**Immagine che contiene arte, cerchio, disegno, Elementi grafici

Il contenuto generato dall'IA potrebbe non essere corretto.**

Corso di Laurea in Informatica

TESI DI LAUREA

Progettazione di un Nodo FastFlow

per Integrazione di Acceleratori

**Relatore: Candidato:**

**Marco Danelutto Davide De Leonardis**

**Correlatore:**

**Patrizio Dazzi**

**ANNO ACCADEMICO 2024/2025**

**Indice**

[Elenco delle Figure 5](#_Toc214542138)

[Introduzione e Motivazioni 7](#_Toc214542139)

[1.1 La Complessità del Calcolo Eterogeneo 8](#_Toc214542140)

[1.1.1 L'importanza del Kernel Giusto per l'Hardware Giusto 8](#_Toc214542141)

[1.1.2 La Legge di Amdahl e il Costo dell'Accelerazione 9](#_Toc214542142)

[1.2 La Soluzione Proposta: Nodo FastFlow per l'Offloading 10](#_Toc214542143)

[1.3 Obiettivi della Tesi e Contributi 11](#_Toc214542144)

[1.4 Struttura della Tesi 11](#_Toc214542145)

[Fondamenti e Tecnologie Abilitanti 13](#_Toc214542146)

[2.1 Le Architetture Hardware a Confronto 14](#_Toc214542147)

[2.1.1 CPU: l'Orchestratore 14](#_Toc214542148)

[2.1.2 GPU: l'Acceleratore "Forza Bruta" 15](#_Toc214542149)

[2.1.3 FPGA: l’Acceleratore a Pipeline Specializzato 16](#_Toc214542150)

[2.2 Framework di Parallelismo Software 18](#_Toc214542151)

[2.2.1 FastFlow 18](#_Toc214542152)

[2.2.2 OpenMP 19](#_Toc214542153)

[2.3 API per l'Offloading Hardware 20](#_Toc214542154)

[2.3.1 OpenCL 20](#_Toc214542155)

[2.3.2 Apple Metal 22](#_Toc214542156)

[Progettazione dell'Architettura Software 25](#_Toc214542157)

[3.1 Lo "Strategy" Pattern: Separare il "Cosa" dal "Come" 27](#_Toc214542158)

[3.2 Il "Factory" Pattern: Centralizzare la Creazione 28](#_Toc214542159)

[3.3 Architettura della Strategia Acceleratore 29](#_Toc214542160)

[3.3.1 L'"Adapter" Pattern per l'Hardware: IAccelerator 29](#_Toc214542161)

[3.3.2 Il "Composition" Pattern per i Buffer 32](#_Toc214542162)

[3.4 Architettura delle Strategie CPU 33](#_Toc214542163)

[3.5 Il Design del Nodo ff\_node\_acc\_t 35](#_Toc214542164)

[Implementazione del Prototipo 37](#_Toc214542165)

[4.1 Implementazione delle strutture dati comuni 38](#_Toc214542166)

[4.1.1 BlockingQueue: Il motore della sincronizzazione 38](#_Toc214542167)

[4.1.2 Task e StatsCollector: messaggi e risultati 39](#_Toc214542168)

[4.2 Implementazione dei nodi della pipeline di offloading 40](#_Toc214542169)

[4.2.1 Emitter: Il generatore di Task 40](#_Toc214542170)

[4.2.2 ff\_node\_acc\_t: il nodo orchestratore 41](#_Toc214542171)

[4.3 Implementazione degli “Adapter” acceleratori 46](#_Toc214542172)

[4.3.1 Implementazioni basate su OpenCL (GPU e FPGA) 46](#_Toc214542173)

[4.3.2 Implementazione Metal 49](#_Toc214542174)

[4.4 Implementazione delle strategie CPU 52](#_Toc214542175)

[4.5 Implementazione della portabilità 55](#_Toc214542176)

[Analisi Sperimentale e Discussione dei Risultati 57](#_Toc214542177)

[5.1 Setup Sperimentale 57](#_Toc214542178)

[5.1.1 Stack Software e Versioni 58](#_Toc214542179)

[5.2 Metodologia di Benchmark e Metriche 58](#_Toc214542180)

[5.2.1 Definizione delle Metriche 59](#_Toc214542181)

[5.3 Presentazione dei Risultati 61](#_Toc214542182)

[5.4 Discussione e Analisi dei Risultati 63](#_Toc214542183)

[5.4.1 Validazione Architetturale: la Prova della Sovrapposizione 63](#_Toc214542184)

[5.4.2 Identificazione del Collo di Bottiglia: il consumerLoop() 64](#_Toc214542185)

[5.4.3 L'Overhead di Accodamento: Sintomo di Successo 65](#_Toc214542186)

[5.4.4 Analisi del Collo di Bottiglia: Spostamento da "API-Bound" a "Compute-Bound" 67](#_Toc214542187)

[5.4.5 Confronti Finali tra Piattaforme 68](#_Toc214542188)

[Conclusioni 71](#_Toc214542189)

[6.1 Sintesi del Lavoro e Raggiungimento degli Obiettivi 71](#_Toc214542190)

[6.2 Analisi Critica 73](#_Toc214542191)

[6.2.1 Limitazioni del prototipo 73](#_Toc214542192)

[6.2.2 Problemi Riscontrati 74](#_Toc214542193)

[6.3 Considerazioni Personali 77](#_Toc214542194)

[6.4 Sviluppi Futuri 78](#_Toc214542195)

[6.5 Conclusione Finale 78](#_Toc214542196)

[Bibliografia 81](#_Toc214542197)

[Codice 85](#_Toc214542198)

[Ringraziamenti 183](#_Toc214542199)

Elenco delle Figure

[Figura 3.1: Diagramma delle classi dell’architettura 24](#_Toc214538579)

[Figura 4.1: Diagramma di sequenza per il ff\_node\_acc\_t 44](#_Toc214538580)

[Figura 4.2: Diagramma di sequenza per strategie CPU 53](#_Toc214538581)

[Figura 5.1: Risultati Benchmark per N = 10.000 60](#_Toc214538582)

[Figura 5.2: Risultati Benchmark per N = 1.000.000 61](#_Toc214538583)

[Figura 5.3: Risultati Benchmark per N = 7.449.999 61](#_Toc214538584)

[Figura 5.4: Validazione della pipeline (N=1.000.000). 63](#_Toc214538585)

[Figura 5.5: Analisi della composizione della latenza (N=1.000.000). 65](#_Toc214538586)

[Figura 5.6: Spostamento del collo di bottiglia (FPGA, N=1.000.000). 67](#_Toc214538587)

[Figura 5.7: Confronto della scalabilità del throughput 69](#_Toc214538588)

**Capitolo 1**

Introduzione e Motivazioni

L'informatica moderna si trova di fronte a un bivio storico. Per decenni, la Legge di Moore [1] e il Dennard scaling [2] hanno garantito un incremento prestazionale quasi "gratuito", basato sul semplice aumento della frequenza di clock. Oggi, scontrandosi con i limiti fisici della litografia e i vincoli termodinamici, questa strada è divenuta impraticabile. La risposta dell'industria a questa stasi è il calcolo eterogeneo: un paradigma in cui la CPU general-purpose non opera più in isolamento, ma orchestra una costellazione di acceleratori hardware specializzati (GPU, FPGA), ciascuno progettato per eccellere in una specifica forma di parallelismo.

Tuttavia, l'adozione di queste tecnologie introduce una barriera di complessità significativa. Integrare acceleratori in applicazioni C++ esistenti non è un'operazione plug-and-play, ma una sfida architetturale. Ogni dispositivo impone il proprio modello di memoria, le proprie API proprietarie e, soprattutto, latenze di comunicazione che possono facilmente annullare i benefici del calcolo accelerato se non gestite correttamente. La sfida si intensifica quando si tenta di integrare questi dispositivi in framework di stream processing ad alte prestazioni, che impongono vincoli stringenti di non-blocking pur dovendo interagire con API hardware intrinsecamente lente e bloccanti.

Lo scopo di questa tesi è affrontare questa complessità di integrazione. Il lavoro propone la progettazione e l'implementazione di un componente software avanzato capace di astrarre l'eterogeneità dell'hardware sottostante. L'obiettivo è fornire un meccanismo di orchestrazione che gestisca in modo trasparente e asincrono il ciclo di vita dell'offloading (trasferimento dati, esecuzione, sincronizzazione), permettendo allo sviluppatore di sfruttare la potenza di GPU e FPGA senza dover riscrivere la logica di controllo dell'applicazione.

1.1 La Complessità del Calcolo Eterogeneo

La potenza degli acceleratori deriva da architetture profondamente diverse tra loro, progettate per massimizzare forme specifiche di parallelismo. Questa eterogeneità, tuttavia, si traduce in un insieme di complessità che emergono sin dalle prime fasi di integrazione software.

Ogni acceleratore è associato al proprio ecosistema: CUDA domina sulle GPU NVIDIA [4], ROCm [5] e oneAPI [6] coprono rispettivamente AMD e Intel, Metal è la tecnologia di riferimento per Apple [7], mentre le FPGA Xilinx si programmano tramite Vitis HLS [8] o OpenCL [9]. Anche standard aperti come OpenCL, pur offrendo una sintassi comune, nascondono differenze prestazionali e comportamentali tra le varie implementazioni dei vendor. L’architettura proposta in questa tesi (approfondita nel Capitolo 3) mira a risolvere questo problema introducendo un livello di astrazione software che disaccoppia la logica dell’applicazione dai dettagli specifici delle API sottostanti.

Un ostacolo critico è rappresentato dalla gestione della memoria. Le GPU discrete utilizzano una memoria separata accessibile tramite trasferimenti PCIe, mentre sistemi moderni come Apple Silicon o le APU AMD adottano un modello a memoria unificata. Queste differenze influenzano profondamente la latenza del trasferimento dati, la necessità di copie esplicite, la gestione della coerenza e il costo dell’offloading di kernel leggeri. Un’infrastruttura di integrazione efficace deve quindi essere progettata per funzionare correttamente in entrambi gli scenari.

1.1.1 L'importanza del Kernel Giusto per l'Hardware Giusto

Un tema ricorrente in questa tesi sarà la dimostrazione che non esiste un "migliore acceleratore" in assoluto, ma solo scelte ottimali per specifici carichi di lavoro. Come discusso nell'introduzione, la Legge di Amdahl [3] ci insegna che l'accelerazione è limitata dalla frazione seriale del codice, che include l'overhead fisso dell'acceleratore. Ci aspettiamo di confermare che:

* la GPU eccelle con kernel di parallelismo dati massivo (migliaia di operazioni identiche su dati contigui), ma soffre con pattern di accesso irregolari o kernel troppo leggeri (dove l'overhead supera il calcolo);
* l'FPGA è ideale per pipeline di calcolo complesse e specializzate ma inefficiente per semplici operazioni vettoriali. L'implementazione HLS usata in questa tesi—una singola pipeline sequenziale—non è ottimizzata per parallelismo dati e ci aspettiamo che l'FPGA sia la piattaforma più lenta nei test;
* la CPU, pur essendo più lenta in termini di throughput assoluto su kernel massivi, mantiene un vantaggio sulla latenza singola e rappresenta spesso il miglior compromesso per applicazioni reali con carichi misti.

1.1.2 La Legge di Amdahl e il Costo dell'Accelerazione

Per comprendere l'approccio ingegneristico adottato, è fondamentale comprendere i limiti teorici imposti dalla Legge di Amdahl [3]. Questa legge, formulata da Gene Amdahl nel 1967, afferma che lo speedup massimo ottenibile parallelizzando un'applicazione è limitato dalla frazione di codice che rimane intrinsecamente seriale

Nel contesto dell'offloading su acceleratori, la "frazione seriale" include l'overhead fisso introdotto dall'acceleratore stesso: il costo di inizializzazione, l'overhead di sincronizzazione e, in modo cruciale, la latenza di trasferimento dati I/O attraverso i bus.

Se l'applicazione attende passivamente il completamento di un task sull'acceleratore, la latenza dell'I/O diventa un collo di bottiglia insormontabile. L'unica soluzione efficace è l'overlapping (sovrapposizione): orchestrare un flusso continuo in cui il trasferimento dati del task N+1 avviene contemporaneamente al calcolo del task N. Questa tesi dimostra come tale orchestrazione possa essere incapsulata efficacemente all'interno di un nodo software specializzato.

1.2 La Soluzione Proposta: Nodo FastFlow per l'Offloading

Per rispondere a queste sfide, la tesi propone l'utilizzo di FastFlow [10], un framework C++ per il parallelismo strutturato, come infrastruttura di base. Tuttavia, FastFlow nativamente non risolve il problema dell'interazione con hardware eterogeneo bloccante.

La soluzione specifica progettata in questo lavoro è il ff\_node\_acc\_t: un nodo FastFlow specializzato che agisce da "ponte" tra il runtime veloce della CPU e l'hardware accelerato.

In FastFlow, ogni nodo di elaborazione esegue un metodo svc() che deve essere rigorosamente non-bloccante. Se un nodo si blocca in attesa di un evento esterno (come la fine di un calcolo GPU o un trasferimento dati), l'intera pipeline si stalla, degradando le prestazioni globali. Le API degli acceleratori, al contrario, sono spesso bloccanti o richiedono una gestione complessa degli eventi per operare in modo asincrono.

Il ff\_node\_acc\_t risolve questo conflitto incapsulando una pipeline interna asincrona. Invece di eseguire il lavoro direttamente nel metodo svc(), il nodo delega le operazioni a thread dedicati (Producer/Consumer) che gestiscono il ciclo di vita dell'offloading in background. Questo permette al nodo di accettare nuovi task alla massima velocità della CPU, mentre internamente gestisce le latenze dell'hardware e sovrappone il trasferimento dati di un task con l'esecuzione del successivo.

1.3 Obiettivi della Tesi e Contributi

La tesi mira a superare la frammentazione tecnologica e il problema dell'asincronia con una soluzione architetturale unica e riutilizzabile.

**Obiettivo Finale:** progettare, implementare e valutare un nodo FastFlow asincrono (ff\_node\_acc\_t) per l'integrazione efficiente di kernel esistenti per acceleratori hardware (FPGA Alveo, GPU programmata con OpenCL e Metal) all'interno di una pipeline FastFlow.

Questo obiettivo è stato raggiunto e validato attraverso tre contributi chiave:

* architetturale (Capitoli 3 & 4): progettazione e implementazione di un nodo FastFlow asincrono basato su un pattern Producer-Consumer interno. Questo design risolve il problema del metodo svc() bloccante, garantendo la sovrapposizione delle operazioni di I/O con il calcolo e massimizzando il throughput.
* software engineering (Capitoli 3 e 4): l'applicazione rigorosa di design pattern consolidati [11] per costruire un sistema modulare e estendibile, capace di unificare API hardware radicalmente diverse sotto un'unica interfaccia pulita.
* sperimentale (Capitolo 5): un'analisi prestazionale completa che confronta CPU multi-core, GPU e FPGA, validando l'efficacia dell'architettura e identificando i trade-off tra overhead, throughput e tipologia di carico di lavoro.

1.4 Struttura della Tesi

Con queste premesse, la tesi è organizzata come segue, costruendo un percorso logico dalle basi teoriche alla convalida sperimentale:

* Capitolo 2 - Fondamenti e Tecnologie: analisi dei modelli di parallelismo di CPU, GPU, FPGA, dei framework FastFlow e OpenMP e delle API OpenCL e Metal;
* Capitolo 3 - Progettazione dell'Architettura: dettaglio del design software, dei Design Pattern e della logica del nodo ff\_node\_acc\_t;
* Capitolo 4 – Implementazione del Prototipo: presenta i dettagli implementativi delle strutture dati, del nodo orchestratore, degli adapter concreti e della gestione della portabilità cross-platform;
* Capitolo 5 - Analisi Sperimentale: definisce l'ambiente di test, la metodologia di benchmark e analizza i risultati, validando l'architettura e confrontando le piattaforme;
* Capitolo 6 - Conclusioni e Sviluppi Futuri: sintetizza il lavoro, analizza criticamente i problemi riscontrati e traccia le linee guida per i futuri sviluppi.

**Capitolo 2**

Fondamenti e Tecnologie Abilitanti

Questo capitolo fornisce il contesto tecnico necessario per comprendere le scelte architetturali (Capitolo 3) e implementative (Capitolo 4) del progetto. Per costruire un framework di orchestrazione efficace, è essenziale prima comprendere la "natura" fondamentale di ogni componente hardware e software che si intende integrare.

La ricerca della massima efficienza computazionale impone di non affidarsi più a un'unica architettura, ma di selezionare l'hardware, fra quelli disponibili, il cui modello di parallelismo si adatta meglio alla natura specifica del problema.

