Progetto 2: Analisi Real-Time di difetti nella produzione L-PBF con Apache Flink

Alessandro Finocchi
matricola n. 0340543
Università di Roma Tor Vergata

Giuseppe Marseglia
matricola n. 0350066
Università di Roma Tor Vergata

***Panoramica*—La Laser Powder Bed Fusion (L-PBF) è una tecnologia di stampa 3D impiegata per realizzare parti in metallo a partire da un letto di polveri metalliche fuse per mezzo di un laser, strato dopo strato. Questo approccio consente una maggiore flessibilità progettuale e riduce gli sprechi di materiale rispetto ai metodi tradizionali. Nonostante i suoi vantaggi, la L-PBF è soggetta a diversi tipi di difetti: tra i più critici vi è la porosità, ovvero la presenza di vuoti all'interno del materiale, che può compromettere significativamente la resistenza meccanica e la durata del prodotto finale.**
Rilevare i difetti solo a posteriori comporta uno spreco di tempo, energia e risorse, poiché i pezzi difettosi devono essere scartati. Per ridurre queste inefficienze, possono essere utilizzate tecniche di monitoraggio e di rilevazione dei difetti in tempo reale. La tomografia ottica (OT) consente di catturare immagini termiche del letto di polvere a ogni strato del processo. Queste immagini mostrano la distribuzione delle temperature e, se analizzate correttamente, possono rivelare irregolarità legate a potenziali difetti. L'analisi in tempo reale dei dati OT consente interventi immediati per prevenire il degrado della qualità.

***Indice dei termini*—Docker-compose, L-PBF, Apache Flink, JMP**

I. Introduzione

Si vogliono analizzare le immagini OT, provenienti dal Dataset della conferenza ACM DEBS 2025, il quale è fornito tramite un server REST chiamato **challenger**. Ogni letto è suddiviso in strati detti **layers**, e ogni strato è partizionato in sezioni dette **tiles**, che sono gli elementi dello stream fornito dal challenger. In particolare, ogni elemento ha il seguente formato:

```
(seq_id, print_id, tile_id, layer, tiff)
```

dove `tiff` è l'immagine che codifica i valori della temperatura per ogni punto del letto.
Nella specifica assegnata sono indicate le query a cui il sistema deve rispondere più una fase intermedia di windowing:

- Q_1: **filtrare** i punti di ciascun tile il cui valore di temperatura è sotto la soglia di `5000` (da escludere) o sopra quella di `65000` (da identificare come difettosi, escludere e contare).
- *Windowing*: mantenere una **finestra** scorrevole degli ultimi `3` layer per analizzare l'evoluzione della temperatura tra layer consecutivi.
- Q_2: calcolare per ogni punto del layer più recente della finestra la **deviazione di temperatura locale**, e se tale valore supera la soglia di `6000` **classificare** il punto come outlier.
- Q_3: **clusterizzare** gli outlier identificati nella Q_2 con l'algoritmo DBSCAN e fornire i risultati al challenger.

Infine è richiesta la valutazione sperimentale dei **throughput** e dei tempi di **latenza** delle query sulla piattaforma di riferimento usata per la realizzazione del progetto.

Tra le richieste opzionali della specifica, quelle implementate sono state:

- **Ottimizzazione** della logica della Q2 sfruttando la simmetria spaziale tramite la convoluzione discreta.

Inoltre è stato implementato un ulteriore parallelismo che permette di processare più finestre relative alla stessa coppia `print_id`, `tile_id` contemporaneamente.

II. Architettura

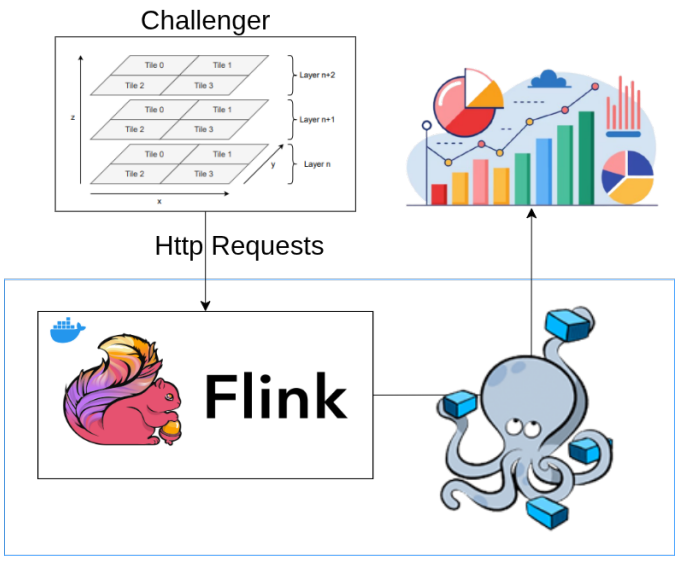

Fig. 1: Architettura

Il sistema è composto da 2 componenti, come riportato in Fig.1:

1) **REST Challenger**: container con il dataset .
2) **Apache Flink**: framework utilizzato per lo stream processing.

Questi componenti sono istanziati come servizi tramite l'orchestratore Docker Compose.
Le frecce in Fig.1 rappresentano i flussi di dati:

- Il challenger mette a disposizione un'interfaccia RESTful con cui interrogarlo per ottenere i tiles.
- Flink, tramite le sue API DataSource, effettua l'ingestion delle tiles nel sistema.
- Al completamento del processamento, tramite le API Sink di Flink, si salvano i dati, riguardanti le query e le metriche di performance, per poterli analizzare.

