# Programmazione di Sistema

# STONNX


- Alessio Rosiello – s317659
- Tcaciuc Claudiu Constantin – s317661


Politecnico di Torino

# Indice

# Requisiti

Il progetto consiste nella realizzazione di un interprete ONNX utilizzando il linguaggio Rust. Le richieste da rispettare sono le seguenti:

Creazione di un parser per estrarre dal file ONNX l'informazione necessaria per la creazione della rete

Implementazione di un sotto-set di operatori ONNX

Utilizzo di parallelismo per l'esecuzione della rete

(Opzionale) binding con altri linguaggi.

# Modalità di Utilizzo

- Scaricare i modelli ONNX supportati:
  - `download_models.ps1` (windows)
  - `download_models.sh` (linux/MacOS)
- Compilare il progetto:
  - `cargo build --release` (utilizza rayon)
  - `cargo build –-release --features custom-threadpool` (utilizza il nostro ThreadPool personale)
- Esegui l'inferenza con un modello a scelta:
  - `cargo run --release -- --model <model_name>`
  - `cargo run --release --features custom-threadpool -- --model <model_name>`

# Modelli Supportati

| | | | | |
|---|---|---|---|---|
| AlexNet | MobileNet | GoogleNet | ResNet | GPT2 |
| Emotion | CaffeNet | Inception | Mnist | SqueezeNet |
| Shufflenet | Super Resolution | VGG | ZFNet | |

Questi modelli sono quelli testati più a fondo e con gli output confermati corretti, tuttavia il nostro programma porta correttamente a completamento 73 modelli su 184*.

* I modelli sono quelli del vecchio ONNX Model Zoo, prima che venisse rimodernato

# Struttura del codice

- `src/main.rs`
  - Contiene il main del programma e la gestione degli argomenti da linea di comando
- `src/operators`
  - Contiene i file con le implementazioni degli operatori ONNX
  - Gli operatori sono stati implementati seguendo la reference in Python dal repository ufficiale di ONNX
- `src/onnxparser`
  - Contiene i file con l'implementazione del parser per estrarre le informazioni dal file ONNX
- `src/executor`
  - Contiene l'implementazione per l'esecuzione della rete, sono presenti:
    - logica per la creazione del grafo e l'esecuzione degli operatori;
    - logica per la creazione di un pool di thread (personalizzato o utilizzando la libreria rayon) e la gestione della comunicazione tra i thread;
    - logica per il confronto degli output attesi con quelli ottenuti;

# Struttura del codice

- `src/parallel`
  - Contiene i file per l'implementazione del parallelismo con ThreadPool personale;
    - Il parallelismo è implementato in due modi diversi:
    - utilizzando la libreria rayon (di default);
    - utilizzando un thread pool custom (attivabile con il flag `--features custom-threadpool`);
- `src/protograph`
  - Contiene i file per l'implementazione della creazione di un file `.json` contenente il grafo della rete;
- `src/protos`
  - Contiene il file `onnx.proto` utilizzato per la creazione del file `.rs` contenente le strutture dati per la gestione dei file protobuf;
- `src/common/mod.rs`
  - Contiene le strutture dati utilizzate per la gestione dei file ONNX
- `src/utils`
  - Gestione di operazioni utili per la creazione dei tensori e la gestione di questi ultimi;

Si può inoltre vedere la documentazione creata con cargo doc qui.

# Architettura del progetto

- Deserializzazione e lettura del file `.onnx`
  - processo di deserializzazione e inizializzazione dei tensori con input esterni.

- Creazione di 2 grafi, rappresentati da due `HashMap`:
  - uno che connette ogni operatore ai suoi input (che chiameremo «dipendenze»)
  - uno che connette ogni input (tensore) agli operatori in cui viene usato

- Esecuzione dell'inferenza
  - esecuzione in parallelo partendo dagli operatori senza dipendenze o con dipendenze soddisfatte.
  - successivo aggiornamento del grafo delle dipendenze quando un operatore completa l'esecuzione.