Questa crescente specializzazione, tuttavia, crea una significativa frammentazione tecnologica. Un'architettura ottimizzata per il data-parallelism (la GPU) ha un'API, un modello di memoria e un paradigma di esecuzione radicalmente diversi da un'architettura ottimizzata per il pipeline (quindi per le FPGA).

Lo scopo di questo capitolo è quindi duplice: fornire una panoramica delle tre architetture hardware utilizzate (CPU, GPU, FPGA), evidenziando i loro distinti modelli di parallelismo che ne giustificano l'utilizzo, e approfondire i framework (FastFlow, OpenMP) e le API di offloading (OpenCL, Metal) utilizzati per programmare questi dispositivi.

Questa analisi servirà da fondamento per giustificare le decisioni di design—come la necessità di un'interfaccia di astrazione per l'hardware e di una pipeline asincrona interna—che verranno introdotte nei capitoli 3 e 4.

2.1 Le Architetture Hardware a Confronto

Il progetto si confronta con tre architetture hardware che rappresentano tre filosofie di calcolo fondamentalmente diverse: CPU (Central Processing Unit), GPU (Graphics Processing Unit), FPGA (Field-Programmable Gate Array).

La scelta di quale architettura utilizzare dipende interamente dalla natura del carico di lavoro da accelerare, come si vedrà nelle misurazioni descritte nel capitolo 5.

2.1.1 CPU: l'Orchestratore

La CPU è l'architettura general purpose che agisce da "cervello" e host per l'intera applicazione.

Le moderne CPU sono sistemi MIMD (Multiple Instruction, Multiple Data) [12], o "task-parallel". Sono composte da un numero relativamente ridotto di core (es. 4-32) altamente sofisticati, ognuno dotato di complesse unità di predizione dei salti, cache gerarchiche e la capacità di eseguire task diversi su dati diversi in modo indipendente e asincrono. I processori moderni includono unità vettoriali (SIMD) per un limitato parallelismo dati. La forza delle CPU moderne risiede nella possibilità di eseguire un singolo compito con la minima latenza possibile.

Proprio per questa sua flessibilità, nel progetto la CPU ricopre due ruoli.

Primariamente, agisce come host: è il dispositivo che esegue il main, gestisce e orchestra la pipeline FastFlow, inviando i task agli acceleratori.

Secondariamente, funge da baseline per le performance, tramite le implementazioni di calcolo parallelo su CPU utilizzate per misurare le prestazioni ottenibili sfruttando unicamente il parallelismo multi-core della CPU.

2.1.2 GPU: l'Acceleratore "Forza Bruta"

Il limite della CPU è il throughput su calcoli semplici e ripetitivi di tipo data-parallel. Per questo, ci si affida alla GPU, un'architettura specializzata ottimizzata per l'alto throughput computazionale, a scapito della latenza per un singolo task.

La GPU è un'architettura data-parallel che implementa il modello SIMT (Single Instruction, Multiple Threads) [13]. È composta da migliaia di core semplici, raggruppati. Quando si lancia un kernel, migliaia di thread vengono lanciati, ognuno con un proprio ID, ed eseguono la stessa istruzione su dati diversi. A differenza della CPU, la GPU opera con un modello host-device. L'host (CPU) orchestra le operazioni, ma il device (GPU) riceve i dati in input e calcola su di essi i kernel.

L’utilizzo della memoria, legato al fatto che la GPU è un device connesso tramite bus PCIe alla CPU, introduce un overhead di comunicazione che è cruciale da gestire. A tal fine, come descritto nel capitolo 3.5, l’architettura del nodo FastFlow è progettata per orchestrare l’offloading della computazione in modo di sovrapporre i trasferimenti relativi ad un task con il calcolo relativo ad un altro task.

La GPU è l'acceleratore "forza bruta" per il parallelismo su dati. È stata scelta per testare la capacità del framework di orchestrare carichi di lavoro massivamente paralleli, dove la stessa operazione viene applicata milioni di volte. Il suo scopo è saturare il bus di memoria e le unità di calcolo con un numero enorme di operazioni identiche.

Le GPU sono programmabili tramite API come OpenCL, la stessa interfaccia standard che abbiamo utilizzato per l'FPGA, permettendo un confronto diretto dell'overhead dell'API su due architetture diverse. La GPU è stata inoltre programmata tramite MacOS Metal per confrontare le performance di uno standard cross-platform (OpenCL) con un'API nativa e a basso overhead.

2.1.3 FPGA: l’Acceleratore a Pipeline Specializzato

L'FPGA è l'architettura più specializzata e concettualmente la più distante dalle altre due. Non è un processore che esegue istruzioni software da una memoria, ma è un circuito integrato composto da un insieme di blocchi logici riconfigurabili (LUTs - Look-Up Tables, Flip-Flops, BRAMs - Block RAM, ecc.).

Tramite un processo di configurazione, questi blocchi logici vengono "configurati" per creare un circuito digitale customizzato. In breve, l'hardware “implementa direttamente” l'algoritmo.

La programmazione di una scheda FPGA richiede di norma la conoscenza di linguaggi di descrizione hardware come Verilog, che richiedono per la programmazione un approccio lento e complesso. Questo progetto, invece, utilizza la Sintesi ad Alto Livello (HLS), fornita dalla toolchain Xilinx Vitis [8]. L'HLS analizza il codice C++ fornito e lo sintetizza in una descrizione hardware (RTL). Questa descrizione viene poi "mappata" sui blocchi logici dell'FPGA per creare un circuito custom, che viene infine salvato in un file bitstream (.xclbin). Questo bitstream è ciò che viene caricato sull'FPGA (nel nostro caso, tramite OpenCL [9]) per configurarlo.

La vera forza dell'FPGA non è il data-parallelism (come la GPU), ma il parallelismo di tipo pipeline. Quando si applica la direttiva #pragma HLS PIPELINE a un loop, il compilatore HLS tenta di trasformare le operazioni all'interno del loop in una sorta di "catena di montaggio" hardware.

Per esempio, se un loop ha 5 stadi che richiedono 5 cicli di clock, un processore tradizionale impiegherebbe 5 \* N cicli per N iterazioni. L'FPGA, invece, trasforma il loop in 5 stadi hardware fisici. Dopo una latenza iniziale per riempire la pipeline (5 cicli), questa può processare 5 dati contemporaneamente (uno per stadio). L'obiettivo è raggiungere un Initiation Interval di 1, ovvero la capacità di iniziare l'elaborazione di un nuovo dato e produrre un risultato finito ad ogni singolo ciclo di clock.

Oltre al pipelining, l'HLS permette di sfruttare il parallelismo dataflow (#pragma HLS DATAFLOW). Questo trasforma diverse funzioni C++ in moduli hardware distinti che operano in parallelo, scambiandosi dati tramite code FIFO (implementate come hls::stream).

L'FPGA è stata inclusa per testare il framework per due paradigmi di accelerazione completamente diversi, uno per FPGA e uno per GPU, che hanno implicazioni diverse sia sull'overhead di comunicazione che sui tipi di kernel ottimali.

È importante notare che, sebbene l'hardware sia radicalmente diverso, l'FPGA (nello specifico, la Xilinx Alveo U50 [14]) è programmata lato host tramite la stessa API OpenCL della GPU. Questo permette un confronto diretto molto interessante, valutando come la stessa API standard si comporti nell'orchestrare due modelli di parallelismo hardware fondamentalmente opposti.

2.2 Framework di Parallelismo Software

Avere a disposizione hardware multi-core non è sufficiente per ottenere un'accelerazione. È necessario uno strato software che permetta di "programmare" il parallelismo. Gestire manualmente i thread è un'operazione complessa, verbosa e sorgente di errori (race condition, deadlock spesso non facili da individuare e risolvere). Per questo motivo, il progetto si affida a framework di parallelismo di più alto livello: FastFlow [10] (usato sia come orchestratore della pipeline dagli acceleratori – GPU e FPGA – che come strategia di calcolo dalla CPU) e OpenMP (usato come baseline per le performance).

2.2.1 FastFlow

FastFlow è un framework C++ per il parallelismo strutturato. Il suo obiettivo è astrarre la complessità della programmazione concorrente, fornendo al programmatore un set di "pattern" paralleli parametrici rispetto alla “business logic” da implementare, riutilizzabili e ad alte prestazioni. Questi pattern incapsulano la logica di sincronizzazione e comunicazione, spesso basandosi su meccanismi efficienti come code lock-free e algoritmi non-bloccanti, per supportare parallelismo a grana fine con latenze di comunicazione core-to-core molto basse.

FastFlow eccelle nella creazione di pipeline complesse tramite il pattern ff\_Pipe (ff/pipeline.hpp). Questo pattern permette di definire una catena di stadi di elaborazione concorrenti (nodi) che comunicano tramite code ad alte prestazioni. Questa è l'astrazione ideale per orchestrare un flusso di offloading verso un acceleratore, dove uno stadio può "emettere" task mentre un altro li "consuma" ed esegue l'offloading. Questo è precisamente il modello che verrà utilizzato per orchestrare la strategia di accelerazione, come descritto nel Capitolo 3.3.

Oltre alle pipeline, FastFlow fornisce pattern di parallelismo dati. Il costrutto ff::parallel\_for (ff/parallel\_for.hpp) [10] è un'astrazione che parallelizza un loop for sui core disponibili, distribuendo le iterazioni tra i thread worker. Questo ha permesso di utilizzare FastFlow anche per implementare una delle strategie di calcolo su CPU, consentendo un confronto diretto delle performance tra il parallelismo dati offerto da FastFlow e quello offerto dallo standard OpenMP.

2.2.2 OpenMP

OpenMP (Open Multi-Processing) [15] è uno standard API per il parallelismo shared-memory. A differenza di FastFlow, che è una libreria C++, OpenMP è tipicamente implementato come un insieme di direttive di compilazione (#pragma) che istruiscono il compilatore su come parallelizzare il codice.

OpenMP utilizza un modello di esecuzione noto come "fork-join". L'esecuzione di un programma OpenMP inizia sempre con un singolo thread, chiamato master thread.

1. Il master thread esegue la parte sequenziale del codice.
2. Quando incontra una direttiva di parallelismo (come #pragma omp parallel for), il master "crea" (o risveglia) un team di thread worker (il fork).
3. L'intera squadra di thread (master + workers) esegue il codice all'interno della regione parallela. Nel caso di un parallel for, le iterazioni del loop vengono distribuite tra i thread.
4. Alla fine della regione parallela, i thread si sincronizzano in una barriera. Una volta che tutti hanno terminato, i thread worker tornano inattivi e solo il master thread prosegue (il join).

OpenMP è lo standard de-facto per il parallelismo dati su CPU. Per questo motivo, è stato utilizzato per implementare una delle strategie di esecuzione del calcolo su CPU (Cpu\_OMP\_Runner). Questa implementazione serve da baseline di performance con cui confrontare l'efficienza del costrutto ff::parallel\_for di FastFlow nello stesso compito.

2.3 API per l'Offloading Hardware

Per permettere al software "host" (in esecuzione sulla CPU) di comunicare con l'hardware device (GPU, FPGA), è necessario utilizzare un'interfaccia di programmazione delle applicazioni su device “esterni”. Queste API astraggono la complessità dell'hardware e forniscono un set di comandi standard per allocare memoria, trasferire dati e avviare calcoli.

Questo progetto utilizza due API distinte, OpenCL e Metal, le cui differenze architetturali sono centrali per l'analisi delle performance.

2.3.1 OpenCL

OpenCL (Open Computing Language) è uno standard aperto e cross-platform gestito dal Khronos Group [9], creato per fornire un'unica API in grado di programmare un'ampia varietà di acceleratori eterogenei, incluse CPU multi-core, GPU di diversi produttori e, in particolare, FPGA (come le Xilinx Alveo).

OpenCL astrae l'hardware attraverso un modello a più livelli ben definito:

1. platform (Piattaforma): rappresenta l'implementazione di un singolo vendor (es. "NVIDIA CUDA", "Intel(R) OpenCL HD Graphics", "Xilinx");
2. device (Dispositivo): è l'acceleratore fisico (es. la GPU o la scheda FPGA);
3. context (Contesto): è l'ambiente di esecuzione che "lega" l'host a uno o più device. È all'interno di un contesto che vengono creati gli oggetti di memoria e i kernel;
4. command queue (Coda di Comandi): è il meccanismo centrale di interazione. L'host (la CPU) accoda comandi (es. tramite clEnqueueWriteBuffer, clEnqueueNDRangeKernel) in questa coda in modo asincrono. Il device esegue i comandi, tipicamente in ordine. Questa natura asincrona è ciò che permette al producerLoop del nostro nodo FastFlow di non bloccarsi;
5. kernel: è la funzione (scritta in C per OpenCL, .cl) che viene eseguita sul device;
6. buffer (cl\_mem): sono gli oggetti che rappresentano la memoria sul device.

OpenCL presuppone un modello a memoria separata. L'host (CPU) ha la sua RAM e il device (GPU/FPGA) ha la sua VRAM/HBM. Qualsiasi dato su cui il kernel deve operare deve essere esplicitamente trasferito dall'host al device tramite un trasferimento DMA, invocato da comandi come clEnqueueWriteBuffer. Allo stesso modo, i risultati devono essere ritrasferiti indietro con clEnqueueReadBuffer.

Questi trasferimenti di memoria, che avvengono tipicamente sul bus PCIe, sono una fonte significativa di latenza e overhead. Diventa quindi fondamentale progettare un orchestratore software (come verrà analizzato nel Capitolo 3) in grado di nascondere questa latenza, sovrapponendo i trasferimenti di un task con il calcolo di un altro. Questa strategia è resa possibile, soprattutto nelle computazioni di tipo stream parallel, grazie all'utilizzo di tecniche note come doppia o tripla bufferizzazione. Tali tecniche permettono di sovrapporre diversi tipi di comunicazioni (come i dati di un task in arrivo e i risultati di un altro in partenza) al calcolo vero e proprio.

Una caratteristica fondamentale di OpenCL è il suo sistema di eventi. Ogni comando accodato può restituire un oggetto cl\_event. Questo evento è un "handle" che rappresenta quel comando specifico all'interno della coda.

Gli eventi sono il meccanismo primario per gestire le dipendenze. Un comando può essere accodato con una lista di eventi da cui dipende: non verrà eseguito finché tutti gli eventi in quella lista non saranno completati. Allo stesso modo, l'host può bloccarsi e attendere un evento specifico o attendere che l'intera coda sia vuota.

OpenCL è stata l'API fondamentale per questo progetto, per una ragione strategica: è l'unica API che ci ha permesso di creare un'interfaccia IAccelerator, come verrà dettagliato nel capitolo 3, in grado di gestire, con la stessa logica, sia la GPU sia l'FPGA. Questo permette un confronto dell'overhead della pipeline di gestione dell’acceleratore su due architetture hardware radicalmente diverse, ma controllate dalla stessa API software.

2.3.2 Apple Metal

Apple Metal [7] è l'API grafica e di calcolo proprietaria di Apple, progettata specificamente per i suoi sistemi (macOS, iOS) e il suo hardware (SoC Apple Silicon).

Metal è stata creata per sostituire OpenCL e OpenGL, che Apple ha ufficialmente deprecato. La sua filosofia progettuale è quella di essere un'API "vicina all'hardware" con un basso overhead. "Overhead" in questo contesto si riferisce al costo (in cicli CPU) necessario per il driver per validare le chiamate API e tradurle in comandi hardware. Metal è progettata per ridurre drasticamente questo costo, permettendo all'applicazione di inviare un numero molto più elevato di comandi al secondo.

A differenza del modello OpenCL a memoria separata, qui la CPU e la GPU condividono lo stesso spazio di memoria fisica (la RAM di sistema) [16].

Quando si crea un buffer (MTLBuffer), la GPU ottiene un accesso diretto alla stessa memoria usata dalla CPU. Questo elimina la necessità di costosi trasferimenti DMA espliciti per l'upload dei dati. Come verrà mostrato nel Capitolo 4.3.2, questo cambia radicalmente l'implementazione del metodo responsabile di inviare i dati all’acceleratore: invece di una chiamata API asincrona (con OpenCL: clEnqueueWriteBuffer), l'operazione diventa un memcpy diretto dalla memoria del Task alla memoria del buffer. La coerenza della cache è gestita dall'hardware.

Anche il modello di sincronizzazione di Metal è diverso da quello di OpenCL.