III. Deployment

A. Macchina host

Il deployment dell'architettura è stato effettuato su un nodo standalone tramite l'uso di container via Docker Compose. In particolare, il nodo standalone è una macchina Linux con 1 processore i7-1260P dotato di 12 cores e 16 GB RAM LPDDR5 (Dual Channel, 5200 MHz).

B. Flink deploy

La versione usata è la 2.0.0: è stato effettuato il deploy di Flink in modalità **Session** in modo da poter essere usato per eseguire molteplici job, soprattutto per semplificare la fase di benchmarking in cui occorreva eseguire il job con diverse configurazioni di parametri.

IV. Data ingestion

Il data ingestion è stato effettuato usando l'API `Source` nativa di Flink, senza aggiungere altri layer tra il processamento e la sorgente.
In Flink una Source ha tre componenti chiave:

- `Split`: la porzione di dati consumata dalla Source, quindi rappresenta la granularità con cui vengono consumati e parallelizzati i dati.
- `SourceReader`: richiede gli Split e produce stream di eventi/record; esegue quindi in parallelo sui taskmanagers.
- `SplitEnumerator`: genera gli Split e li assegna alle SourceReader; esegue come un'istanza singola nel jobmanager, mantenendo il backlog degli Split pendenti e bilanciando il carico tra i lettori.

Con questa distribuzione delle responsabilità è facile definire come i dati devono entrare nel sistema e configurarne il flusso. É all'interno di queste classi che è stata quindi incapsulata la comunicazione verso il challenger, le quali hanno messo a disposizione le loro politiche di failure detection di base e il ridirezionamento automatico del flusso dentro i task manager, in modo da poter dar vita al processamento.
Nel corso dell'implementazione delle API REST per comunicare con il challenger è stato di cruciale importanza utilizzare le specifiche delle API fornite insieme al challenger, dal momento che alcuni flussi di dati venivano serializzati in modo diverso (in particolare con il formato MessagePack e OctetStream). Il mapping delle risposte in formato JSON è stato effettuato con la nota libreria Jackson.

V. Processamento

Il processamento delle tile avviene in 4 passi:

1) Preprocessamento.
   - Lo scopo di questa fase è quello di convertire le informazioni della tile da byte di una immagine b/n in formato TIF a 16 bit ad una matrice di interi con valori da `0` a `65536`.
   - È implementata tramite operatore stateless e replicato, basato sull'operazione `map`.
2) Query 1: analisi basata su soglia.
   - Lo scopo di questa fase è contare quanti valori della matrice della tile superano la soglia `SATURATION_THRESHOLD=65000`. Questo valore verrà indicato come `saturated`.
   - L'output della fase deve indicare per ogni tile: il valore `saturated` calcolato.
   - È implementata tramite operatore stateless e replicato, basato sull'operazione `map`.
3) Query 2: windowing e analisi degli outlier.
   - Lo scopo di questa fase è individuare gli *outlier*, ovvero i punti la cui "deviazione di temperatura locale" supera la soglia `DEVIATION_THRESHOLD=6000`.
   - La **deviazione di temperatura locale** è il valore della "differenza assoluta" tra la media delle temperature calcolate sui "vicini prossimi" e quella calcolata sui "vicini esterni".
   - Ulteriori dettagli sono trattati successivamente.
   - L'output della fase deve indicare, per ogni tile, i 5 punti a "deviazione di temperatura locale" maggiore.
   - L'implementazione è più complessa ed è tratta in un seguente paragrafo.
4) Query 3: clustering degli outlier.
   - Lo scopo di questa fase è clusterizzare gli outlier ottenuti nella Query 2 "usando l'algoritmo DBSCAN, [e] usando la distanza euclidea come metrica.".
   - L'output della fase deve indicare per ogni tile:
     - Il valore di `saturated` calcolato in Q1.
     - Le coordinate x, y del centroide e il numero di punti per ogni cluster individuato.
   - L'output della fase viene anche fornito al `LOCAL-CHALLENGER`.
   - È implementata tramite operatore stateless e replicato, basato sull'operazione `map` e l'implementazione di `DBSCANClusterer` di Apache Commons Math.

A. Query 2

Alcuni dettagli riguardo al processamento richiesto dalla Query 2:

- I "vicini prossimi" e "vicini esterni" di un punto su una tile si trovano anche in altre tile oltre a quella in considerazione: quelle lo stesso `tile_id` e in una

finestra di 3 layer consecutivi (quello in considerazione e i 2 precedenti).
Come indicato nella specifica, "Questa finestra temporale consente l'analisi dell'evoluzione della temperatura tra layer consecutivi."

- I punti la cui temperatura è inferiore alla soglia `EMPTY_THRESHOLD=5000` o superiori alla soglia `SATURATION_THRESHOLD=65000` non possono essere considerati come outlier. Difatti questi indicano, rispettivamente, aree di vuoto e punti saturati che possono indicare difetti.
- Prendendo come riferimento l'implementazione in Python, si può notare che:
  - Per le tile appartenent al layer 0 e 1 non è possibile formare completamente la finestra di 3 layer: per esse non vengono calcolati gli outlier e di conseguenza i cluster.
  - Non vengono considerate le adiacenze tra tile dello stesso layer, ma alle tile viene applicato un padding con valore `0`.
    Ad esempio, considerando un solo layer, i punti sul bordo destro delle tile con `tile_id=0` sono adiacenti ai punti sul bordo sinistro delle tile con `tile_id=1`. Ciò comporta che parte dei vicini sono localizzati su tile con `tile_id` diversi.
    Nel calcolo della "deviazione di temperatura locale" il valore dei vicini che si trovano su queste tile viene assunto a `0` invece del reale valore. Questo permette un'implementazione più semplice e maggiormente parallelizzabile a discapito di una minore fedeltà.