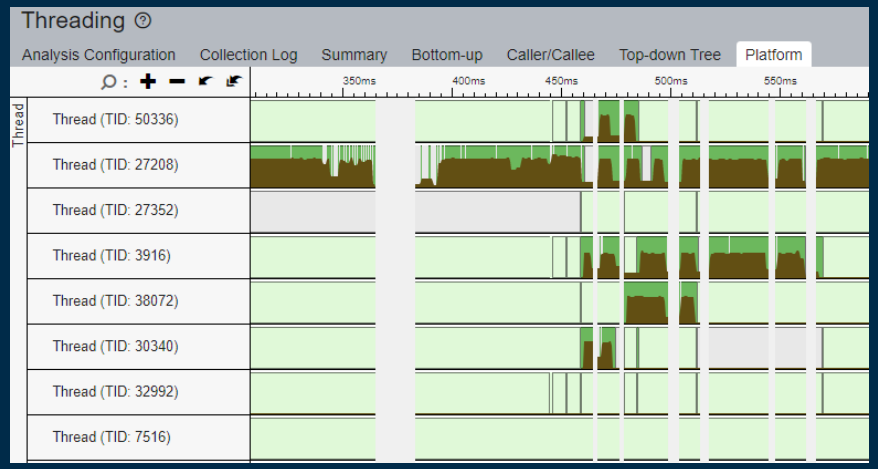
- Implementazione del `ThreadPool personale`
  - struttura principale composta da
    - `Queue` -> coda di sincronizzazione per la gestione dei compiti
    - `Workers` -> collezione di thread
    - `QueueStateType` -> contatore per tracciare il numero di operazioni in coda