Invece di una singola coda (command\_queue in OpenCL), Metal usa un sistema a tre livelli. L'unità di lavoro fondamentale è il MTLCommandBuffer. L'host crea un buffer, "codifica" i comandi al suo interno (tramite MTLCommandEncoder), e infine lo "sottomette" ([commandBuffer commit]) alla MTLCommandQueue. La sincronizzazione bloccante più semplice, usata in questo progetto, consiste nell'attendere che l'intero buffer abbia completato la sua esecuzione, tramite la chiamata [commandBuffer waitUntilCompleted].

Per una sincronizzazione più granulare (simile a cl\_event), Metal fornisce gli oggetti MTLEvent e MTLSharedEvent. Questi sono essenzialmente semafori che possono essere segnalati (encodeSignalEvent) o attesi (encodeWaitForEvent) dalla GPU all'interno di un command buffer. Questo permette di creare dipendenze tra command buffer diversi, anche su code diverse, o di sincronizzare la GPU con la CPU. Sebbene non utilizzati nell'implementazione finale di questo progetto (che usa il più semplice waitUntilCompleted), essi rappresentano il meccanismo di sincronizzazione avanzato della piattaforma.

Metal è stata scelta per servire da termine di paragone ad alte prestazioni. Includendo Gpu\_Metal\_Accelerator, è possibile confrontare direttamente, sullo stesso hardware (un Mac M2 Pro), le performance dell'API nativa e ottimizzata (Metal) contro lo standard cross-platform e a più alta astrazione (OpenCL). Questo permette di quantificare il "costo dell'astrazione" di OpenCL e i benefici del modello a memoria unificata [16].

Questa natura ibrida C++/Objective-C, sebbene necessaria per usare Metal, introduce una significativa sfida di interoperabilità per un'applicazione C++ pura. Come verrà analizzato nel Capitolo 4.5, ciò richiede tecniche di bridging e una configurazione di compilazione specifica (CMakeLists.txt) per isolare il codice Objective-C dal resto del progetto.

**Capitolo 3**

Progettazione dell'Architettura Software

Questo capitolo illustra le scelte di progettazione e i pattern architetturali utilizzati per costruire l'applicazione. L'obiettivo primario, oltre al funzionamento corretto del Progetto, è stato quello di creare un sistema flessibile, disaccoppiato e manutenibile, capace di gestire modalità di esecuzione fondamentalmente diverse—il parallelismo dati su CPU e l'offloading asincrono su acceleratori—nascondendo la complessità dietro un'interfaccia pulita.

Come illustrato nella Figura 3.1, raffigurante il diagramma UML delle classi, l'architettura usufruisce dei seguenti Design Pattern: Factory, Strategy, Adapter e Composition, come definiti in letteratura [11].

Immagine che contiene testo, diagramma, Piano, schematico

Il contenuto generato dall'IA potrebbe non essere corretto.

Figura 3.1: Diagramma delle classi dell’architettura

3.1 Lo "Strategy" Pattern: Separare il "Cosa" dal "Come"

Il requisito fondamentale del progetto era supportare modalità di esecuzione (CPU vs. GPU vs. FPGA) di un kernel con paradigmi radicalmente diversi: dal parallelismo dati sincrono su CPU a una pipeline asincrona per l'offloading. Per evitare che la funzione main diventasse un blocco rigido e accoppiato, a ogni implementazione di interfaccia per un device hardware è stato applicato lo Strategy Pattern [11].

Questo pattern ha permesso di definire una famiglia di algoritmi (le "strategie"), incapsularli in oggetti separati e renderli intercambiabili. A livello architetturale, come mostrato nella Figura 3.1, questo si è tradotto nella creazione di un'interfaccia C++ pura, IDeviceRunner, che definisce un unico "contratto": il metodo execute(N, NUM\_TASKS). Ogni modalità di esecuzione è stata incapsulata in una classe "Strategia" concreta (Cpu\_FF\_Runner, Cpu\_OMP\_Runner, AcceleratorPipelineRunner). Il main agisce ora come "Client", operando solo tramite l'interfaccia IDeviceRunner senza conoscerne l'implementazione.

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

3.2 Il "Factory" Pattern: Centralizzare la Creazione

L'uso dello Strategy Pattern ha reso il main agnostico rispetto all'implementazione, ma ha spostato la responsabilità di decidere quale strategia concreta istanziare. Per non demandare questo compito al client, è stato applicato il Factory Pattern [11] per disaccoppiare il client dalla creazione degli oggetti di cui ha bisogno.

La classe DeviceRunner\_Factory centralizza questa logica. Il main ora delega interamente la creazione a questa factory, che istanzia l'oggetto corretto, sia esso un acceleratore come GPU o FPGA o utilizzando la CPU, e restituisce al main un puntatore all'interfaccia IDeviceRunner.

Immagine che contiene testo, schermata

Il contenuto generato dall'IA potrebbe non essere corretto.

Analizzeremo ora in dettaglio le due famiglie di strategie, partendo dalla più complessa, AcceleratorPipelineRunner, per poi descrivere l'architettura più semplice delle strategie CPU.

3.3 Architettura della Strategia Acceleratore

La strategia AcceleratorPipelineRunner è l'implementazione concreta di IDeviceRunner che gestisce l'offloading. Il suo metodo execute(N, NUM\_TASKS)è responsabile di costruire l'intera pipeline FastFlow [10], che consiste principalmente di un nodo Emitter che ha la responsabilità di generare lo stream di Task, e del nodo ff\_node\_acc\_t, che li elabora. Inoltre, AcceleratorPipelineRunner avvia la pipeline, aspettandone il completamento.



All'interno di questa strategia, emerge un secondo livello di astrazione per gestire la complessità dell'hardware, l’adattatore IAccelerator.

È importante sottolineare che la struttura scelta in questo caso — una pipeline lineare Emitter --> ff\_node\_acc\_t — è una delle tante possibili applicazioni, e non l'unica. L’architettura di ff\_node\_acc\_t lo rende indipendente dalla configurazione del programma FastFlow in cui è inserito. Ad esempio, lo stesso ff\_node\_acc\_t potrebbe essere impiegato come worker in un ff\_farm di FastFlow o come stadio intermedio in una pipeline più complessa.

3.3.1 L'"Adapter" Pattern per l'Hardware: IAccelerator

Il nodo ff\_node\_acc\_t deve orchestrare l’intera gestione dell'hardware (GPU o FPGA), ma le API (per esempio quelle considerate in questa tesi, OpenCL e Metal) possono essere incompatibili/diverse in caso di tipi diversi di acceleratori. Per risolvere questa incompatibilità è stato applicato il Pattern Adapter [11].

È stata creata l'interfaccia IAccelerator, che funge da "contratto" generico per un dispositivo di calcolo. Essa definisce metodi astratti come initialize(), send\_data\_to\_device(), execute\_kernel(), ecc.

L'interfaccia IAccelerator agisce quindi da Adapter, adattando le API specifiche dell'hardware (come OpenCL [9] o Metal [7]) a un'interfaccia comune. Allo stesso tempo, funge da Strategy per il ff\_node\_acc\_t, che opera su un puntatore a questa interfaccia senza conoscerne l'implementazione concreta.

Immagine che contiene testo, schermata, software

Il contenuto generato dall'IA potrebbe non essere corretto.

Le classi Gpu\_OpenCL\_Accelerator, Fpga\_Accelerator e Gpu\_Metal\_Accelerator agiscono come Adattatori concreti: implementano IAccelerator e traducono le chiamate generiche nelle chiamate API specifiche della piattaforma.

Una sfida chiave che l'interfaccia deve risolvere è la gestione del modello di memoria del dispositivo. Tecnologie come OpenCL [9] richiedono una gestione esplicita dei buffer sulla memoria del device (es. VRAM). Al contrario, API moderne come Metal [7] su Apple Silicon sfruttano la memoria unificata (Unified Memory) [16], dove CPU e GPU condividono lo stesso spazio di indirizzamento, eliminando la necessità di copie esplicite.

Il design dell'interfaccia tiene conto di questa differenza:

* le implementazioni OpenCL richiederanno una logica condivisa per allocare, gestire e riciclare i buffer sul device (tramite un BufferManager).
* l'implementazione Gpu\_Metal\_Accelerator (l'Adapter per Metal) non utilizzerà questa logica, ma implementerà una propria strategia di buffering interna, basata su memcpy diretti verso la memoria condivisa.

È importante notare che, sebbene Gpu\_OpenCL\_Accelerator e Fpga\_Accelerator utilizzino entrambi OpenCL e condividano quindi molta logica (come la creazione del contest e della coda dei comandi, la gestione dei cl\_mem, ecc.), si è fatta una esplicita scelta di progettazione per non creare una classe base comune (es. AbstractOclAccelerator), che ha portato necessariamente alla duplicazione di un po’ di codice.

L'introduzione di un ulteriore livello di ereditarietà, infatti, avrebbe aggiunto un grado di complessità architetturale ritenuto inutile. L'API Gpu\_Metal\_Accelerator ha un ciclo di vita e una gestione della memoria completamente diversi dagli acceleratori che utilizzano OpenCL. Trattare ogni "Adapter" concreto come un'implementazione completamente indipendente e auto contenuta dell'interfaccia IAccelerator ha portato a un'architettura più semplice da capire e più facile da manutenere. Sebbene questa scelta sacrifichi il principio DRY (Don't Repeat Yourself) — risultando in una voluta duplicazione di codice tra Gpu\_OpenCL\_Accelerator e Fpga\_Accelerator — garantisce un accoppiamento più lasco tra le implementazioni concrete degli acceleratori.

3.3.2 Il "Composition" Pattern per i Buffer

Sebbene la duplicazione della logica di orchestrazione API fosse una scelta voluta, quella della logica di gestione dei buffer non lo era.

Per gli adattatori Gpu\_OpenCL\_Accelerator e Fpga\_Accelerator, che condividono la stessa identica logica di gestione del pool di buffer OpenCL, è stato applicato il pattern Composition [11]. La logica di gestione dei buffer (allocazione, riallocazione e riciclo) è stata estratta e incapsulata in una classe separata, BufferManager. Le due classi acceleratore che sfruttano OpenCL ora hanno un BufferManager come membro e delegano ad esso questo compito specifico.

Per quanto riguarda l’acceleratore che sfrutta il framework Metal, come anticipato nella sezione precedente, la sua gestione della memoria unificata ha reso conveniente implementare un buffer manager specifico (MetalBufferManager), definito direttamente nel file Gpu\_Metal\_Accelerator.mm

Entrambi i buffer manager allocano dinamicamente la dimensione dei buffer, riutilizzandoli per ogni iterazione, in base alla dimensione del task.

Questi manager sono stati configurati con un POOL\_SIZE di 3. Questo valore è stato scelto come compromesso ottimale per la nostra pipeline interna a 2 stadi (producer/consumer). Un pool di 1 buffer forzerebbe un'esecuzione seriale, impedendo qualsiasi sovrapposizione. Un pool di 2 è il minimo teorico per abilitare l'overlapping (permettendo al producerLoop di lavorare sul Task N+1 mentre il consumerLoop è bloccato sul Task N). L'uso di 3 buffer aggiunge un "cuscinetto" che garantisce che lo stadio producerLoop, tipicamente più veloce, possa sempre preparare un task in anticipo. Questo assicura che lo stadio consumerLoop (identificato come il collo di bottiglia del sistema) non resti mai inattivo in attesa di lavoro, massimizzando così il throughput senza allocare una quantità eccessiva di memoria sul dispositivo. Aumentare ulteriormente la dimensione del buffer pool sarebbe stato inutile. Non sarebbe aumentato il throughput in quanto il consumer è sempre al limite, sarebbe servito allocare una quantità di memoria gigantesca (per 1 set di buffer—3 buffer--: 30MB x 3 x POOL\_SIZE. Es. Con POOL\_SIZE = 100, dovrei allocare 9GB di VRAM su FPGA) e sarebbe quindi aumentata la latenza in quanto i task avrebbero passato molto più tempo di coda.

3.4 Architettura delle Strategie CPU

A differenza della complessa strategia di offloading, l'architettura per le strategie CPU (Cpu\_FF\_Runner e Cpu\_OMP\_Runner) è significativamente più semplice. Queste strategie di esecuzione servono come baseline di performance; il loro scopo è permettere un confronto diretto con le prestazioni del parallelismo dati puro su un processore multi-core, consentendo così di quantificare e isolare l'overhead e la latenza introdotti dalla pipeline di offloading.

Per garantire un confronto equo, entrambe le strategie CPU condividono lo stesso algoritmo: un loop esterno sequenziale che itera sui NUM\_TASKS, e all'interno di ogni iterazione esegue un calcolo parallelo (parallelismo dati) su N elementi.

Per centralizzare questa logica ed evitare duplicazioni, è stato utilizzato il Template Method Pattern [11].

È stata creata una classe base astratta, AbstractCpuRunner. Questa classe base implementa il metodo execute() pubblico (il "Template"). Questo metodo definisce l'intero scheletro dell'algoritmo:

Immagine che contiene testo, schermata, software

Il contenuto generato dall'IA potrebbe non essere corretto.

All'interno del loop sequenziale, execute() invoca il metodo virtuale puro execute\_parallel\_loop(0, N), responsabile dell'esecuzione del calcolo parallelo.

Le classi concrete, Cpu\_FF\_Runner e Cpu\_OMP\_Runner, ereditano da AbstractCpuRunner e hanno il solo compito di fornire l'implementazione del metodo virtuale. In particolare, implementano execute\_parallel\_loop() utilizzando la tecnologia di parallelismo specifica (rispettivamente ff::parallel\_for di FastFlow [10] o la direttiva #pragma omp parallel for di OpenMP [15]) per poi invocare il metodo concreto execute\_kernel\_work(i) fornito dalla classe base AbstractCpuRunner, il quale definisce il lavoro effettivo da eseguire su tutti gli N elementi.

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

Es. Esecuzione con CPU che sfrutta il parallel\_for di FF

3.5 Il Design del Nodo ff\_node\_acc\_t

L'applicazione dei pattern visti finora definisce la struttura di alto livello. Il cuore pulsante dell'applicazione, e la componente tecnicamente più critica, è tuttavia l'implementazione del ff\_node\_acc\_t. Reso possibile dall'astrazione IAccelerator, il suo design interno è progettato per risolvere il problema fondamentale dell'integrazione con FastFlow: il metodo svc() di un nodo non deve mai bloccarsi, altrimenti l'intera pipeline va in stallo.

Per orchestrare operazioni asincrone e bloccanti senza bloccare il svc(), il nodo incapsula un pattern Producer-Consumer multi-threaded. Il design del nodo si articola lungo il suo intero ciclo di vita, gestito dai metodi standard del framework FastFlow:

1. inizializzazione (svc\_init()): questo metodo viene chiamato una sola volta dal runtime di FastFlow prima che il nodo processi qualsiasi task. La sua responsabilità è duplice: inizializzare l’acceleratore e avviare la pipeline interna producer-consumer. I due thread, producer e consumer entrano immediatamente in esecuzione e si mettono in attesa passiva sulle rispettive code vuote (inQ\_ e readyQ\_).
2. esecuzione (svc()): questo è il punto di ingresso del nodo, chiamato dal runtime per ogni task da processare. La sua esecuzione deve essere ultra-rapida e non-bloccante.

* Durante l'esecuzione: si limita a ricevere il Task\*, registrarne il timestamp (arrival\_time) e inserirlo nella prima coda interna, inQ\_, sbloccando così il Producer. Ritorna subito GO\_ON.
* All'atto della terminazione: quando svc() riceve il segnale EOS (End-of-Stream) dal nodo precedente, la sua responsabilità è innescare lo spegnimento della pipeline interna, propagando loro un segnale di terminazione.

1. pipeline Interna (Thread Producer e Consumer): questi sono i thread di lavoro che eseguono il pattern Producer-Consumer:

* il Producer (producerLoop) ha la responsabilità di prendere i task dall'ingresso (inQ\_) ed eseguire tutte le operazioni asincrone (upload, avvio kernel) tramite IAccelerator nel modo più veloce possibile. Passa quindi il task "in volo" allo stadio successivo (readyQ\_).
* il Consumer (consumerLoop) ha la responsabilità di prendere i task pronti (readyQ\_) ed eseguire l'unica operazione bloccante (get\_results\_from\_device()), aggiornare l'oggetto StatsCollector con le metriche di performance del task appena completato e finalizzare il task. Questa architettura logica permette la sovrapposizione: mentre lo stadio Consumer è bloccato sul Task N, lo stadio Producer può processare il Task N+1 e il svc() può accettare il Task N+2.

1. terminazione (svc\_end()): questo metodo viene chiamato dal runtime dopo che svc() ha restituito un EOS. La sua unica responsabilità è garantire una chiusura pulita, assicurandosi che entrambi i thread interni siano terminati, che tutte le risorse siano state rilasciate e che nessun task sia rimasto "in volo".