Dalla specifica e dal relativo paper, sono delineati 2 approcci equivalenti per il calcolo della "deviazione di temperatura locale". Entrambi gli approcci partono dallo stack composto dalle tile con lo stesso `tile_id` di 3 layer consecutivi, dove la tile in cima allo stack è la più recente:

1) Approccio **naive**: per ogni punto della tile in cima allo stack, si calcolano esplicitamente la media delle temperature dei "vicini prossimi" e quella delle temperature dei "vicini lontani", e poi se ne prende il valore assoluto della differenza.
   - È l'approccio seguito dall'implementazione di riferimento in Python.
2) Approccio **kernel-based**: allo stack viene applicato un kernel 3D tramite un'operazione di convoluzione, seguito da un'operazione che prende il valore assoluto.
   - La convoluzione 3D può essere ulteriormente parallelizzata con un idea simile al **map-reduce**. Questo permette di aggregare le tile mano a mano che arrivano, invece di dover attendere che tutte le tile della finestra siano disponibili.
     Map: ad ogni tile della finestra, viene applicato un kernel 2D, slice di quello 3D, che dipende dalla profondità che quella tile ha all'interno della finestra.
     Reduce: quando 2 tile a cui è stata applicata la convoluzione sono disponibili, queste vengono aggregate tramite un'operazione di somma tra matrici, che è commutativa e associativa.
   - Un ulteriore ottimizzazione è quella che sfrutta la simmetria della computazione richiesta, precalcolando il numero di "vicini prossimi" e "vicini lontani" in modo da fare il calcolo differenza delle medie con un'unica iterazione.

Con $t_{i,j}$ si indica la tile del layer i-esimo e con `tile_id`=j.
In entrambe le implementazioni, la strategia per la parallelizzazione della Q_2 è stata la seguente:

1) All'ingresso della Q_2, il flusso di tile in entrata $t_{0,0}, t_{0,1}, \ldots, t_{0,16}, t_{1,0}, \ldots, t_{2,0}, \ldots$ viene **triplicato**, accoppiando alla tile un'indicazione della profondità che quella tile andrà a ricoprire nella relativa finestra, che verrà indicata come `depth`: $t_{0,0} \rightarrow (t_{0,0}, 0), (t_{0,0}, 1), (t_{0,0}, 2)$, dove $0, 1, 2$ rappresentano la `depth`.
   - Ciò è stato implementato con una serie di `map` e `union`, ma può essere anche implementato in una singola `flatMap`.
   - Nelle implementazioni con approccio naive, le tile vengono semplicemente copiate.
   - Nelle implementazioni con approccio kernel, alle tile viene applicata la fase map: viene quindi fatta la convoluzione con il kernel 2D corrispondente alla `depth` indicata.
2) Il flusso intermedio viene partizionato, tramite una `keyBy`, per una chiave composta da 3 elementi:
   - `print_id`: le finestre devono essere composte da tile dello stesso oggetto.
   - `tile_id`: le finestre devono essere composte da tile nella stessa posizione relativamente al proprio layer.
   - `layer_id + depth`: combinando in questo modo l'informazione del layer e della profondità è possibile avere una chiave per ogni finestra, che verrà popolata dalle sole 3 tile con stesso `print_id` e `tile_id` e in 3 layer consecutivi.

   Ad esempio, assumendo che tutte le tile abbiano lo stesso `print_id`, e avendo come chiave `tile_id`=j e `layer_id + depth`=k, la finestra verrà popolata dalle tuple $\{(t_{k,j}, 0), (t_{k-1,j}, 1), (t_{k-2,j}, 2)\}$, cioè dalle tuple con `tile_id`=j nei layer k-esimo, $(k-1)$-esimo e $(k-2)$-esimo.
3) Al flusso partizionato per chiave, viene applicato il meccanismo di windowing. In particolare ne sono state fatte 2 implementazioni:
   - Esplicitamente, con `countWindow` di dimensione fissata a 3.
   - Implicitamente, con `KeyedProcessFunction`.

4) Sulle finestre ottenute viene fatta la computazione necessaria al calcolo degli outlier. In particolare è stata implementata sia la tecnica naive sia quella basata sui kernel.

Quindi si ottengono 4 diverse implementazioni della Q2, corrispondenti alle combinazioni di {`countWindow`, `KeyedProcessFunction`} × {`naive`, `kernel`}.

Infine, alcuni dettagli sulle diverse implementazioni:

- Entrambe le implementazioni con `countWindow` presentano le seguenti problematiche:
  - In quanto la grandezza della finestra è fissata, i primi due layer vanno filtrati, poiché questi non riescono a comporre una finestra abbastanza grande da far attivare il trigger.
  - L'attivazione del trigger al riempimento della finestra impedisce tecniche che aggregano le tile mano a mano che arrivano.
  - Probabilmente a causa dell'elevato numero finestre che vengono aperte e della mancanza di un meccanismo esplicito di deallocazione, queste implementazioni funzionano solo per la prima parte dell'esecuzione, andando a causare un `java.lang.OutOfMemoryError` a causa dell'esaurimento del `Java heap space`. Questo problema si presenta sia in deployment locali sia in deployment clusterizzati, anche aumentando la quantità di spazio a disposizione.
- Queste problematiche sono state risolte utilizzando l'API di più basso livello delle `ProcessFunction`. Infatti, entrambe le implementazioni con `KeyedProcessFunction` presentano i seguenti vantaggi:
  - È possibile implementare finestre di dimensione variabile, evitando così di dover filtrare i primi 2 layer. Inoltre, è possibile implementare un meccanismo di processamento ad hoc per i primi 2 layer: come nell'implementazione di riferimento, viene assunto che non ci siano outlier.
  - L'API delle `ProcessFunction` permette di controllare esplicitamente lo stato dell'operatore tramite le variabili `ValueState`. Ciò permette di deallocare correttamente lo stato al momento del completamento del processamento della finestra e di evitare il problema dell'esaurimento del `Java heap space` presente nelle implementazioni con `countWindow`.
  - In quanto il processamento viene attivato per ogni elemento del flusso, è possibile implementare tecniche che aggregano le tile mano a mano che arrivano. In particolare, ciò è stato fatto per l'implementazione che sfrutta i kernel, che somma le matrici delle tile senza aspettare che tutte siano disponibili.
- Entrambe le implementazioni con approccio `naive` hanno il vantaggio di poter ignorare totalmente la computazione sui punti con temperatura inferiore alla soglia minima o maggiore della soglia massima. Ciò risulta vantaggioso in presenza di tile particolarmente vuote o con molti punti difettosi.
- Entrambe le implementazioni con approccio `kernel` hanno il vantaggio di poter beneficiare di accelerazione hardware. Nell'attuale implementazione, però, l'operazione di convoluzione non supporta accelerazione hardware.

VI. Uso del sistema

É stato realizzato un Makefile per rendere l'uso semplice e rapido:

1) Scaricato il progetto, la prima cosa da fare è l'unzip del file `micro-challenger/gc25cdocker.zip` nella stessa cartella in cui si trova.
2) Ora è tutto pronto per avviare la composizione di servizi Docker: il deploy può essere effettuato con un solo taskmanager tramite il comando `make gen` o con un numero di taskmanager n arbitrario tramite il comando `make gen_s TM=`n.
3) Per lanciare l'esecuzione dell'applicazione una volta e ottenere i risultati delle query in `/results/queries`, lanciare il comando `make q`.
4) Per lanciare l'esecuzione del benchmarking e ottenere le metriche di performance delle query in `performance_analyzer/input`, lanciare il comando `make b`.
5) Infine per fermare e rimuovere il cluster istanziato di container, lanciare il comando `make clean`.[1]

VII. Esperimenti

A. Introduzione

Gli esperimenti condotti sono mirati a valutare l'impatto sul throughput e sulla latenza di:

1) Numero di Task Manager.
2) Grado di replicazione degli operatori.[2]
3) Implementazione di Q2.[3]

Da notare che durante lo svolgimento degli esperimenti si è mantenuto coerente il background di ognuno di essi, che è stato eseguito sulla stessa macchina con il medesimo (leggero) carico di CPU di base: questo ha reso di conseguenza coerenti i risultati degli esperimenti, poiché i valori con cui si sono calcolati i timestamp da cui nascono le metriche provenivano dal medesimo clock fisico, e quindi i risultati dei diversi container sono comparabili.

[1] Gli errori di questo comando possono essere ignorati.
[2] tramite la `setParallelize`.
[3] solo con `KeyedProcessFunction`.

B. Metriche

Per la raccolta delle metriche si è cercato un approccio che fosse quanto più modulare e riutilizzabile, e la soluzione trovata è stata la classe `MetricsRichMapFunction`: questa estende la classe astratta `RichMapFunction` offerta da Flink per implementare un operatore map, agendo a tutti gli effetti come un tracker plug-and-play real-time della latenza e del throughput ad ogni elemento dello stream. La classe offre così un modo pulito e generico di strumentare una pipeline Flink con metriche di performance salvandole nello strato di persistenza sotto forma di CSV, in modo da poterli lavorare in analisi successive, e per come è stata idealizzata rende anche flessibile il suo uso e aperto a estensioni.

C. Analisi delle metriche

Il progetto contiene anche altri due sotto progetti indipendenti dall'applicazione Flink, creati appositamente per lo studio delle metriche:

- Il primo, **baseline**, è un progetto Python che estende il `client_ref.py` di base fornito dal challenger, in modo da poter reperire le performance dall'implementazione Python come in Flink.
  Ai fini della comparazione non è stato utilizzato poiché il tempo d'esecuzione delle run era dell'ordine delle ore, mentre quello di Flink nell'ordine dei pochi minuti. Viene ugualmente a disposizione per eventuali utilizzi.
- Il secondo, **performance-analyzer**, è un insieme di script bash e Python per analizzare e aggregare i risultati provenienti dalla `MetricsRichMapFunction`.
  - Una volta a disposizione i risultati in `performance_analyzer/input`, è possibile eseguire gli script `analyzer.py` e `challenger_agg.py` per ottenere i risultati aggregati in `performance_analyzer/output`.
  - I risultati ottenuti nello step precedente sono poi salvati per ogni esperimento dentro `results/metrics/metrics_raw`, e aggregati tramite lo script `aggregator.py`. I risultati aggregati sono visibili nelle Tab.I, II, III, IV e V.
  - Infine, gli script all'interno di `performance_analyzer/combiner` sono invece usati per ottenere i dati per la realizzazione dei grafici, in JMP Pro, riportati nelle figure 2, 3, 4, 5, 6 e 7.

