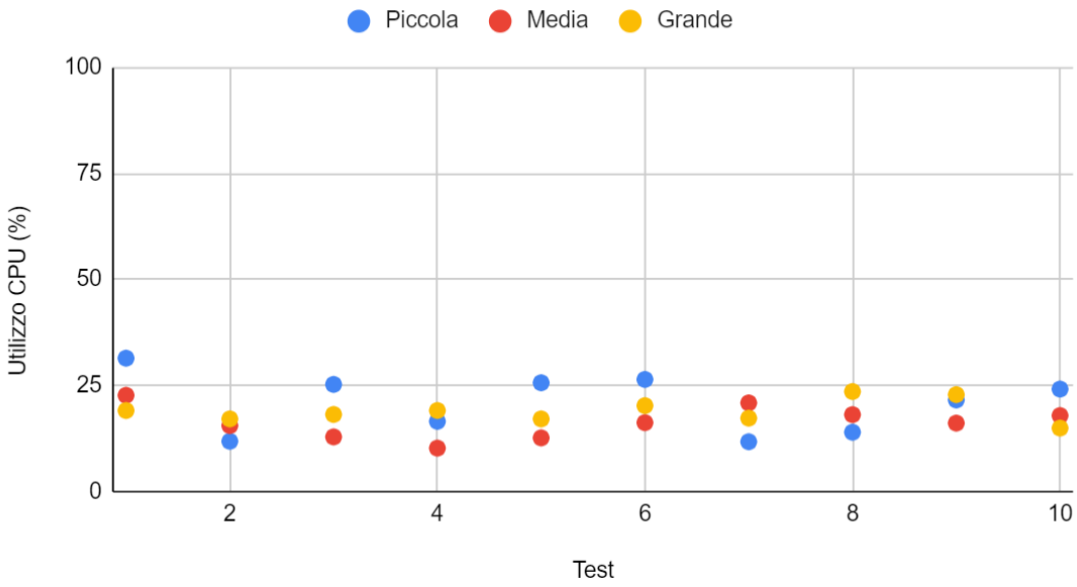

Figura 3.6: Utilizzo CPU misurato utilizzando Node.js

3.6.3 Consumo memoria

Come menzionato in precedenza, l'ultimo test eseguito riguarda il consumo di memoria durante l'esecuzione di una richiesta di elaborazione.
Per quanto riguarda Rust sono stati ottenuti i seguenti risultati relativi al consumo di memoria:

$$E[X]_{small} = 50.24\text{MB}, \qquad \sigma_X = 4.91\text{MB}$$
$$E[X]_{medium} = 48.79.\text{MB}, \qquad \sigma_X = 1.02\text{MB}$$
$$E[X]_{large} = 47.51.\text{MB}, \qquad \sigma_X = 0.90\text{MB}$$

Per quanto riguarda Node.js, invece:

$$E[X]_{small} = 80.51\text{MB}, \qquad \sigma_X = 2.73\text{MB}$$
$$E[X]_{medium} = 79.80\text{MB}, \qquad \sigma_X = 1.16\text{MB}$$
$$E[X]_{large} = 79.92\text{MB}, \qquad \sigma_X = 1.55\text{MB}$$

Durante quest'ultima analisi, i risultati ottenuti sono stati molto simili per entrambi gli approcci, ma sostanzialmente differenti rispetto ai test precedenti.
In particolare, Node.js presenta un consumo di memoria leggermnete maggiore, ma in entrambi i casi, tale parametro viene mantenuto pressochè costante per ciascuna tipologia di immagine, non presentando differenze significative.