**Capitolo 4**

Implementazione del Prototipo

Il Capitolo 3 ha descritto il "perché" dell'architettura—le scelte di design, i pattern utilizzati e le diverse responsabilità logiche—questo capitolo descrive il "come", ovvero come le scelte di design e i pattern sono stati effettivamente implementati.

L'analisi si focalizzerà sulle sfide tecniche principali: l'implementazione del nodo orchestratore asincrono (ff\_node\_acc\_t), le soluzioni architetturali adottate per astrarre e supportare API di offloading (come OpenCL e Metal) e la gestione della comunicazione in un contesto concorrente.

Infine, verranno mostrati i dettagli implementativi delle strategie CPU (utilizzate come baseline prestazionale) e della portabilità cross-platform.

Per illustrare visivamente le due diverse logiche di esecuzione, in questo capitolo verranno presentati e analizzati due diagrammi di sequenza UML: il primo dedicato al flusso sincrono delle strategie CPU, il secondo alla pipeline asincrona interna del nodo ff\_node\_acc\_t.

4.1 Implementazione delle strutture dati comuni

Alla base del funzionamento dell'intera architettura, e in particolare della strategia di offloading, vi sono le strutture dati definite nella directory src/common/, che sono descritte nelle seguenti sottosezioni.

4.1.1 BlockingQueue: Il motore della sincronizzazione

Il design del ff\_node\_acc\_t si basa su una pipeline interna Producer-Consumer. Il meccanismo che abilita questa architettura è la BlockingQueue, una coda concorrente thread-safe. Questa implementazione è cruciale perché garantisce un'attesa passiva (0% CPU), a differenza di un'attesa attiva (es. spin-lock) che sprecherebbe risorse. Per lo stesso motivo non sono state usate le code SPSC native di FastFlow in quanto sono non-bloccanti; se il consumerLoop cercasse di prendere un task da una coda FastFlow vuota, il thread dovrebbe chiedere di nuovo, migliaia di volte al secondo, consumando il 100% della CPU.

Questo è ottenuto tramite l'uso di std::mutex per la mutua esclusione e std::condition\_variable per la notifica [17]:

* il metodo pop() implementa l'attesa passiva: il thread (Producer o Consumer) che chiama pop() su una coda vuota rilascia automaticamente il mutex e si "addormenta" (cv\_.wait()), finché non viene svegliato da un altro thread;
* il metodo push() inserisce un elemento e "sveglia" (cv\_.notify\_one()) un eventuale thread addormentato in attesa sulla coda.

4.1.2 Task e StatsCollector: messaggi e risultati

La BlockingQueue trasporta un "messaggio". Questo messaggio è la struct Task, una semplice struttura dati che incapsula i puntatori ai dati e i metadati di stato (come arrival\_time) necessari per l'elaborazione e la misurazione.

Immagine che contiene testo, schermata, Carattere, schermo

Il contenuto generato dall'IA potrebbe non essere corretto.

Infine, il consumerLoop (che gira su un thread separato) deve comunicare i risultati delle misurazioni al thread main in modo asincrono, tramite lo StatsCollector.

Questa classe implementa la comunicazione in due modi:

* tramite std::atomic, gli accumulatori delle metriche (es. computed\_ns, total\_InNode\_time\_ns), vengono aggiornati in modo thread-safe dal consumerLoop ad ogni task completato;
* tramite std::promise e std::future [17], il main (che attende sulla future) viene notificato che l'intero stream è terminato e il conteggio finale dei task è pronto.

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

4.2 Implementazione dei nodi della pipeline di offloading

La strategia AcceleratorPipelineRunner implementa una ff\_Pipe a due stadi. Nelle sottosezioni che seguono, analizziamo l'implementazione di entrambi i nodi che la compongono.

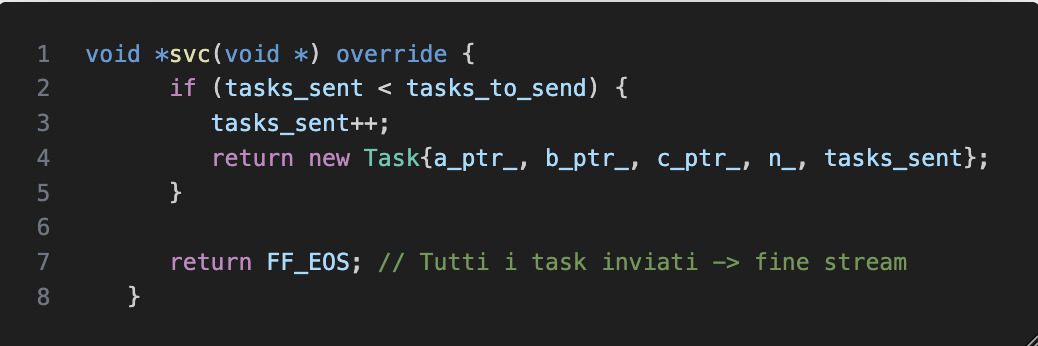
4.2.1 Emitter: Il generatore di Task

Un'applicazione che integra FastFlow [10] opera su uno stream di task. Per alimentare il ff\_node\_acc\_t, è necessario un nodo sorgente che generi questo stream.

In un'applicazione reale, questo sarebbe un qualsiasi stadio precedente (come un ff\_farm o un altro nodo di elaborazione) che produce i dati. Per questo progetto, tale ruolo è svolto da un nodo Emitter, che funge da generatore dello stream.

Si assume, come avverrebbe in uno scenario reale, che i buffer di memoria siano stati allocati dagli stadi che precedono l'offloading. L'Emitter, quindi, si occupa solo di inizializzare questi dati una volta (l'inizializzazione implementata con valori diversi [ a[i] = int(2 \* i) ] serve a evitare che il compilatore ottimizzi eccessivamente il calcolo).

Il suo metodo svc() si limita a creare un nuovo oggetto Task (che incapsula i puntatori ai dati) e a inviarlo, fino al raggiungimento del limite, dopodiché invia il segnale FF\_EOS.



Questo puntatore Task appena creato viene quindi passato dal runtime di FastFlow al metodo svc() del ff\_node\_acc\_t, che ne avvia l'elaborazione asincrona.

4.2.2 ff\_node\_acc\_t: il nodo orchestratore

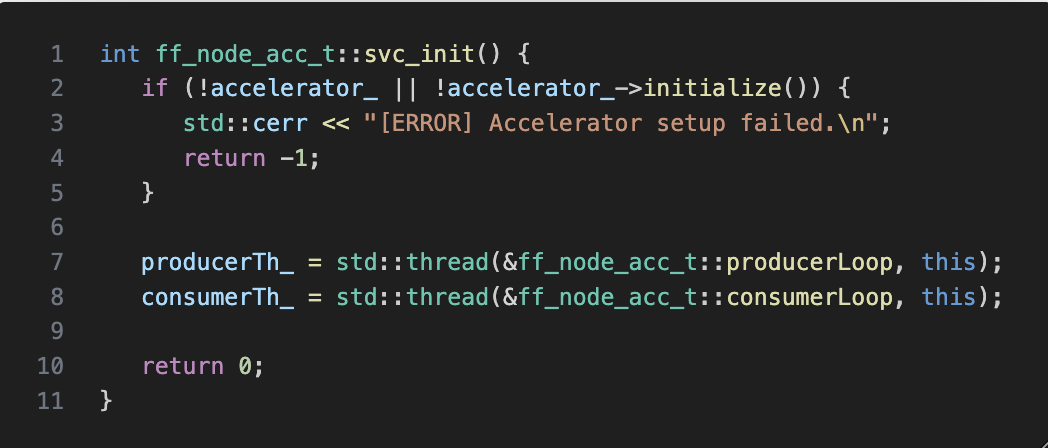
Questo è il componente implementativo asincrono chiave. Questo nodo acceleratore realizza il design logico Producer-Consumer asincrono discusso nel Capitolo 3, orchestrando le strutture dati comuni e i thread. Alla fine di questa sezione, la Figura 4.1 presenta un diagramma di sequenza che mostra le operazioni effettuate dall’ff\_node\_acc\_t.

**Il meccanismo "SENTINEL" per la terminazione**

Per gestire il ciclo di vita dei thread interni (producerLoop e consumerLoop) ed evitare deadlock, l'intera implementazione si basa su un meccanismo a "sentinella". Un messaggio speciale, SENTINEL (un puntatore statico), viene inviato attraverso le stesse BlockingQueue usate per i dati. Quando un thread riceve SENTINEL, sa che non arriveranno più dati e può terminare.

**Inizializzazione e avvio dei thread**

Il metodo svc\_init() viene chiamato una volta sola dal runtime di FastFlow. Oltre a inizializzare l'acceleratore (tramite l'interfaccia IAccelerator), la sua responsabilità principale è creare e avviare i due thread std::thread [17] (producerTh\_ e consumerTh\_) che entrano nei rispettivi loop e si mettono in attesa passiva sulle BlockingQueue (inQ\_ e readyQ\_) fino all’arrivo di un Task.



**Flusso di esecuzione del task**

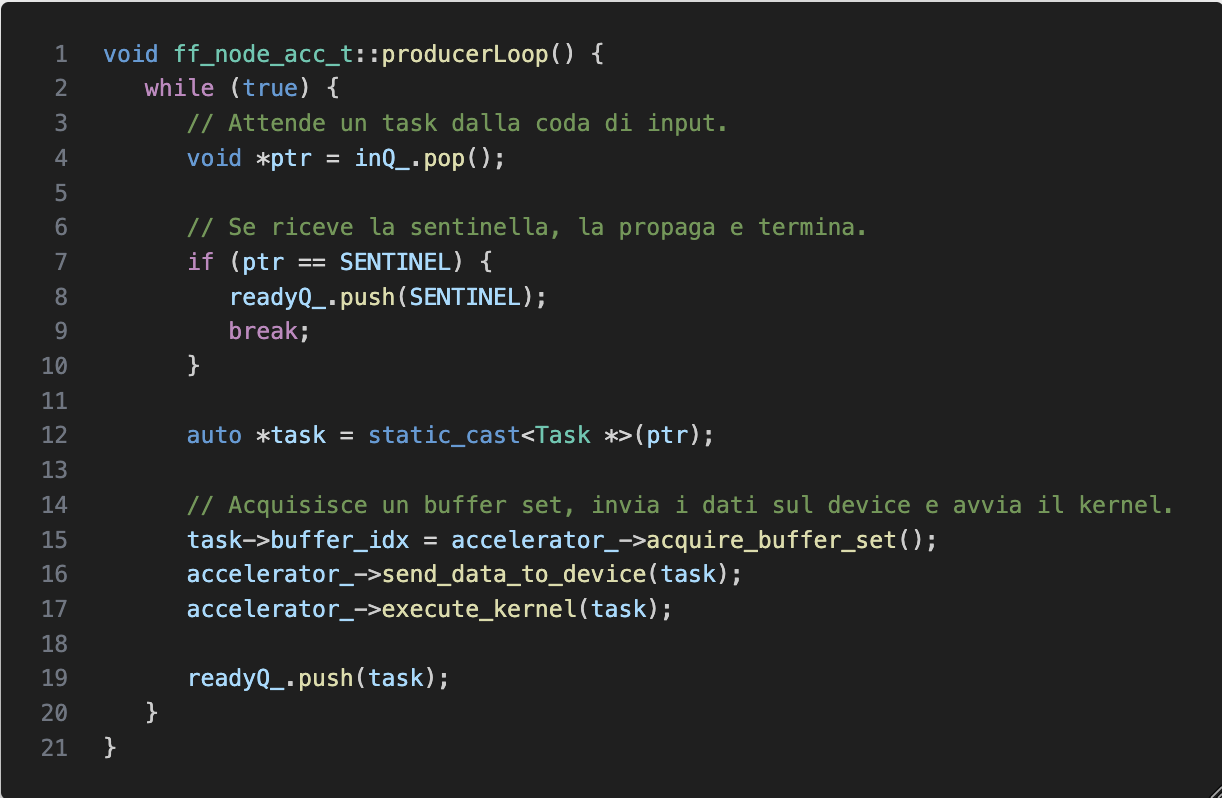
Il funzionamento della pipeline interna è una coreografia tra i tre thread:

* **thread svc()** (non bloccante): quando un Task arriva dall'Emitter, il metodo svc() chiamato da FF lo riceve. Per rimanere non-bloccante, questo metodo imposta il timestamp arrival\_time e inserisce immediatamente il Task nella inQ\_. Questo push trasferisce la responsabilità del Task dal thread di FastFlow al thread producerLoop interno, sbloccando quest'ultimo (che era in attesa su inQ\_.pop()) e permettendo a svc() di ritornare subito FF\_GO\_ON.

Immagine che contiene testo, schermata, software, Software multimediale

Il contenuto generato dall'IA potrebbe non essere corretto.

* **thread producerLoop()** (asincrono): il Producer si sveglia, estrae (pop) il Task dalla inQ\_. Esegue quindi tutte le operazioni veloci e asincrone tramite l'interfaccia IAccelerator: acquisisce un buffer (acquire\_buffer\_set()), avvia l'upload (send\_data\_to\_device()) e lancia il kernel (execute\_kernel()). Il Task è ora "in volo. Il producerLoop ha terminato il suo compito per questo task e passa la responsabilità del Task 'in volo' al consumerLoop inserendolo nella readyQ\_, per poi tornare immediatamente in attesa su inQ\_ per un nuovo task. Se riceve SENTINEL, lo propaga a readyQ\_ per terminare il Consumer.



* **thread consumerLoop()** (bloccante): il Consumer si sveglia ed estrae (pop) il Task dalla readyQ\_. Qui esegue l'unica operazione bloccante dell'intera pipeline: accelerator->get\_results\_from\_device(). Questa chiamata invoca il metodo bloccante definito dall'interfaccia IAccelerator (descritta nel Cap 3.3.1), che nell'implementazione OpenCL (usata da GPU e FPGA) corrisponde a clEnqueueReadBuffer con CL\_TRUE. Questa chiamata attende il completamento del kernel e il download dei dati. Una volta sbloccata, il thread aggiorna lo StatsCollector (usando std::atomic per la thread-safety) e infine distrugge l'oggetto Task.

Immagine che contiene testo, schermata

Il contenuto generato dall'IA potrebbe non essere corretto.

**La sovrapposizione fra calcolo e comunicazione: i "Task In-Flight"**

Questo flusso realizza la sovrapposizione nella pipeline che massimizza il throughput.

Grazie al disaccoppiamento fornito dalle BlockingQueue e ai due thread interni, il nodo può gestire contemporaneamente più "Task in-flight" (in volo), ognuno in uno stadio diverso: mentre il consumerLoop è bloccato in attesa del Task N, il producerLoop sta già processando l'upload e il launch del Task N+1, in modo tale da permettere all’svc() di accettare il Task N+2.

**Terminazione e meccanismo SENTINEL**

La terminazione ordinata e corretta della pipeline interna è gestita dal meccanismo SENTINEL e orchestrata dai metodi svc e svc\_end del nodo acceleratore.

1. Innesco (Thread svc): quando svc() riceve FF\_EOS dall'Emitter (segnalando che non arriveranno più Task), la sua unica responsabilità è inserire il puntatore SENTINEL nella inQ\_ e ritornare FF\_EOS;

2. Propagazione (Thread Producer): il producerLoop, che era in attesa su inQ\_.pop(), si sblocca e riceve il SENTINEL. Riconoscendolo, sa che deve terminare. Prima di farlo, propaga il SENTINEL inserendolo a sua volta nella readyQ\_ (per sbloccare il consumer) e termina il proprio loop;

3. Chiusura (Thread Consumer): il consumerLoop, che era in attesa su readyQ\_.pop(), si sblocca e riceve il SENTINEL. Capisce che tutti i task "in volo" sono stati processati. A questo punto, comunica il conteggio finale dei task al thread main (tramite la std::promise dello StatsCollector) e termina il proprio loop;

4. Join (Thread svc\_end): solo dopo che svc ha ritornato FF\_EOS, il runtime di FastFlow invoca svc\_end() . Questo metodo esegue il join() su producerTh\_ e consumerTh\_. Questa chiamata si sblocca quasi immediatamente, poiché entrambi i thread hanno già terminato la loro esecuzione grazie alla propagazione del SENTINEL, garantendo una chiusura pulita, senza race condition e senza lasciare task incompleti.



Figura 4.1: Diagramma di sequenza per il ff\_node\_acc\_t

4.3 Implementazione degli “Adapter” acceleratori

Come descritto nel Capitolo 3, la strategia AcceleratorPipelineRunner scelta dalla factory DeviceRunner\_Factory, incapsula l'intera logica della pipeline di offloading. Questa strategia, nel suo metodo execute(), crea e passa al costruttore del ff\_node\_acc\_t un puntatore all'interfaccia IAccelerator.