D. Risultati

Esperimento 1: Throughput e latenza

Nell'esperimento 1 vengono confrontati i throughput e le latenze medie ottenuti su 20 run, al variare di:

- Numero di Task Manager $\in \{2, 4\}$.
- Grado di replicazione degli operatori $\in \{8, 16\}$.
- Implementazione di Q_2 con `KeyedProcessFunction` $\times\{$`kernel`, `naive`$\}$.

I risultati ottenuti sono riportati in Fig.2 e Fig.3

Dai risultati ottenuti si può dedurre che:

- Il throughput massimo si ottiene con la configurazione `2_8_naive`. In particolare, a questo throughput corrisponde una latenza quasi uguale a quella dell'esperimento con configurazione `2_8_kernel`.
  Questo, probabilmente, implica che:
  - La latenza simile indica che il tempo di computazione necessario dai 2 approcci è comparabile, specialmente considerando la mancanza di accelerazione hardware per l'approccio basato su convoluzione.
  - La latenza simile ma con throughput quasi raddoppiato, probabilmente, indica che la soluzione con i kernel presenta un minore grado di parallelismo all'interno della finestra. Ciò potrebbe essere dovuto a dettagli implementativi legato all'uso differente dello stato dell'operatore nelle 2 implementazioni.
- Raddoppiare il grado di replicazione da 8 a 16, comporta un raddoppiamento della latenza ma senza un aumento del throughput. Inoltre, la variazione del numero di Task Manager da 2 a 4 non comporta nessuna variazione sulle metriche considerate.
  Questo, probabilmente, implica che:
  - L'insufficiente disponibilità delle risorse reali sulla macchina host ha portato la maggiore allocazione di risorse virtuali ad introdurre ulteriori ritardi di comunicazione. Ciò è andato controbilanciare i benefici introdotti dal maggior parallelismo.

Esperimento 2: Valutazione per fase

Nell'esperimento 2 vengono confrontate le latenze medie per singola fase (non cumulative), al variare degli stessi parametri dell'esperimento 1.

I risultati ottenuti sono riportati in Fig.4 e in Fig.5.
Dai risultati ottenuti si può dedurre che:

- La fase più computazionalmente onerosa è la Q2, seguita da quella di preprocessamento. Q1 e Q3, invece, introducono latenze quasi trascurabili.
- L'implementazione con approccio naive presenta una variabilità della latenza maggiore, in particolare con una caratteristica più a heavy tail.
  Questo, probabilmente, è dovuto al fatto che l'implementazione naive può eseguire la maggior parte della computazione solo quando la finestra è completamente popolata, ed è quindi più sensibile a ritardi di tuple interni.

Esperimento 3: Andamento durante una run

Nell'esperimento 3, vengono confrontati i throughput e le latenze sperimentate dal sistema durante una singola run, in due configurazioni diverse: `2_8_naive` e `4_16_kernel`.

I risultati ottenuti sono riportati in Fig.6 e Fig.7.
Dai risultati ottenuti si può dedurre che:

- La latenza rimane stabile durante l'esecuzione in entrambi gli esperimenti.
  Questo, probabilmente, potrebbe essere causato dal fatto che il sistema è abbastanza ben dimensionato per gestire il carico o potrebbe essere effetto della back pressure, che sicuramente è attiva durante il processamento, come si può notare dalla Web UI.

VIII. Conclusioni

A. Commenti

Flink è decisamente un framework potente, che come nel nostro studio può essere usato in ogni fase di una pipeline di Big Data, dall'ingestion, al processing fino anche allo storing dei risultati, nonché anche al monitoring delle performance, ma "da grandi poteri derivano grandi responsabilità", e saper usare appieno il framework è una prerogativa per sfruttare al meglio le sue potenzialità.
I risultati ottenuti sono decisamente controintuitivi: ci si aspettava un vantaggio nell'utilizzo della soluzione di Q2 kernel-based, ma così non è stato, quindi sarebbe interessante studiare più precisamente quali sono le operazioni che hanno creato il collo di bottiglia e studiare configurazioni multidimensionali a grana più fine per ottimizzare il bottleneck del sistema e sfruttare appieno il parallelismo in funzione delle risorse a disposizione.

B. Sviluppi futuri

I possibili sviluppi futuri del progetto sono:

- Valutare l'impatto dell'accelerazione hardware sulle implementazioni con l'approccio `kernel`.
- Estendere il preprocessing effettuando un calcolo più preciso, in cui si considerano anche i punti vicini ma appartenenti ad altre tiles.
- Effettuare un confronto con il progetto `baseline`.
- Effettuare un confronto con l'implementazione del windowing con `CountWindow` usando come **event time** il `layer_id` e il partizionamento per chiave suggerito `(print_id, tile_id)`.

Tab. I: Metriche del challenger

config	thr	lat_min	lat_mean	lat_p99	lat_max
$(2, 16)$	39.6315	106	335.85	731	1255
pf_naive	71.8943	31	183.4	529	909
$(2, 8)$	40.7982	42	192.6	368	765
$(4, 16)$	38.921	76	342.35	798	2001
$(4, 8)$	42.2172	57	187	375	922

Tab. II: Metriche del preprocessamento

config	thr	lat_min	lat_mean	lat_p99	lat_max
$(2, 16)$	39.376	9	45.3521	117	479
pf_naive	70.8971	8	27.4063	69	321
$(2, 8)$	40.5274	8	22.2276	45	265
$(4, 16)$	38.6339	9	47.4202	133	999
$(4, 8)$	41.8333	8	21.5897	46	353

Tab. III: Metriche di q1

config	thr	lat_min	lat_mean	lat_p99	lat_max
$(2, 16)$	39.3749	10	48.0847	122	527
pf_naive	70.8946	9	28.8423	77	334
$(2, 8)$	40.5269	8	23.2994	46	292
$(4, 16)$	38.6327	10	50.6416	139	1113
$(4, 8)$	41.833	9	22.6884	48	409