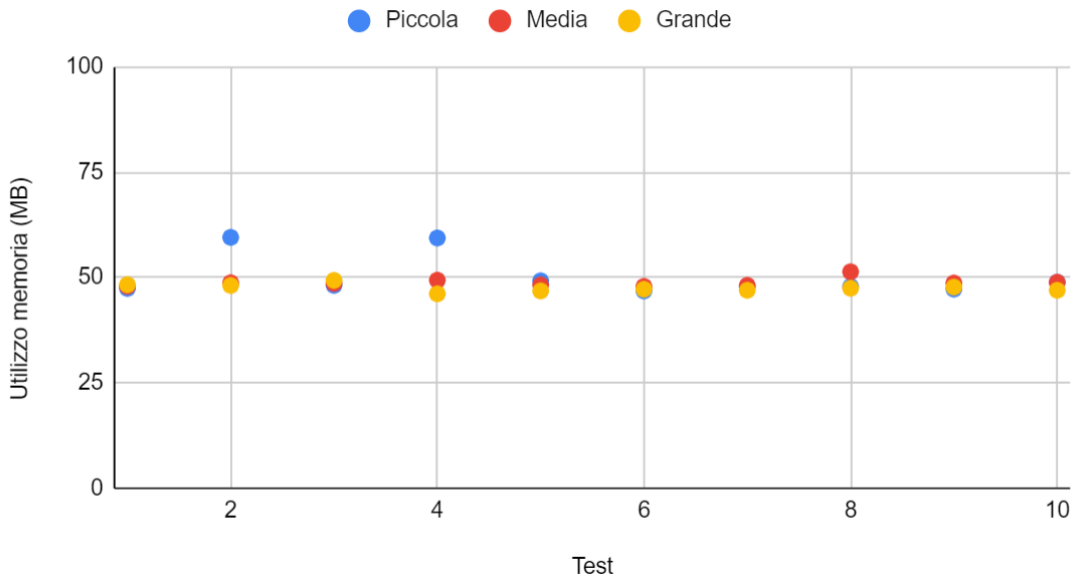

Figura 3.7: Consumo di memoria misurato utilizzando Rust in combinazione con Wasm.

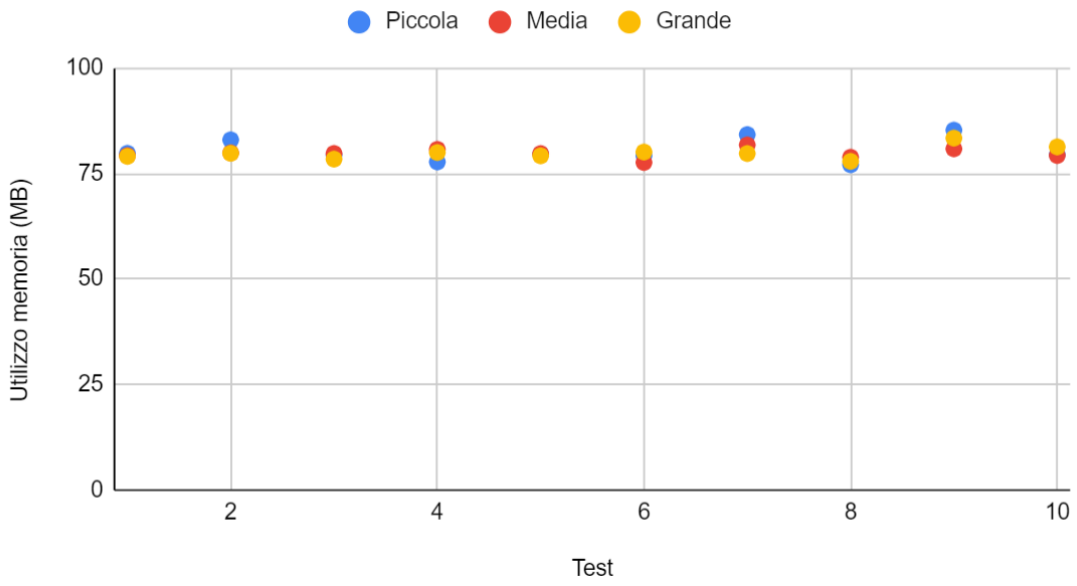

Figura 3.8: Consumo di memoria misurato utilizzando Node.js

3.7 Conclusioni

Per concludere questo lavoro di tesi, vengono presentate le considerazioni derivate dai risultati precedentemente esposti.

In primo luogo, è emerso che l'approccio basato su Rust e WebAssembly rappresenta una **scelta eccellente** in situazioni caratterizzate da operazioni ad **alta intensità computazionale**. Tuttavia, si rende necessaria un'analisi approfondita per determinare le funzionalità dell'applicazione che potrebbero effettivamente beneficiare di un'implementazione attraverso WebAssembly.

L'integrazione di codice Wasm all'interno di un programma Rust si è rivelata tutt'altro che immediata, comportando inoltre un **overhead** non trascurabile per un'applicazione che deve gestire un elevato volume di richieste.

D'altro canto Node.js offre un approccio di sviluppo più lineare e accessibile, specialmente per coloro che provengono dalle tecnologie web e posseggono una conoscenza consolidata del linguaggio JavaScript.

Tuttavia, l'adozione di Node.js potrebbe introdurre ulteriori sfide in quanto, per sua natura, risulta più indicato per applicazioni che fanno un ampio utilizzo del file system e in seguito alla debole tipizzazione potrebbe introdurre un numero di controlli molto elevato rispetto a Rust.

Durante lo svolgimento dei test è stato possibile notare, che Node.js introduce tempi di latenza peggiori rispetto a Wasm già con immagini relativamente piccole per gli standard odierni (5 Megapixel, 4,6 MegaByte). Inoltre è stata evidenziata una scarsa capacità nello sfruttare appieno la CPU, con un utilizzo che è rimasto praticamente costante durante tutti i test.

In conclusione è possibile affermare che l'utilizzo di WebAssembly server side può rappresentare una **scelta ottimale** per applicazioni che devono gestire richieste comportanti un elevato numero di calcoli da parte della CPU. Tuttavia, è importante evitare un uso eccessivo di questa tecnologia per richieste troppo semplici, al fine di prevenire un overhead non necessario e che potrebbe essere evitato anche utilizzando un linguaggio interpretato come JavaScript.