Questa interfaccia è il contratto che disaccoppia il nodo FastFlow dall'hardware.

Come definito nel design del ff\_node\_acc\_t (Cap 3.5), i metodi di questa interfaccia sono invocati dai due thread interni del nodo:

* il Producer (thread producerLoop) esegue le operazioni asincrone e veloci: send\_data\_to\_device() e execute\_kernel();
* il Consumer (thread consumerLoop) esegue l'unica operazione bloccante: get\_results\_from\_device().

L'implementazione di questa interfaccia presenta sfide diverse per ciascuna tecnologia, e questa sezione analizza come le tre classi "Adapter" concrete (Gpu\_OpenCL\_Accelerator, Fpga\_Accelerator e Gpu\_Metal\_Accelerator) ne implementano i metodi, traducendo le chiamate generiche in comandi API specifici.

4.3.1 Implementazioni basate su OpenCL (GPU e FPGA)

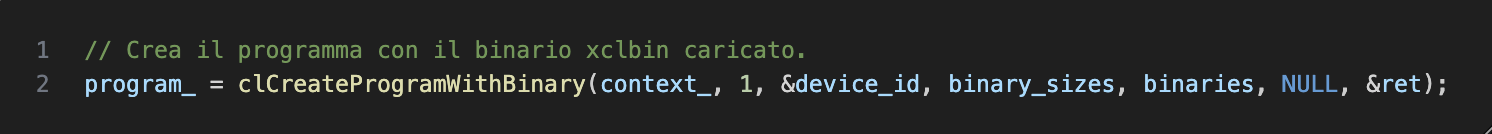
Le classi Gpu\_OpenCL\_Accelerator e Fpga\_Accelerator utilizzano entrambe l'API OpenCL. Il loro flusso di lavoro è simile, ma differisce in modo cruciale nel modo in cui il kernel viene caricato, come evidenziato nel metodo initialize().

Infatti, se entrambe le classi trovano una piattaforma OpenCL, creano il contesto e la coda dei comandi OpenCL [9], e chiamano il costruttore di bufferManager che inizializza il pool di buffer, La GPU esegue una compilazione "Just-in-Time" (JIT) del codice sorgente .cl, leggendolo come stringa e usando clCreateProgramWithSource seguito da clBuildProgram [9].

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

L'FPGA invece, carica un file binario pre-compilato (.xclbin) che contiene il bitstream HLS, usando clCreateProgramWithBinary [24]:



Il **flusso di esecuzione** (chiamato dal ff\_node\_acc\_t) è il seguente:

1. send\_data\_to\_device(): dopo aver acquisito i buffer, riallocandone la memoria a seconda della dimensione del task, l’acceleratore (usando una macro di controllo errori - OCL\_CHCK) avvia un trasferimento DMA asincrono dei dati (non-bloccante) verso la memoria del device, tramite clEnqueueWriteBuffer con blocking\_write = CL\_FALSE. L'evento OpenCL (task->event) viene generato dall'ultima operazione di upload (clEnqueueWriteBuffer). Questo evento viene poi usato da execute\_kernel() come dipendenza esplicita (clEnqueueNDRangeKernel... &previous\_event... &task->event). Ciò garantisce che l'hardware rispetti l'ordine delle operazioni (l'esecuzione non partirà prima che l'upload sia finito), anche se il producerLoop che le accoda ritorna immediatamente.

Immagine che contiene testo, schermata, software, Sistema operativo

Il contenuto generato dall'IA potrebbe non essere corretto.

1. execute\_kernel(): imposta gli argomenti con clSetKernelArg e accoda l'esecuzione del kernel con clEnqueueNDRangeKernel, che ritorna immediatamente.

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

1. get\_results\_from\_device(): è l'unica chiamata bloccante. Esegue clEnqueueReadBuffer con blocking\_read = CL\_TRUE. Questo blocca il thread consumerLoop fino a che il kernel ha terminato la sua esecuzione e il DMA di lettura è completo.

.Immagine che contiene testo, schermata, software, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

4.3.2 Implementazione Metal

L'implementazione Gpu\_Metal\_Accelerator (per MacOS) è radicalmente diversa, a causa dell'interoperabilità C++/Objective-C e del modello a memoria unificata, ed è divisa in tre parti:

1. header C++ puro (.hpp): Gpu\_Metal\_Accelerator.hpp nasconde i tipi Objective-C (es. id<MTLDevice>) dichiarandoli come puntatori void\*;
2. sorgente Objective-C++ (.mm): il file Gpu\_Metal\_Accelerator.mm è compilato come OBJCXX. Qui, i tipi void\* sono riconvertiti ai tipi Metal tramite "bridging cast" (\_\_bridge). Inoltre, Gpu\_Metal\_Accelerator è costretto a disporre della sua specifica classe di gestione buffer;
3. flag di compilazione: CMakeLists.txt imposta il flag -fobjc-arc [7], abilitando l'Automatic Reference Counting che gestisce la memoria degli oggetti Objective-C.

**Flusso di lavoro**

Il modello a Memoria Unificata di Apple Silicon cambia il modo in cui i dati vengono trasferiti, eliminando la necessità di trasferimenti DMA espliciti.

* initialize(): dopo aver creato la coda dei comandi e aver chiamato il costruttore del manager dei buffer, esegue la compilazione JIT del kernel .metal usando [device newLibraryWithSource:...]. Successivamente crea un oggetto che rappresenta il kernel compilato.

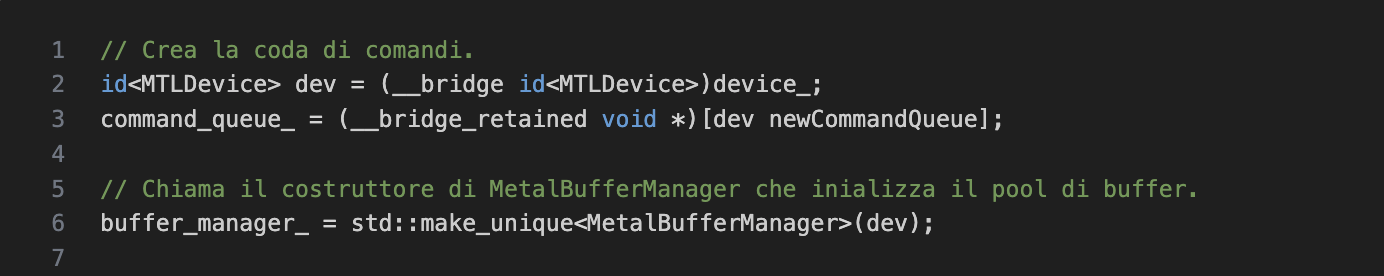
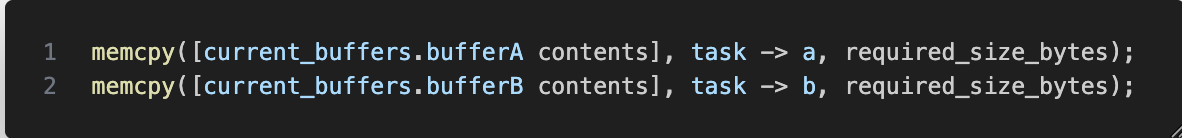


Immagine che contiene testo, schermata, software, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

* send\_data\_to\_device(): l'"upload" non è un'operazione asincrona, ma un memcpy diretto dalla memoria del Task alla memoria del buffer (MTLBuffer), che è già visibile alla GPU grazie alla memoria unificata. Utilizzando MTLResourceStorageModeShared [16], il trasferimento dati non è più una chiamata al driver, ma un memcpy ultra-veloce da una zona all'altra della stessa RAM Questa è l'ottimizzazione hardware più efficiente possibile su un Mac.



* execute\_kernel(): utilizza il paradigma "encoder" di Metal. Un MTLCommandEncoder viene creato per impostare i buffer ([encoder setBuffer:...]) e "dispatchare" i thread ([encoder dispatchThreads:...]). L'esecuzione viene sottomessa in modo asincrono con [commandBuffer commit]

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

* get\_results\_from\_device(): questa è la chiamata bloccante. A differenza di OpenCL, è un processo a due fasi: prima attende il completamento del kernel ([commandBuffer waitUntilCompleted]) e poi esegue un memcpy dalla memoria del buffer di output ([buffers\_c\_ contents]) al puntatore del Task.

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

4.4 Implementazione delle strategie CPU

Le strategie di esecuzione su CPU (Cpu\_FF\_Runner e Cpu\_OMP\_Runner) non rappresentano l'obiettivo finale del progetto, ma servono come baseline di performance. Alla fine di questa sezione, la Figura 4.2 raffigura il diagramma di sequenza per le strategie CPU.

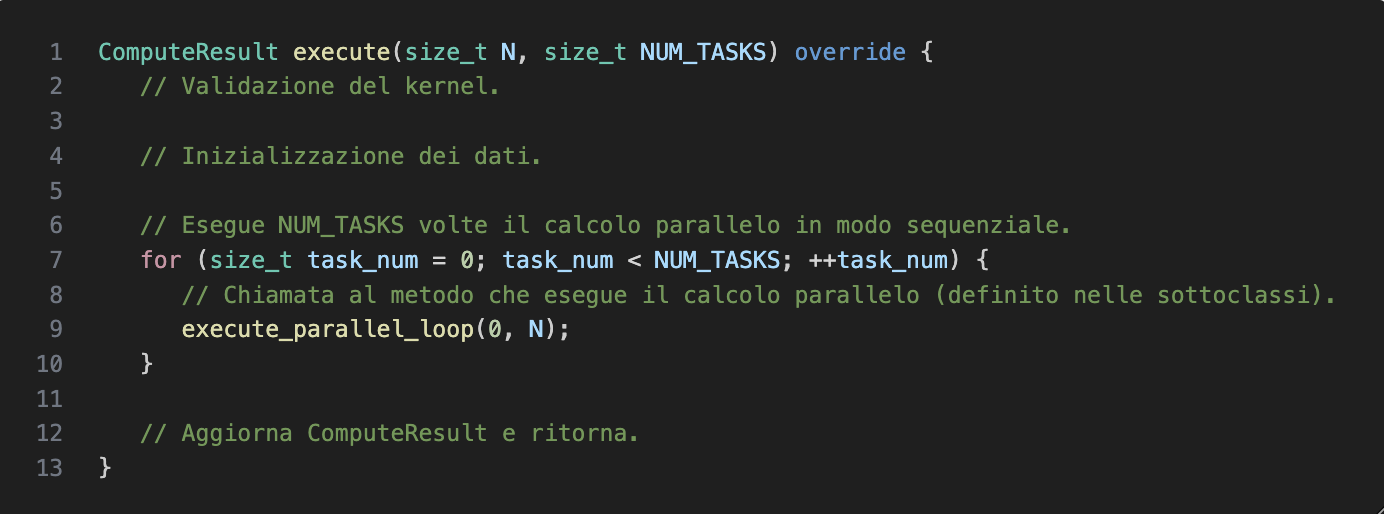
Queste implementazioni sono state sviluppate per due scopi principali:

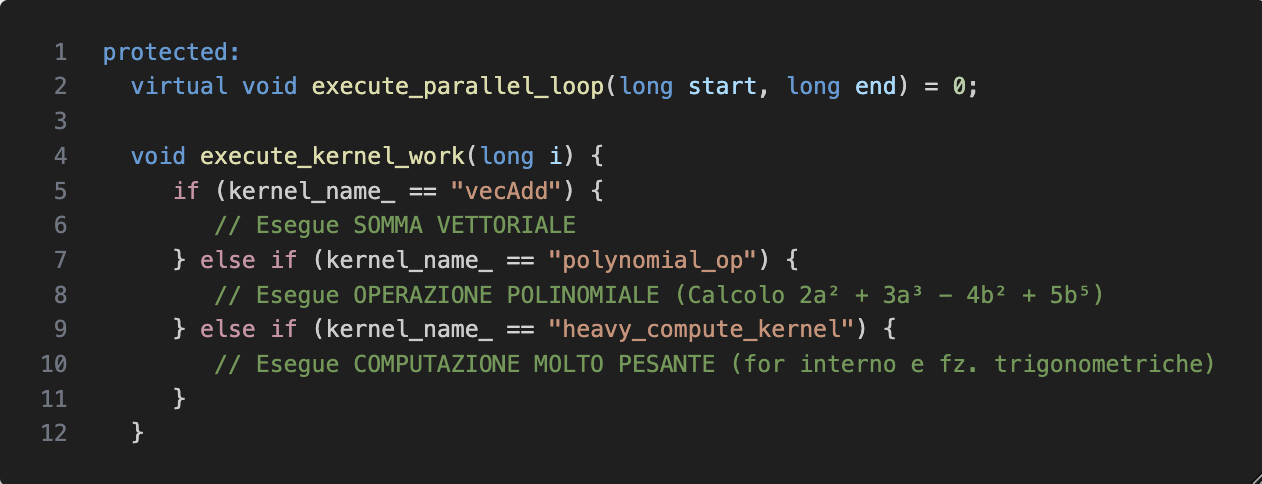
* confronto: forniscono una misura “ideale” delle prestazioni ottenibili eseguendo i kernel direttamente su un processore multi-core moderno, utilizzando parallelismo dati puro;
* quantificazione dell'Overhead: permettono di valutare il *costo* della strategia di offloading. Confrontando l'esecuzione su CPU con quella su acceleratore, specialmente per kernel molto leggeri (low-compute), è possibile isolare e misurare la latenza introdotta dalla pipeline asincrona, ovvero la latenza di calcolo (Avg\_Pure\_Compute\_Time\_ms) e l’overhead totale (Avg\_Overhead\_Time\_ms).

A differenza della complessa pipeline di offloading, l'implementazione delle strategie CPU (Cpu\_FF\_Runner e Cpu\_OMP\_Runner) è molto più diretta. Come descritto nel Capitolo 3, queste classi implementano il Template Method Pattern ereditando da AbstractCpuRunner.

La classe base AbstractCpuRunner fornisce il metodo execute(), che contiene la logica comune (allocazione vettori, loop sequenziale sui task, ecc.).

Le classi concrete (come Cpu\_FF\_Runner e Cpu\_OMP\_Runner) devono solo implementare il metodo astratto execute\_parallel\_loop() ed eseguire execute\_kernel\_work(), per definire il calcolo effettivo (es. polynomial\_op) da eseguire su ogni singolo elemento i.





- Cpu\_FF\_Runner: Implementa execute\_parallel\_loop() utilizzando la parallelizzazione dati di FastFlow, ff::ParallelFor [10], il quale distribuisce le iterazioni del loop sui core disponibili.

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

- Cpu\_OMP\_Runner: Implementa lo stesso metodo utilizzando la direttiva #pragma di OpenMP [15].

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.



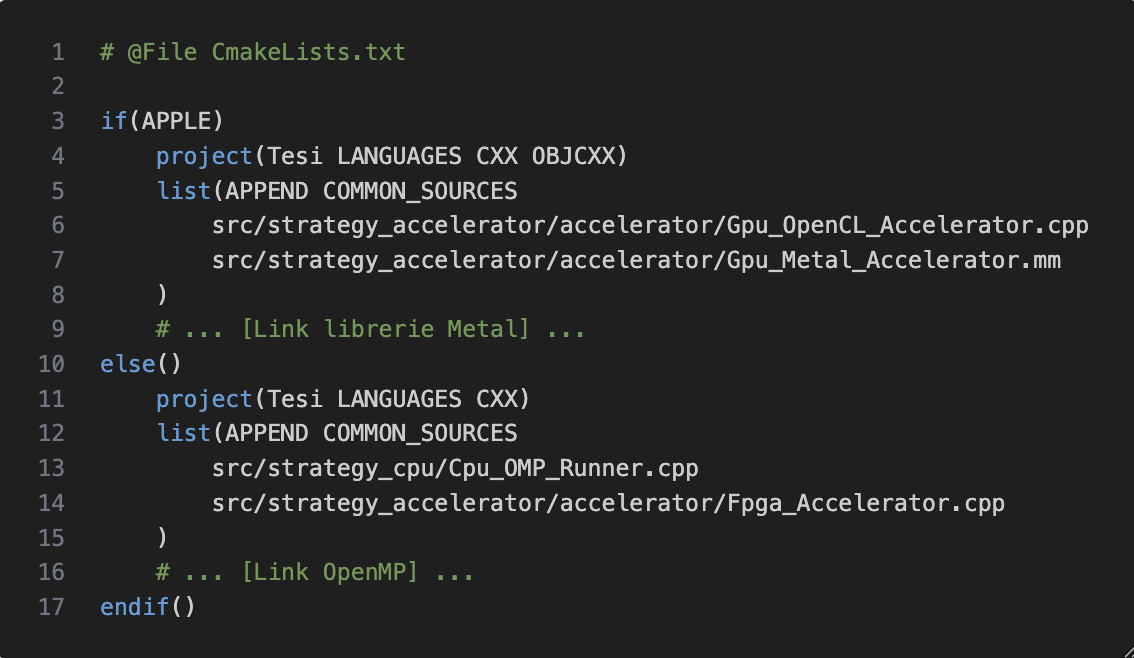
Figura 4.2: Diagramma di sequenza per strategie CPU

4.5 Implementazione della portabilità

Infine, l'intera architettura cross-platform (MacOS e Linux) è gestita dal file CMakeLists.txt [18]. Questo file utilizza direttive condizionali per assemblare l'eseguibile, compilando e linkando i sorgenti corretti in base al sistema operativo su cui viene eseguito.