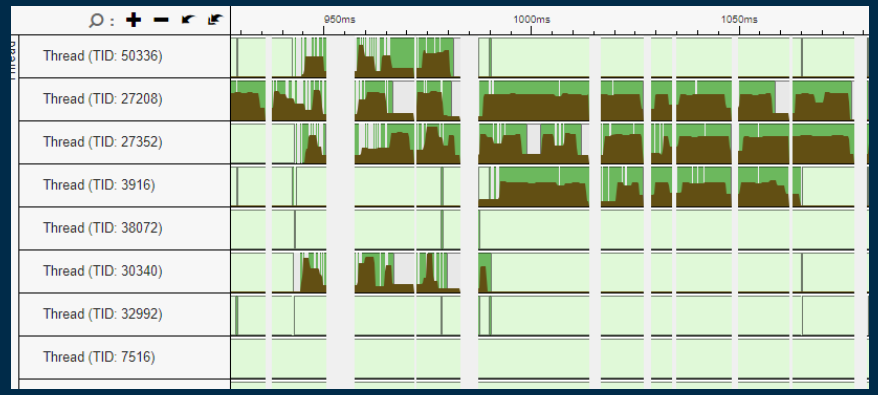
# Architettura del progetto

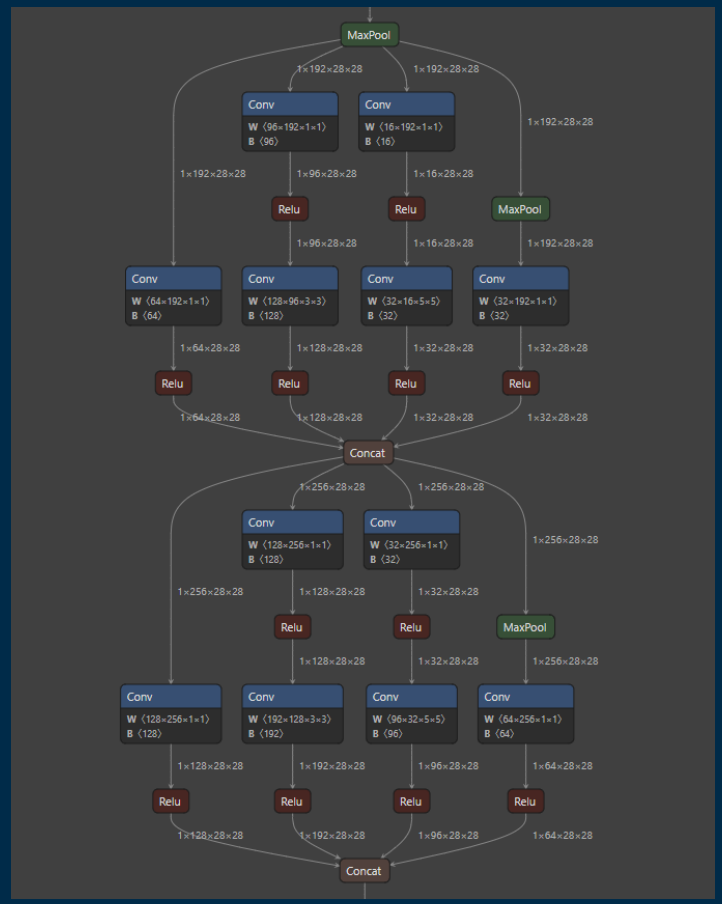
- Comparazione degli output
  - il programma legge inoltre anche gli output di "reference" che dovrebbero essere ottenuti dall'esecuzione del modello e li confronta con quelli effettivamente ottenuti.
  - controllando che la differenza tra i singoli valori dei due risultati sia massimo 10e-4 e stampando qualche statistica del confronto.

- Parallelismo ThreadPool personale vs Rayon
  - La ragione per la quale abbiamo implementato l'esecuzione concorrente della rete sia con un ThreadPool personalizzato sia con rayon, è principalmente per avere un modo per verificare la correttezza e performance della nostra implementazione, dopo aver effettuato i dovuti test e confronti, siamo giunti alla conclusione che nel nostro caso specifico, la nostra implementazione è allo stesso livello di Rayon, in quanto i tempi d'esecuzione e i risultati non cambiano.