Tab. IV: Metriche di q2

config	thr	lat_min	lat_mean	lat_p99	lat_max
$(2, 16)$	39.2675	77	319.7508	569	1133
pf_naive	70.6769	28	175.1758	508	906
$(2, 8)$	40.4494	39	185.7348	309	601
$(4, 16)$	38.523	54	324.1195	589	1752
$(4, 8)$	41.7462	47	180.2032	322	724

Tab. V: Metriche di q3

config	thr	lat_min	lat_mean	lat_p99	lat_max
$(2, 16)$	39.2667	81	329.0175	700	1174
pf_naive	70.6745	30	180.0854	526	907
$(2, 8)$	40.4492	40	190.113	362	607
$(4, 16)$	38.5224	62	334.54238	767	1753
$(4, 8)$	41.7458	49	184.4492	364	725

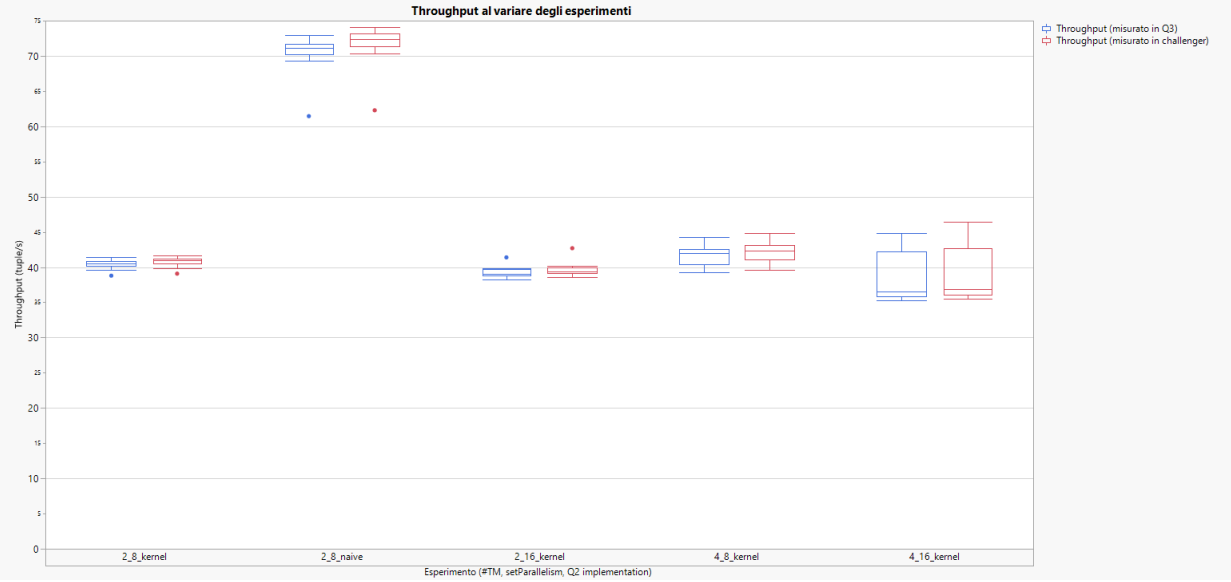

Fig. 2: Throughput al variare degli esperimenti.

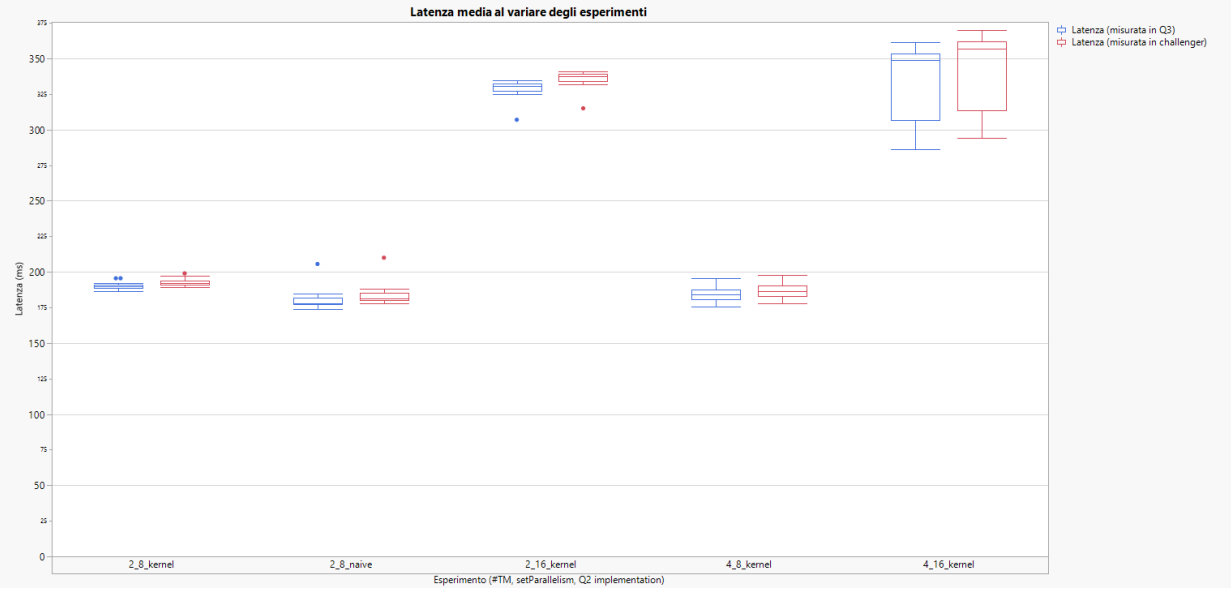

Fig. 3: Latenza media al variare degli esperimenti.