La logica di selezione si basa sulla variabile APPLE:

* selezione dei Linguaggi e dei Sorgenti: su MacOS, il progetto deve supportare C++ (CXX) e Objective-C++ (OBJCXX). Su Linux, è sufficiente il C++. Questo determina quali file sorgente vengono inclusi nella compilazione, oltre al linking delle rispettive librerie supportate dallo specifico OS:



Come mostrato, la direttiva if(APPLE) gestisce i due target hardware e software:

* su MacOS, vengono compilati gli adapter per l'hardware Apple: Gpu\_Metal\_Accelerator.mm (che richiede Objective-C++) e Gpu\_OpenCL\_Accelerator.cpp;
* su Linux, viene compilato il codice per l'host fisico Ubuntu (dotato di CPU Intel e scheda FPGA su PCIe). Questo include Fpga\_Accelerator.cpp (per l'offloading sull'acceleratore) e Cpu\_OMP\_Runner.cpp (per il confronto prestazionale su CPU Intel).
* gestione dell'Interoperabilità (Objective-C++): per implementare correttamente l'interoperabilità con Metal (descritta nella Sezione 4.3.2), CMakeLists.txt deve istruire il compilatore affinché tratti il file .mm in modo speciale, attivando l'Automatic Reference Counting (ARC):

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

L'impostazione del flag -fobjc-arc è fondamentale, poiché abilita la gestione automatica della memoria per gli oggetti Objective-C (come id<MTLDevice>) istanziati tramite i bridging cast.

**Capitolo 5**

Analisi Sperimentale e Discussione dei Risultati

Questo capitolo definisce l'ambiente sperimentale utilizzato per la valutazione del prototipo, descrive la metodologia di benchmark adottata e analizza e discute i risultati ottenuti. L'obiettivo primario dell'analisi è validare l'efficacia dell'architettura software disaccoppiata, valutare l’efficienza del nodo accelerato e misurare il costo di integrazione (overhead) di diversi acceleratori hardware al variare del carico di lavoro.

5.1 Setup Sperimentale

Le misurazioni sono state eseguite su due host di calcolo distinti:

1. MacBook M2 Pro (ambiente di sviluppo):

* architettura: ARM64;
* CPU: Apple M2 Pro [23], con 10 core;
* GPU: integrata Apple M2 Pro (con 16 unità di calcolo);
* sistema operativo: MacOS Sonoma 15.6.1.

1. Host Linux:

* CPU: Intel Xeon E5-2650 v3 [24] @ 2.30GHz, fornisce 40 thread logici (20 core fisici);
* sistema operativo: Ubuntu 22.04.5 LTS;
* acceleratore: scheda Xilinx Alveo U50 [14] (xilinx\_u50\_gen3x16\_xdma\_base\_5), connessa tramite bus PCIe.

5.1.1 Stack Software e Versioni

La riproducibilità dei risultati è garantita dalle seguenti versioni software:

1. compilatore:

* MacOS: Clang versione 20.1.7 [20] (con Target: x86\_64-apple-darwin24.6.0);
* Linux: GCC 15.1.0 [21].

1. librerie:

* FastFlow: versione 3.0.0 [10] (utilizzata per implementare la pipeline sui nodi e per confronto prestazionale su CPU Apple M2 pro e CPU Intel)
* OpenMP: versione 4.5 [15] (utilizzata per il confronto prestazionale su CPU Linux);
* Toolchain Vitis: versione v2023.1 [8] (utilizzata per la Sintesi ad Alto Livello su FPGA);
* API OpenCL: versione 1.2 [9] (utilizzata per l'interfacciamento con GPU Apple M2 Pro e FPGA);
* Driver XRT: versione 2.16.204 [19] (driver di runtime Xilinx).

5.2 Metodologia di Benchmark e Metriche

Tutti i test sono stati eseguiti con NUM\_TASKS = 100 per garantire la saturazione della pipeline e una misurazione statistica stabile del throughput a regime.

La dimensione del problema (N) è stata fatta variare da 10000 a 7449999 elementi per coprire lo spettro da carichi limitati dall'overhead di memoria (memory-bound) a carichi limitati dal calcolo (compute-bound).

L'analisi comparativa si è concentrata sulle performance di tre kernel eseguiti in modo omogeneo su tutte le piattaforme compatibili (CPU Apple, CPU Intel, GPU OpenCL, GPU Metal, FPGA), con l’obiettivo di analizzare lo spostamento del collo di bottiglia del sistema:

1. **vecAdd** (*vsum*): semplice somma vettoriale [25], elemento per elemento (c[i] = a[i] + b[i]). È utilizzato per misurare l'overhead API e la latenza minima di trasferimento dati che il sistema è in grado di raggiungere. Funge da baseline per il regime memory-bound.
2. **polynomial\_op** (*poly*): un calcolo polinomiale leggermente più complesso di vecAdd (c[i] = 2a2 + 3a3 - 4b2 + 5b5). Polynomial\_op è stato utilizzato per analizzare il costo di un carico di lavoro più realistico e valutare come una maggiore complessità del calcolo, pur mantenendo lo stesso pattern di accesso ai dati, influenzi le performance.
3. **heavy\_compute\_kernel** (*heavy*): calcolo iterativo con funzioni trigonometriche (sin, cos) in un loop di cinque iterazioni sulla coppia di elementi di input (a[i] e b[i]). Questo kernel è stato progettato specificamente per essere compute-bound. Il suo scopo è testare i limiti di potenza di calcolo degli acceleratori e verificare se il tempo di computazione pura riesce finalmente a superare l'overhead di trasferimento dati, ottenendo un significativo speedup.

Questi tre kernel corrispondono alle tre logiche di calcolo implementate sia nel metodo execute\_kernel\_work() della strategia AbstractCpuRunner (Capitolo 4.4) che fornite come file .cl, .metal e .xclbin agli Adapter IAccelerator (Capitolo 4.3).

5.2.1 Definizione delle Metriche

Per valutare l'efficacia dell'architettura producer-consumer implementata nel nodo ff\_node\_acc\_t, è stato adottato un set di metriche che disaccoppia il tempo di calcolo di un task dal suo tempo di attesa in coda:

* **Avg Service Time** (ms): misura il tempo medio tra il completamento di due task consecutivi (tfine, i – tfine, i-1). Il suo valore è dettato dalla velocità dello stadio più lento della pipeline e ne indica il ritmo di produzione a regime.
* **Avg In-Node Time** (ms): per GPU e FPGA, misura il tempo totale che un singolo task trascorre all'interno del nodo acceleratore, dall'ingresso (svc()) all'uscita (consumerLoop()). Questa metrica è calcolata usando il timestamp arrival\_time (impostato in svc() come visto in Cap 4.2.2) e il momento in cui il consumerLoop() finalizza il task. È la latenza percepita dal task, inclusa l'attesa in coda. [Per CPU, invece, *Avg In-Node Time* misura il tempo medio per completare un singolo task in modo sequenziale.]
* **Avg Pure Compute Time** (ms): misura il tempo medio di esecuzione del kernel sull'acceleratore, incluso il tempo di trasferimento dati (download dal device all’host). Isola il costo I/O e calcolo necessario per un task. Misura il tempo impiegato dalla funzione bloccante get\_results\_from\_device() per un task.
* **Avg Overhead Time** (ms): quantifica il costo medio di gestione di un task (Latenza – Calcolo puro). La sua elevata entità è la prova del tempo di attesa in coda accumulato quando la pipeline è satura.
* **Throughput** (tasks/sec): misura la portata complessiva del sistema, calcolata come Task totali eseguiti / Tempo totale elapsed, per fornire un indicatore finale di efficienza.

5.3 Presentazione dei Risultati

In questa sezione vengono presentati i risultati sperimentali ottenuti dai benchmark, la cui metodologia è stata descritta nella sezione precedente. I dati grezzi sono riepilogati nelle figure 5.1, 5.2 e 5.3, che raggruppano le misurazioni per le tre dimensioni del problema analizzate (N=10.000, N=1.000.000, N=7.449.999).

Questi dati costituiscono la base per l'analisi critica e la discussione approfondita che verranno affrontate nella successiva Sezione 5.4.

Figura 5.1: Risultati Benchmark per N = 10.000



Figura 5.2: Risultati Benchmark per N = 1.000.000



Figura 5.3: Risultati Benchmark per N = 7.449.999



5.4 Discussione e Analisi dei Risultati

In questa sezione, i dati sperimentali presentati nella Sezione 5.3 vengono analizzati in profondità. L'obiettivo è interpretare le metriche di performance per validare l'architettura software proposta, identificare i colli di bottiglia del sistema, analizzare le metriche più contro-intuitive e confrontare l'efficienza delle diverse piattaforme hardware e API software.

5.4.1 Validazione Architetturale: la Prova della Sovrapposizione

L'obiettivo primario del design del nodo ff\_node\_acc\_t (progettato nel Capitolo 3.5 e implementato nel Capitolo 4.2.2) era quello di disaccoppiare la ricezione dei task dalla loro finalizzazione, per permettere la sovrapposizione delle operazioni di upload, calcolo e download. I dati confermano in modo inequivocabile il successo di questa architettura.

Come evidenziato nella Figura 5.4, in tutti i test su acceleratore (GPU e FPGA), l'Avg In-Node Time è risultato essere molto superiore all'Avg Service Time.

Ad esempio, nei test su FPGA con N=1,000,000 e kernel polynomial\_op (Tabella 5.2), la latenza media percepita da un singolo task (Avg In-Node Time) è di 399.89 ms, ma il tempo medio tra il completamento di due task (Avg Service Time), dettato dalla velocità dello stadio più lento della pipeline, è di soli 7.04 ms.

Un altro esempio lo si ha nei test su GPU che usa OpenCL, kernel pesante, N=7.4M. L'esecuzione del kernel pesante richiede una latenza totale nel nodo di 310.27 ms. Ciononostante, la pipeline è in grado di produrre un risultato finito ogni 5.66 ms.

Questa discrepanza è la prova diretta che la pipeline interna a due stadi (producer/consumer, implementata nel Capitolo 4.2.2) sta funzionando come previsto. Mentre il consumerLoop è bloccato per quasi 400ms in attesa che un task venga completato e scaricato, il producerLoop() e il thread svc() di FastFlow stanno già accodando e processando i task successivi. È questa sovrapposizione che maschera la latenza. L'alto Avg In-Node Time è reale, ma è il costo pagato da un singolo task in un sistema saturo. L'efficienza della pipeline è dimostrata dal basso Avg Service Time, che permette al sistema di sostenere un throughput molto più elevato di quanto una semplice esecuzione seriale (dove Periodo ≈ Latenza) permetterebbe.

Figura 5.4: Validazione della pipeline (N=1.000.000).

5.4.2 Identificazione del Collo di Bottiglia: il consumerLoop()

Una volta validata l'efficacia della sovrapposizione nella pipeline, il passo successivo è identificare quale stadio ne limita la performance. Nell'architettura a due stadi descritta nel Capitolo 4.2.2, la pipeline è composta da un producerLoop() (asincrono e veloce) e un consumerLoop() (bloccante e lento). In un sistema di questo tipo, il throughput massimo teorico è sempre limitato dalla velocità del suo stadio più lento. La nostra ipotesi è che il consumerLoop(), che gestisce l'operazione bloccante get\_results\_from\_device(), sia questo collo di bottiglia.

Confrontando il tempo di servizio della pipeline con il tempo di lavoro dello stadio bloccante, ad esempio su FPGA con dimensione del problema N=1,000,000, i dati sono chiari:

|  |  |  |
| --- | --- | --- |
| Kernel | Avg Service Time (ms) | Avg Pure Compute Time (ms) |
| vecAdd | 6.96 | 7.09 |
| polynomial\_op | 7.04 | 7.16 |
| heavy\_compute\_kernel | 339.77 | 343.37 |

I dati mostrano un'identità quasi perfetta tra le due metriche in tutti i test. La pipeline produce un risultato finito quasi esattamente con la stessa frequenza del tempo impiegato dal suo stadio finale per completare un'operazione. Questo prova in modo inconfutabile che il collo di bottiglia (lo stadio più lento) che determina la portata massima (Throughput) dell'intero sistema è il consumerLoop(). La pipeline è "Consumer-Bound": non può produrre risultati più velocemente di quanto il consumer (che esegue get\_results\_from\_device()) riesca a finalizzare un task.

5.4.3 L'Overhead di Accodamento: Sintomo di Successo

L'osservazione più contro-intuitiva emersa dai benchmark, visibile nella figura 5.5, è l'entità dell'Avg Overhead Time (Latenza - Calcolo Puro), che in molti casi (es. FPGA, heavy\_compute\_kernel, N=7.4M) raggiunge valori enormi, superando i 129 secondi e rappresentando oltre il 98% della latenza totale del task.

A prima vista, un overhead di questa magnitudine sembrerebbe indicare un'inefficienza catastrofica nel codice, come un'attesa attiva (busy-waiting) o un costo di I/O sproporzionato. Tuttavia, questa interpretazione è errata. L'overhead in questo caso non è la causa del rallentamento, ma un sintomo dell'efficienza della pipeline.

Come stabilito nella Sezione 5.4.2, il collo di bottiglia è il consumerLoop() (misurato da Avg Pure Compute Time). L'architettura del ff\_node\_acc\_t è stata progettata con due thread disaccoppiati da code bloccanti (inQ\_ e readyQ\_, la cui implementazione BlockingQueue è discussa in Cap 4.1.1). Di conseguenza, l'unica posizione in cui un task può accumulare tempo che non sia "calcolo puro" è in attesa all'interno di queste code.

Pertanto, l'elevato Avg Overhead Time misurato è quasi interamente tempo di attesa in coda (*Queueing Delay*).

Questo fenomeno è un sintomo diretto del successo dell'architettura a 2 stadi:

* il producerLoop() (Stadio 1) è molto più veloce del consumerLoop() (Stadio 2, il collo di bottiglia);
* il producer riempie rapidamente le code interne (inQ\_ e readyQ\_) con i 100 task pronti per l’esecuzione;
* i task si accumulano in attesa di essere processati dal consumer, al proprio throughput massimo;
* l'Avg Overhead Time che si misura è la media di questo tempo passato in coda.

Un alto overhead (Avg Overhead Time) è la prova che la pipeline sta funzionando correttamente e sta saturando il suo collo di bottiglia. Il sistema sta spingendo l'hardware al suo limite massimo di throughput, e l’overhead in coda ne è la naturale conseguenza.

Ovviamente è presente anche l’overhead dovuto all'inizializzazione una tantum di OpenCL e delle operazioni di setup necessarie. L’FPGA, in particolare, ha un costo di inizializzazione significativo (elapsed time – computed time nel primo task), dovuto al tempo necessario per caricare il file .xclbin sulla scheda e configurare il circuito. Come per la GPU, questo costo viene ammortizzato solo in scenari a throughput elevato.

Figura 5.5: Analisi della composizione della latenza (N=1.000.000).

5.4.4 Analisi del Collo di Bottiglia: Spostamento da "API-Bound" a "Compute-Bound"

Il confronto tra i kernel ci permette di analizzare la natura del collo di bottiglia.

Con i kernel leggeri, vecAdd e polynomial\_op, i dati sperimentali mostrano che il tempo di calcolo è estremamente basso. Ad esempio, su GPU che usa OpenCL (N=1M), l'Avg Pure Compute Time è di soli 1.12 ms per il kernel vecAdd e 2.05 ms per il kernel polynomial\_op.

In questo scenario, il sistema non è limitato dalla potenza di calcolo della GPU. Il collo di bottiglia è il costo fisso dell'overhead dell'API OpenCL e delle operazioni di I/O.