# Esempio – GoogleNet

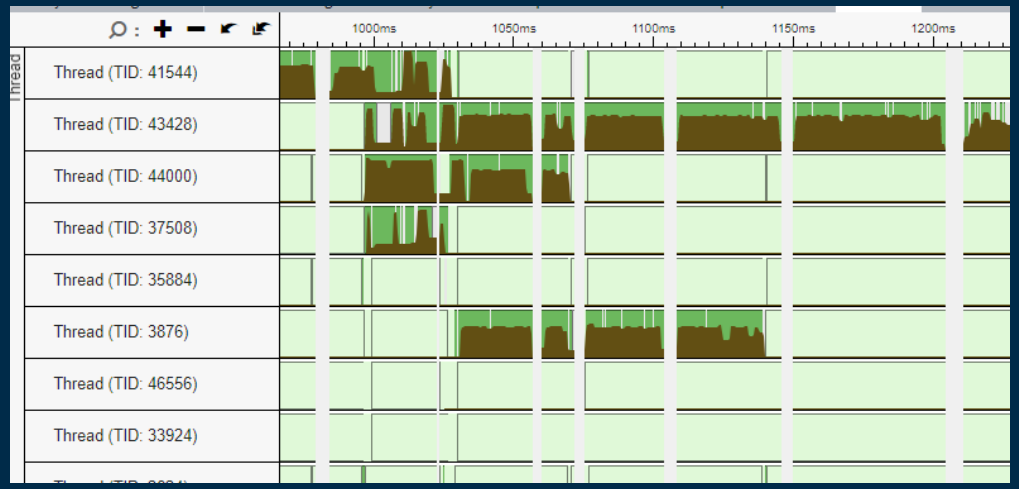


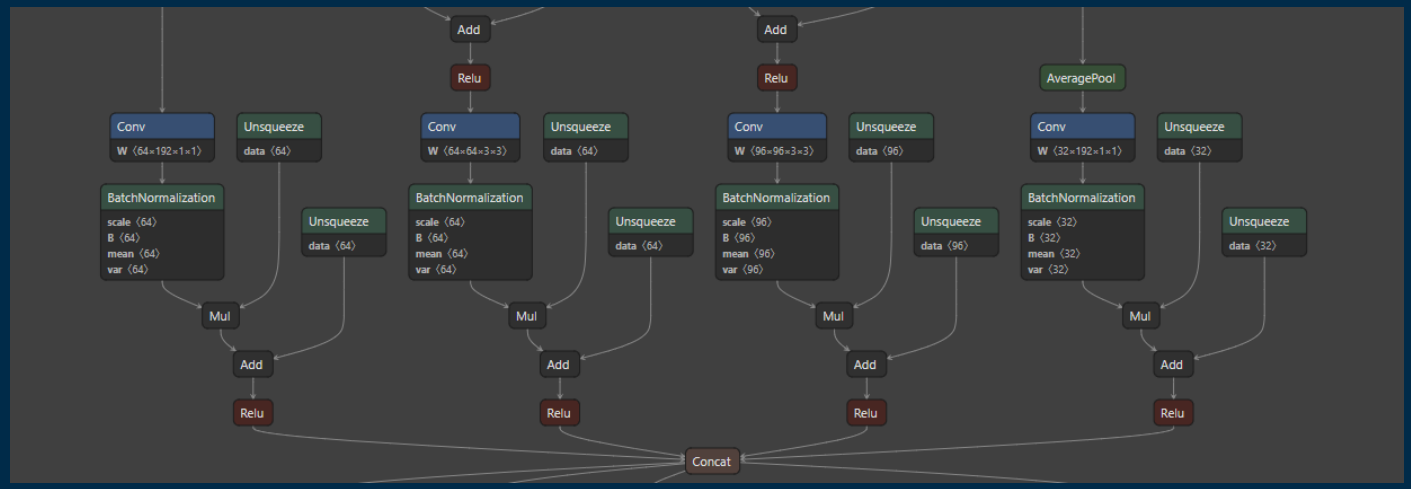




Esempio di uso dei thread in parallelo ottenuti con VTune associato alla porzione del grafico eseguito in quel lasso di tempo

# Esempio - Inception




Esempio di uso dei thread in parallelo ottenuti con VTune associato alla porzione del grafico eseguito in quel lasso di tempo



# Binding

- I binding, molto limitati per il momento, mettono a disposizione solamente alcune funzioni per la creazione e l'esecuzione della rete, possono essere facilmente espansi per fornire più funzionalità.

- Sono forniti verso i seguenti linguaggi
  - Python
  - C
  - C++
  - C#

# Benchmarks

- Benchmark effettuati su una macchina con:
  - CPU: Intel Core i7-12700K
  - RAM: 32 GB DDR4 3600 MHz
  - Ubuntu 22.04.3 LTS su WSL
  - Rustc 1.74.1 (a28077b28 2023-12-04)

- Esempi

- GPT2

```
hyperfine --warmup 2 './target/release/stonnx --verbose 0 --model GPT2'
Benchmark 1: ./target/release/stonnx --verbose 0 --model GPT2
  Time (mean ± σ):     490.1 ms ±  21.3 ms    [User: 352.5 ms, System: 530.9 ms]
  Range (min … max):   466.6 ms … 534.4 ms    10 runs
```

- GoogleNet

```
hyperfine --warmup 2 './target/release/stonnx --verbose 0 --model googlenet-12'
Benchmark 1: ./target/release/stonnx --verbose 0 --model googlenet-12
  Time (mean ± σ):      1.361 s ±  0.071 s    [User: 11.179 s, System: 1.323 s]
  Range (min … max):    1.252 s …  1.445 s    10 runs
```

Per tutti gli altri benchmarks visitare BENCHMARKS.md