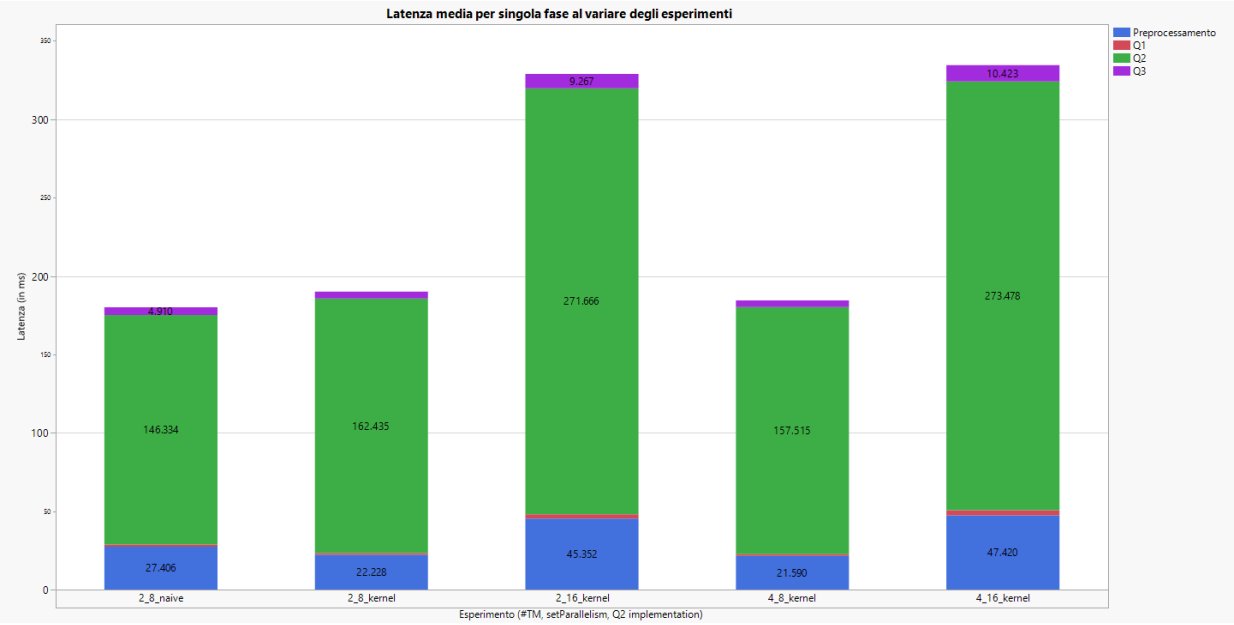

Fig. 4: Latenza media per singola fase al variare degli esperimenti.

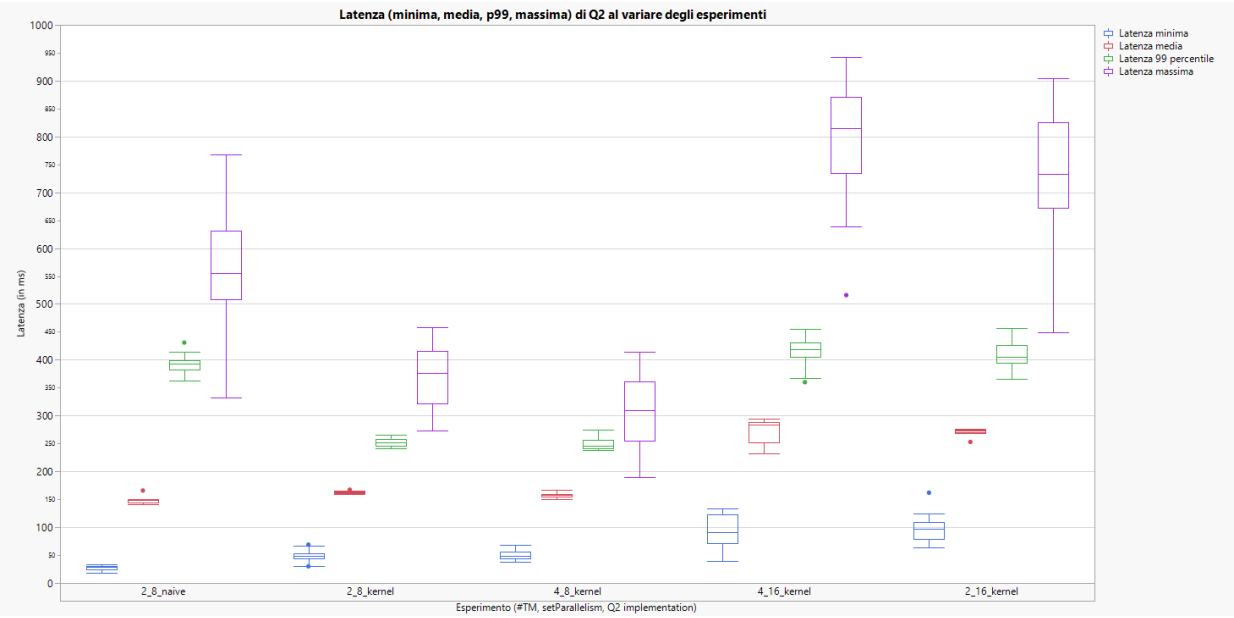

Fig. 5: Latenza minima, media, 99 percentile e massima per Q2 al variare degli esperimenti.