L'obiettivo del heavy\_compute\_kernel (1000 iterazioni di sin/cos) era di invertire questa relazione, presentando un carico di lavoro dove il calcolo fosse così pesante da superare il costo fisso dell'API. Quando si passa al kernel pesante, l'Avg Pure Compute Time incrementa drasticamente, diventando il fattore dominante: come mostrato nella Figura 5.6, su FPGA, con N=1M, il tempo di calcolo passa da 7.1 ms (polynomial\_op) a 343.3 ms (heavy\_compute\_kernel), un aumento di oltre 48 volte, mentre su GPU che usa Metal il tempo di calcolo passa da 0.29 ms a 0.46 ms. Questa efficienza superiore è una diretta conseguenza del modello a Memoria Unificata (discusso nel Capitolo 2.3.2) e dell'implementazione basata su memcpy, che riducono drasticamente l'overhead per task; il collo di bottiglia non è più l'overhead fisso dell'API o delle operazioni di I/O, ma il sistema è diventato "compute-bound". Il tempo totale è ora dettato dalla reale capacità dell'hardware di eseguire calcoli complessi.

Questa analisi conferma che l'offloading su acceleratore è giustificato solo quando la complessità computazionale del kernel è sufficientemente alta da "assorbire" il costo fisso del trasferimento dati e della latenza del driver.

Figura 5.6: Spostamento del collo di bottiglia (FPGA, N=1.000.000).

5.4.5 Confronti Finali tra Piattaforme

L'analisi finale dei dati rivela i profondi compromessi tra le diverse architetture hardware e i framework software testati. Come mostrato nella Figura 5.7, la metrica principale per il confronto è il Throughput (tasks/sec), che indica l'efficienza complessiva del sistema e si osserva un'inversione di tendenza netta all'aumentare di N:

* CPU (FastFlow vs. OpenMP): Su Linux, per carichi di lavoro molto elevati (N=7.4M), l'ottimizzazione diretta del compilatore di OpenMP supera FastFlow. Per il kernel polynomial\_op (21.4 tasks/sec) e heavy\_compute\_kernel (5.7 tasks/sec), OpenMP si dimostra più efficiente. Questo suggerisce che, per task molto lunghi, l'overhead di runtime dello scheduler più complesso di FastFlow diventa un costo maggiore rispetto alla gestione più snella delle direttive #pragma omp del compilatore.
* GPU (Metal vs. OpenCL): Su MacOS, Metal domina su OpenCL in ogni scenario. La causa principale è l'overhead, che ha due origini come discusso nei Capitoli 2 e 4:
  1. Costo API: Metal è un'API nativa a basso overhead. OpenCL su MacOS è deprecato e richiede uno strato di traduzione (cl2Metal), che introduce una latenza fissa.
  2. Modello di Memoria: Metal beneficia della Memoria Unificata [16], trasformando l'upload dei dati su GPU in un memcpy quasi istantaneo (Cap 4.3.2). OpenCL opera su un modello a memoria separata, imponendo costi di trasferimento e sincronizzazione più alti. I dati lo confermano: nel kernel heavy\_compute\_kernel (N=1M), l'overhead di Metal è di 51.0ms, mentre quello di OpenCL è di 88.9ms. Con un overhead significativamente più basso, Metal ottiene un throughput superiore.
* FPGA: l’FPGA è risultata la piattaforma più lenta, fallendo significativamente con i kernel più complessi. L'errore sarebbe incolpare l'overhead I/O del bus PCIe (Avg Overhead Time = 17370ms). Tuttavia, i dati svelano una causa più profonda: il collo di bottiglia è l’ Avg Pure Compute Time stesso. Ad esempio, nel test heavy\_compute\_kernel (N=1M), la CPU Intel con OpenMP termina un task in 31.35ms. L'FPGA, per lo stesso task, impiega 343.37ms (Avg Pure Compute Time). L'FPGA è quindi 10 volte più lenta della CPU nel calcolo puro.

Questo "fallimento" non è dovuto all'I/O (che è comunque un fattore), ma a un disallineamento architetturale tra problema e soluzione:

* 1. I kernel testati sono problemi di parallelismo dati;
  2. Il kernel heavy\_compute\_kernel pur essendo un dataflow (Cap 2.1.3), implementa una singola pipeline hardware. Processa gli N elementi in modo sequenziale (sebbene pipelinizzato). La GPU esegue N operazioni in parallelo; l'FPGA esegue 1 operazione alla volta (con throughput 1/ciclo). Di conseguenza, il tempo di calcolo dell'FPGA scala linearmente con N, mentre quello della GPU scala molto meno. L'immenso Avg Overhead Time (17.3 secondi) non è altro che il *Queueing Delay* (discusso in 5.4.3), sintomo di un producerLoop() che riempie le code, bloccato da un consumerLoop() molto lento (343ms per task).

In conclusione, l'FPGA non è "lenta", ma è stata usata in modo errato, applicando un'architettura HLS ottimizzata per lo streaming a pipeline a un problema di parallelismo dati di “forza bruta”.

Figura 5.7: Confronto della scalabilità del throughput

**Quest’analisi dimostra che non sempre l'hardware più specializzato è il più veloce e che la natura del problema è fondamentale.** La performance di un design su FPGA dipende enormemente dal livello di ottimizzazione del kernel HLS e per calcoli banali o kernel che implementano una singola pipeline, la potenza di migliaia di core di una GPU o di una CPU moderna è spesso superiore a una singola pipeline HLS non ultra-ottimizzata.

**Capitolo 6**

Conclusioni

Questo capitolo finale traccia una sintesi del lavoro svolto, valutando il raggiungimento degli obiettivi prefissati alla luce dell'analisi sperimentale. Verranno inoltre discusse in modo critico le limitazioni dell'architettura e alcuni problemi affrontati durante lo sviluppo del prototipo, prima di concludere con una riflessione personale sul progetto e sugli eventuali sviluppi futuri.

6.1 Sintesi del Lavoro e Raggiungimento degli Obiettivi

L'obiettivo centrale di questa tesi era la progettazione e l'implementazione di un nodo FastFlow [10] asincrono e riutilizzabile (ff\_node\_acc\_t). Lo scopo di questo nodo è orchestrare l'offloading di computazioni pesanti (kernel) su acceleratori hardware (come FPGA [14] e GPU), gestendone il flusso in modo efficiente e flessibile all'interno di un'applicazione C++ parallela.

Come analizzato nel Capitolo 2, la sfida di progettazione era duplice:

* problema esterno (astrazione): la logica di orchestrazione del nodo doveva essere agnostica all'hardware, in modo da poter gestire la sequenza di operazioni (upload, esecuzione, download) senza conoscere i dettagli delle API specifiche (es. OpenCL, Metal).
* problema interno (asincronia): il nodo doveva rispettare il paradigma non-bloccante di FastFlow. Il suo metodo svc() non poteva bloccarsi in attesa dell'I/O dell'acceleratore (un'operazione ad alta latenza), altrimenti avrebbe stallato l'intera pipeline FastFlow.

Per risolvere queste sfide, il ff\_node\_acc\_t è stato progettato (Capitolo 3) e implementato (Capitolo 4) con un'architettura disaccoppiata:

* soluzione al problema esterno: il nodo orchestra le operazioni non direttamente sull'hardware, ma su un'interfaccia Adapter (IAccelerator, Cap 3.3.1) che gli viene fornita al momento della costruzione. Questa interfaccia astrae l'esecuzione hardware (l'upload dei dati, l'avvio del kernel, il download dei risultati), permettendo di "iniettare" qualsiasi logica di esecuzione hardware (OpenCL, Metal, FPGA) senza modificare il codice di orchestrazione del nodo.
* soluzione al problema interno: il nodo è stato implementato come una pipeline-nella-pipeline (Cap 3.5, 4.2.2). Il thread svc() (non-bloccante) delega il lavoro a due thread interni (producerLoop e consumerLoop) che gestiscono il flusso asincrono e l'unica operazione bloccante (get\_results\_from\_device()) utilizzando BlockingQueue in attesa passiva.

L'analisi sperimentale (Capitolo 5) ha pienamente validato l'efficacia di questo design. L'Avg Service Time (tempo medio tra il completamento di due task consecutivi) è risultato essere di ordini di grandezza inferiore all'Avg In-Node Time (latenza totale del task). Questo prova che la sovrapposizione è efficace e che il design asincrono Producer-Consumer ha saturato con successo l'acceleratore, nascondendone la latenza.

Inoltre, come dimostrato nella Sezione 5.4.2, l'Avg Service Time è risultato quasi identico all'Avg Pure Compute Time (tempo medio di esecuzione del kernel sull'acceleratore).

Questo conferma che il sistema è “consumer-bound”: il software di orchestrazione (ff\_node\_acc\_t) è più veloce dell'esecuzione hardware.

6.2 Analisi Critica

Questa sezione discute onestamente i compromessi di design e le sfide affrontate, analizzando prima le limitazioni architetturali del prototipo e poi i problemi pratici riscontrati durante lo sviluppo, che hanno portato a specifiche scelte implementative.

6.2.1 Limitazioni del prototipo

Il design attuale, pur essendo efficace e robusto per gli obiettivi prefissati, presenta tre compromessi architetturali principali, scelti per privilegiare la semplicità e la chiarezza del prototipo:

1. segnatura fissa dei kernel: l'architettura è attualmente specializzata per kernel con segnatura (2 input, 1 output). Questa decisione è cablata sia nella struct Task sia nel BufferManager. Supportare kernel con segnature diverse (es. 2-in, 2-out) richiederebbe una generalizzazione di queste strutture. Se si volesse implementare un kernel che esegue una moltiplicazione matriciale, non andrebbero però toccati i file ff\_node\_acc\_t, l’interfaccia IAccelerator e la funzione main, ma solo i file che definiscono i tipi dei dati e gli acceleratori concreti che li utilizzano.
2. convenzione rigida sul nome del kernel: il prototipo richiede che il nome della funzione kernel (es. vecAdd) coincida con il nome del file (es. vecAdd.cl). Questa convenzione è utilizzata dall'helper extractKernelName() [src/helpers/Helpers.cpp] per semplicità, ma limita la flessibilità, impedendo (ad esempio) di caricare un kernel vecAdd da un file my\_utils.cl.
3. limite allocazione memoria FPGA: come visto nei benchmark (Cap. 5), la dimensione N su FPGA è limitata a 7.449.999. Questo non è un limite hardware, ma una restrizione pratica incontrata sull'host Linux: il metodo clCreateBuffer (invocato dal BufferManager) fallisce con "Operation not permitted" per allocazioni singole superiori a circa 30MB. Si ipotizza sia una quota di sicurezza a livello utente o di driver XRT sull'host.

6.2.2 Problemi Riscontrati

Durante lo sviluppo, sono emersi diversi problemi significativi, alcuni dei quali presentati in questa sezione, che hanno guidato l'evoluzione dell'architettura verso la sua implementazione finale, la quale ha comportato diverse sfide sia a livello di design logico che di ambiente.

**1. L'Evoluzione del producerLoop da Sincrono ad Asincrono**

Il design originale del ff\_node\_acc\_t era sincrono: il producerLoop invocava un unico metodo accelerator->execute() che eseguiva l'intero ciclo di upload, calcolo e download. Questo bloccava l'intero nodo, annullando i benefici della pipeline.

Soluzione: la prima ri-progettazione ha smantellato la chiamata monolitica execute(), suddividendola in metodi separati (send\_data\_to\_device, execute\_kernel, get\_results\_from\_device) e chiamati dai due diversi thread interni (producerLoop e consumerLoop) dal nodo acceleratore, trasformando di fatto il nodo in una pipeline asincrona.

Tuttavia, la prima versione di questa pipeline usava l'attesa attiva (thread::yield()), sprecando il 100% di un core CPU solo per attendere i task. Questo design inefficiente è stato scartato e sostituito dalla BlockingQueue (Cap 4.1.1), che implementa un'attesa passiva (0% CPU) tramite std::condition\_variable.

**2. Il Fallimento della Pipeline a 3 Stadi (il "nastro trasportatore rotto")**

Durante lo sviluppo, è stata testata un'architettura FastFlow a tre stadi: Emitter --> ff\_node\_acc\_t --> Collector.

L'idea era delegare il conteggio finale a un nodo Collector dedicato. Questa architettura ha fallito in modo catastrofico, presentando una contraddizione logica:

* il Calcolo AVVENIVA: le metriche di debug provavano che il nodo acceleratore riceveva i task, il suo consumerLoop li elaborava e li inviava con ff\_send\_out();
* i Risultati NON ARRIVAVANO: l'output finale era “Tasks processed: 0 / N”, dimostrando che il Collector non riceveva nulla.

La diagnosi era che i task si "perdevano" sul canale FastFlow tra il nodo acceleratore e il Collector. La causa era una race condition di terminazione: ff\_node\_acc\_t è un nodo "attivo" (emette dati da un thread interno) e inviava dati quasi simultaneamente all'arrivo dell'FF\_EOS dall'Emitter. Il framework avviava la terminazione prima di aver garantito la consegna dei task in transito.

Invece di tentare di "riparare" questo comportamento instabile del framework, si è scelta la soluzione architetturale di eliminare il Collector. Le sue responsabilità (conteggio, pulizia memoria, notifica std::promise) sono state "promosse" e integrate direttamente nel consumerLoop del ff\_node\_acc\_t, portando alla robusta pipeline finale a due stadi.

**3. Le Sfide della Concorrenza nel Design Finale**

Anche l'architettura finale a due stadi ha presentato un complesso problema di concorrenza. Dopo aver processato tutti i task, il programma ha iniziato a bloccarsi sull'istruzione count\_future.get(). Questo significava che il consumerLoop (l'unico che poteva impostare la promise) non stava terminando correttamente.

La causa era un deadlock: il main attendeva la promise, ma il consumerLoop non poteva impostarla perché attendeva il SENTINEL, che a sua volta non veniva inviato perché svc\_end() (che chiamava il join() sui thread interni al nodo) veniva chiamato prima che svc() ricevesse FF\_EOS.

La soluzione è stata la coreografia di terminazione (Cap 4.2.2): svc() deve rilevare FF\_EOS, inviare SENTINEL a inQ\_, e solo dopo ritornare FF\_EOS al framework. Il metodo svc\_end() esegue solo il join().

**4. Problemi di Piattaforma e Toolchain Esterni**

Infine, sono stati superati numerosi problemi esterni non legati alla logica del nodo, ma all'ambiente di sviluppo.

* Incompatibilità Driver: un'incompatibilità di versione tra la piattaforma Xilinx (2022.x) e i tool Vitis (2023.x) ha reso inutilizzabile l'emulazione hardware (hw\_emu), costringendo a passare direttamente dai test in software emulation all'hardware fisico.
* Errore di Sintesi HLS: il processo di sintesi del heavy\_compute\_kernel per FPGA non terminava, superando le 15 ore. L'errore logico era un #pragma HLS PIPELINE posizionato sul loop esterno (for i...), che istruiva Vitis a creare un hardware in grado di iniziare un'intera iterazione di N elementi ogni singolo ciclo di clock. Per fare questo, ha tentato di srotolare completamente il ciclo interno (for j...), creando un circuito di 200 unità sin/cos in parallelo che l'FPGA non poteva fisicamente gestire.
* Conflitto compilatori e librerie su Host Linux: è stato creato un file custom per avviare il setup dell’ambiente Xilinx Vitis con le versioni corrette di compilatori e librerie da usare sull’host.

6.3 Considerazioni Personali

Il progetto è stato un’esperienza formativa completa, che mi ha permesso di combinare aspetti di programmazione concorrente, integrazione con hardware eterogeneo e ingegneria del software. La parte più impegnativa non è stata scrivere codice, ma gestire il comportamento asincrono del sistema, comprenderne le interazioni interne, capire approfonditamente i pregi e i difetti dei vari driver diversi, oltre all’implementazione del nodo acceleratore, il quale tutto ha richiesto tempo, cura e continui cicli di refactoring.

Le soddisfazioni sono arrivate soprattutto nella fase finale, quando i risultati sperimentali hanno confermato le ipotesi di progetto. La differenza tra Avg Service Time e Avg In-Node Time ha validato il comportamento Producer–Consumer previsto, mentre gli elevati tempi di Overhead hanno dimostrato la saturazione del sistema, e non un malfunzionamento.

Infine, l’architettura basata su Strategy e Adapter ha mostrato concretamente la sua forza: acceleratori completamente diversi sono stati integrati sotto un’unica interfaccia pulita e coerente.

Questo lavoro ha tratto beneficio diretto dalla formazione acquisita durante la laurea triennale.