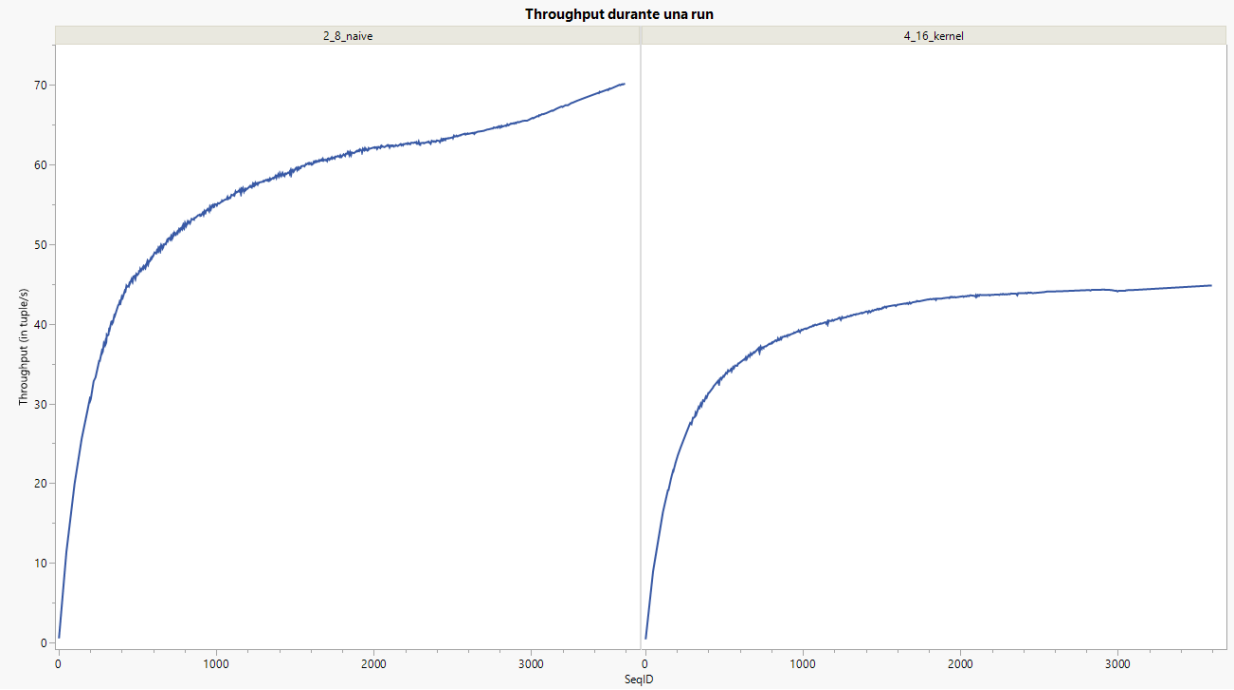

Fig. 6: Throughput durante una run.

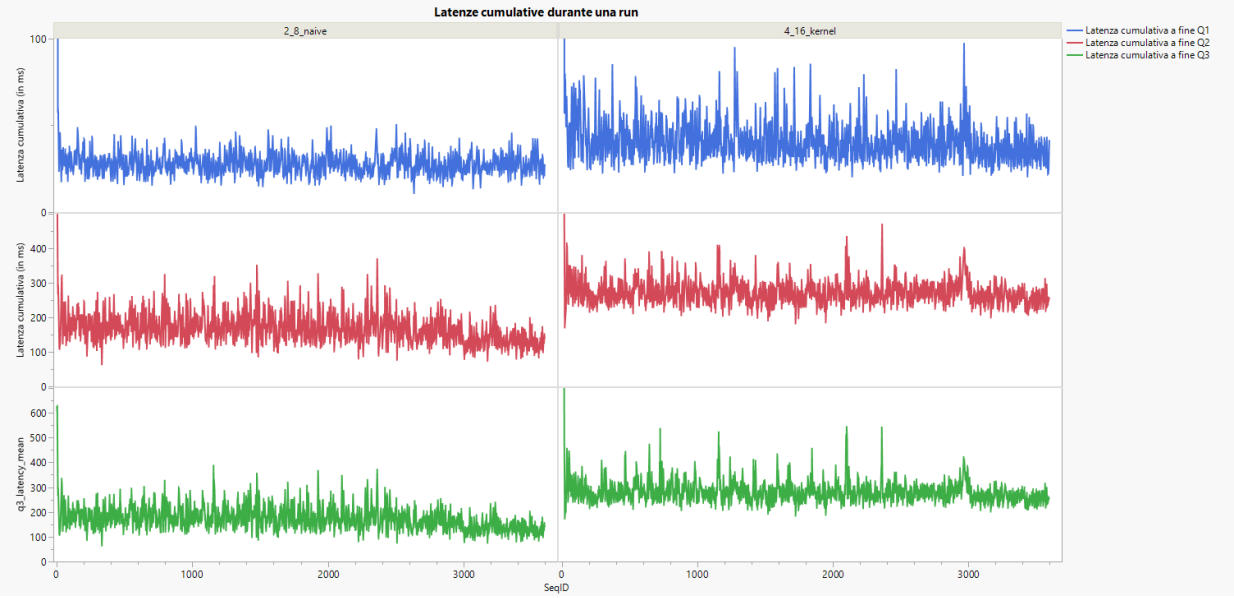

Fig. 7: Latenze cumulative durante una run.