* I concetti di parallelismo, sistemi hardware e gerarchie di memoria studiati nei corsi di Architetture degli Elaboratori e Sistemi Operativi del Prof. Marco Danelutto sono stati fondamentali per comprendere il comportamento dei vari device, del nodo acceleratore e per interpretare i dati dei benchmark.
* Allo stesso tempo, i principi di progettazione, modularità e astrazione trattati nel corso di Ingegneria del Software della Prof.ssa Laura Semini hanno guidato la struttura del sistema: l’uso rigoroso dei design pattern ha reso il codice estendibile e manutenibile, pur affrontando problemi complessi di concorrenza e I/O.

Questo progetto è stato quindi non solo un lavoro tecnico, ma una vera occasione di sintesi dei diversi aspetti della mia formazione.

6.4 Sviluppi Futuri

Il prototipo realizzato può evolvere in varie direzioni:

* kernel FPGA realmente parallelo: sviluppare un kernel HLS con più Compute Units in parallelo permetterebbe di valutare l’FPGA in condizioni più vicine alle sue potenzialità reali;
* integrazione in topologie complesse: l'obiettivo sarebbe integrare il nodo acceleratore come worker all'interno di una ff\_farm.
* generalizzazione della segnatura del kernel: rimuovere la limitazione dei kernel a 2 input e 1 output (Sezione 6.2.1), generalizzando la gestione dei buffer e degli argomenti kernel tramite l'uso di vettori dinamici anziché puntatori hardcoded.

6.5 Conclusione Finale

Questo lavoro ha dimostrato con successo la progettazione e l'implementazione di un'architettura software robusta e portabile, centrata sul nodo ff\_node\_acc\_t. L'efficacia dei design pattern utilizzati ha permesso di astrarre la complessità di hardware eterogeneo (FPGA e GPU) e di integrarlo in un framework parallelo di alto livello come FastFlow.

L'analisi sperimentale (Capitolo 5) ha convalidato la soluzione: il design asincrono del ff\_node\_acc\_t ha superato la sfida della latenza I/O (dimostrando l'overlapping) e ha identificato il limite fisico dell'esecuzione hardware come unico collo di bottiglia del sistema.

Questa analisi finale porta a una lezione fondamentale sul calcolo eterogeneo: l'efficienza del sistema dipende dalla corretta scelta architetturale per lo specifico kernel. Le CPU e le GPU sono macchine potentissime e altamente ottimizzate per il parallelismo dati e l'accesso lineare alla memoria. La vera forza dell'FPGA, al contrario, non è la forza bruta sui task semplici, ma la capacità di implementare pipeline di calcolo complesse e customizzate che non esistono su una CPU/GPU. Per un task semplice come una somma vettoriale, l'overhead di comunicazione con l'FPGA e la sua architettura di memoria generica non riescono a competere con la CPU/GPU.

*In definitiva, usare un'FPGA per una somma vettoriale è come usare un bisturi chirurgico specializzato per aprire una scatola di cartone: funziona, ma un semplice taglierino (la CPU/GPU) è più efficiente per questo specifico task.*

Bibliografia

[1] G. E. Moore, "Cramming more components onto integrated circuits," Electronics, vol. 38, no. 8, pp. 114–117, 1965.

[2] R. H. Dennard et al., "Design of ion-implanted MOSFET's with very small physical dimensions," IEEE Journal of Solid-State Circuits, vol. 9, no. 5, pp. 256–268, 1974.

[3] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in AFIPS Conference Proceedings, vol. 30, pp. 483–485, 1967.

[4] NVIDIA Corporation, "CUDA Toolkit Documentation," [Online]. Available: https://docs.nvidia.com/cuda/.

[5] AMD, "ROCm Information & Documentation," [Online]. Available: https://rocm.docs.amd.com/.

[6] Intel Corporation, "oneAPI Specification," [Online]. Available: https://www.oneapi.io/spec/.

[7] Apple Inc., "Metal Programming Guide," [Online]. Available: https://developer.apple.com/metal/.

[8] AMD/Xilinx, "Vitis High-Level Synthesis User Guide (UG1399)," [Online]. Available: https://docs.xilinx.com/.

[9] The Khronos Group, "OpenCL Specification," [Online]. Available: <https://www.khronos.org/opencl/>.

[10] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Torquati, "FastFlow: high-performance stream parallel programming on multi-cores," in Programming with Actors, Springer, 2017, pp. 195–235. Code available at: https://github.com/fastflow/fastflow.

[11] D. C. Kung, Software Engineering, 2nd ed., New York, NY, USA: McGraw-Hill Education, 2014.

[12] S. L. Harris and D. M. Harris, Sistemi digitali e architettura dei calcolatori. Progettare con tecnologia ARM, ed. italiana a cura di N. Scarabottolo, Bologna: Zanichelli, 2017.

[13] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro, vol. 28, no. 2, pp. 39–55, 2008.

[14] Xilinx Inc., "Alveo U50 Data Center Accelerator Card Data Sheet (DS965)," [Online]. Available: https://www.xilinx.com/support/documentation/data\_sheets/ds965-alveo-u50.pdf.

[15] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 4.5," Nov. 2015. [Online]. Available: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[16] Apple Inc., "Apple Silicon CPU Optimization Guide," [Online]. Available: <https://developer.apple.com/documentation/apple-silicon/tuning-your-code-s-performance-for-apple-silicon>.

[17] ISO/IEC, "ISO International Standard ISO/IEC 14882:2020(E) – Programming Language C++," 2020.

[18] Kitware Inc., "CMake Reference Documentation," [Online]. Available: <https://cmake.org/documentation/>.

[19] AMD/Xilinx, "Xilinx Runtime (XRT) Architecture," [Online]. Available: <https://xilinx.github.io/XRT/>.

[20] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO'04), 2004.

[21] Free Software Foundation, "GCC 15.1 Manual," [Online]. Available: <https://gcc.gnu.org/onlinedocs/>.

[22] Apple Inc., "MacBook Pro (14-inch, 2023) - Technical Specifications," [Online]. Available: <https://support.apple.com/kb/SP889>.

[23] Intel Corporation, "Intel Xeon Processor E5-2600 v3 Product Family Data Sheet," [Online]. Available: <https://www.intel.com/content/www/us/en/products/docs/processors/xeon/xeon-e5-v3-datasheet-vol-1.html>.

[24] AMD, "Vitis Introduction and Getting Started," [Online]. Available: https://docs.amd.com/r/en-US/Vitis-Tutorials-Getting-Started/Vitis-Introduction-and-Getting-Started.

[25] Eunomia, "OpenCL Vector Addition Tutorial," [Online]. Available: https://eunomia.dev/en/others/cuda-tutorial/15-opencl-vector-addition/.

Codice

// src/main.cpp

/\*\*

\* @brief Questo file orchestra l'intera applicazione, non conosce i dettagli

\* implementativi dei vari tipi di calcolo.

\*

\* Utilizza due pattern di design principali:

\* 1. Factory Pattern: Per delegare la creazione dell'oggetto corretto IDeviceRunner

\* tramite la factory DeviceRunner\_Factory in base al tipo di dispositivo scelto (CPU,

\* GPU, FPGA).

\* 2. Strategy Pattern: Per eseguire il calcolo senza sapere quale implementazione

\* concreta (CPU, Accelerator) sia stata scelta.

\*/

#include "common/ComputeResult.hpp"

#include "common/IDeviceRunner.hpp"

#include "factory/DeviceRunner\_Factory.hpp"

#include "helpers/Helpers.hpp"

#include <iostream>

#include <memory>

#include <string>

int main(int argc, char \*argv[]) {

// Parametri inseriti da command line.

size\_t N, NUM\_TASKS;

std::string device\_type, kernel\_path, kernel\_name;

parse\_args(argc, argv, N, NUM\_TASKS, device\_type, kernel\_path, kernel\_name);

print\_configuration(N, NUM\_TASKS, device\_type, kernel\_path, kernel\_name);

ComputeResult results;

try {

// Delega alla Factory la creazione della strategia di esecuzione corretta

// (tramite Cpu\_OMP\_Runner, AcceleratorPipelineRunner, ecc.) in base al device\_type.

std::unique\_ptr<IDeviceRunner> strategy =

create\_runner\_for\_device(device\_type, kernel\_path, kernel\_name);

// Esecuzione della computazione tramite la Strategy scelta che esegue la

// parallelizzazione dei task su CPU multicore o tramite la pipeline con offloading

// su GPU/FPGA.

results = strategy->execute(N, NUM\_TASKS);

} catch (const std::invalid\_argument &e) {

std::cerr << "[ERROR] " << e.what() << "\n";

print\_usage(argv[0]);

exit(EXIT\_FAILURE);

}

PerformanceData metrics = calculate\_metrics(results);

print\_metrics(N, NUM\_TASKS, device\_type, kernel\_name, metrics,

results.tasks\_completed);

return 0;

}

// src/common/IDeviceRunner.hpp

#pragma once

#include "ComputeResult.hpp"

#include <cstddef>

/\*\*

\* @brief Interfaccia per una strategia di esecuzione.

\*

\* Definisce un'unica operazione, 'execute', che il main può chiamare, non sapendo se sta

\* eseguendo su CPU o su acceleratore.

\*/

class IDeviceRunner {

public:

virtual ~IDeviceRunner() = default;

/\*\*

\* @brief Esegue l'intero carico di lavoro. É implementata in modo differente dai

\* diversi runner (CPU, GPU con OpenCL, GPU con Metal, ecc.).

\*

\* @param N La dimensione dei dati per ogni singolo task.

\* @param NUM\_TASKS Il numero totale di task da eseguire.

\* @return ComputeResult Una struct contenente tutte le metriche.

\*/

virtual ComputeResult execute(size\_t N, size\_t NUM\_TASKS) = 0;

};

// src/factory/DeviceRunner\_Factory.hpp

#pragma once

#include "../common/IDeviceRunner.hpp"

#include <memory>

#include <string>

/\*\*

\* @brief Funzione Factory che crea e restituisce la strategia di esecuzione

\* appropriata, ovvero un'istanza concreta di IDeviceRunner (es. Cpu\_OMP\_Runner)

\* in base al tipo di device specificato.

\*

\* @param device\_type Il tipo di device (es. "cpu\_omp", "gpu\_opencl").

\* @param kernel\_path Il percorso al file del kernel (per GPU/FPGA).

\* @param kernel\_name Il nome della funzione kernel (per CPU e GPU/FPGA).

\* @return Un puntatore all'interfaccia IDeviceRunner o nullptr se

\* il device\_type non è valido.

\*/

std::unique\_ptr<IDeviceRunner> create\_runner\_for\_device(const std::string &device\_type,

const std::string &kernel\_path,

const std::string &kernel\_name);

// factory/DeviceRunner\_Factory.cpp

/\*\*

\* Implementazione della Factory per la creazione di uno specifico DeviceRunner fra

\* CPU FastFlow, CPU OpenMP, GPU OpenCL, GPU Metal, FPGA.

\*/

#include "DeviceRunner\_Factory.hpp"

#include "../common/device\_types.h"

#include "../strategy\_accelerator/AcceleratorPipelineRunner.hpp"

#include "../strategy\_cpu/Cpu\_FF\_Runner.hpp"

#ifdef \_\_APPLE\_\_

#include "../strategy\_accelerator/accelerator/Gpu\_Metal\_Accelerator.hpp"

#include "../strategy\_accelerator/accelerator/Gpu\_OpenCL\_Accelerator.hpp"

#else

#include "../strategy\_accelerator/accelerator/Fpga\_Accelerator.hpp"

#include "../strategy\_cpu/Cpu\_OMP\_Runner.hpp"

#endif

std::unique\_ptr<IDeviceRunner> create\_runner\_for\_device(const std::string &device\_type,

const std::string &kernel\_path,

const std::string &kernel\_name) {

if (device\_type == device::CPU\_FF) {

return std::make\_unique<Cpu\_FF\_Runner>(kernel\_name);

}

#ifdef \_\_APPLE\_\_

else if (device\_type == device::GPU\_CL) {

auto accelerator = std::make\_unique<Gpu\_OpenCL\_Accelerator>(kernel\_path, kernel\_name);

return std::make\_unique<AcceleratorPipelineRunner>(std::move(accelerator));

}

else if (device\_type == device::GPU\_MTL) {

auto accelerator = std::make\_unique<Gpu\_Metal\_Accelerator>(kernel\_path, kernel\_name);

return std::make\_unique<AcceleratorPipelineRunner>(std::move(accelerator));

}

#else

else if (device\_type == device::CPU\_OMP) {

return std::make\_unique<Cpu\_OMP\_Runner>(kernel\_name);

}

else if (device\_type == device::FPGA) {

auto accelerator = std::make\_unique<Fpga\_Accelerator>(kernel\_path, kernel\_name);

return std::make\_unique<AcceleratorPipelineRunner>(std::move(accelerator));

}

#endif

throw std::invalid\_argument("Invalid device type '" + device\_type + "' for this OS.");

}

// src/strategy\_accelerator/accelerator/IAccelerator.hpp

#pragma once

#include "../../common/Task.hpp"

/\*\*

\* @brief Interfaccia per un acceleratore hardware (es. GPU, FPGA).

\*

\* Definisce le funzioni per ottenere e rilasciare i buffer, e le 3 funzioni

\* principali che implementano i due thread della pipeline interna:

\* - Thread Producer (stadi 1 e 2): send\_data\_to\_device() e execute\_kernel().

\* - Thread Consumer (stadio 3): get\_results\_from\_device().

\*/

class IAccelerator {

public:

virtual ~IAccelerator() = default;

// Esegue tutte le operazioni di setup una tantum.

// (es. trovare il device, creare il contesto OpenCL, compilare il kernel).

virtual bool initialize() = 0;

/\*\*

\* @brief Stadio 1 - Upload: Invia i dati di input dall'host al device.

\* @param task\_context Puntatore a un oggetto Task che contiene i dati e lo

\* stato (incluso l'indice del buffer da usare).

\*/

virtual void send\_data\_to\_device(void \*task\_context) = 0;

/\*\*

\* @brief Stadio 2 - Execute: Accoda l'esecuzione del kernel sul device.

\* Non attende il completamento del kernel.

\* @param task\_context Puntatore a un oggetto Task che contiene lo stato,

\* incluso l'evento di dipendenza.

\*/

virtual void execute\_kernel(void \*task\_context) = 0;

/\*\*

\* @brief Stadio 3 - Download: Attende il completamento di tutte le

\* operazioni precedenti per un task e recupera i risultati dal device

\* all'host. Questa è l'unica funzione bloccante della pipeline. Si

\* sincronizza con il completamento del kernel e accoda il trasferimento

\* dei dati di output all'host.

\* @param task\_context Puntatore a un oggetto Task.

\* @param computed\_ns Tempo di calcolo effettivo.

\*/

virtual void get\_results\_from\_device(void \*task\_context, long long &computed\_ns) = 0;

/\*\*

\* @brief Acquisisce un set di buffer libero dal pool del device.

\* @return L'indice del set di buffer acquisito.

\*/

virtual size\_t acquire\_buffer\_set() = 0;

/\*\*

\* @brief Rilascia un set di buffer nel pool del device.

\* @param index L'indice del set da rilasciare.

\*/

virtual void release\_buffer\_set(size\_t index) = 0;

};

// src/strategy\_accelerator/AcceleratorPipelineRunner.hpp

#pragma once

#include "../common/IDeviceRunner.hpp"

#include "./accelerator/IAccelerator.hpp"

#include <memory>

/\*\*

\* @brief Strategia concreta che implementa IDeviceRunner (è anche un Adapter).

\*

\* Questa classe "adatta" la logica della pipeline FastFlow + Acceleratore per farla

\* apparire come una semplice strategia eseguibile dal main.

\*

\*/

class AcceleratorPipelineRunner : public IDeviceRunner {

public:

/\*\*

\* @brief Costruttore che prende possesso dell'acceleratore hardware da usare.

\*/

explicit AcceleratorPipelineRunner(std::unique\_ptr<IAccelerator> accelerator);

virtual ~AcceleratorPipelineRunner() = default;

/\*\*

\* @brief Orchestra l'intera pipeline FastFlow per l'offloading su un acceleratore.

\* Crea i due nodi della pipeline FF (Emitter, ff\_node\_acc\_t). Riceve l'acceleratore

\* già inizializzato. Avvia la pipeline. Raccoglie e misura i tempi di esecuzione e il

\* numero di task completati.

\*/

ComputeResult execute(size\_t N, size\_t NUM\_TASKS) override;

private:

std::unique\_ptr<IAccelerator> accelerator\_;

};

// src/strategy\_accelerator/AcceleratorPipelineRunner.cpp

#include "AcceleratorPipelineRunner.hpp"

#include "../../include/ff\_includes.hpp"

#include "../common/StatsCollector.hpp"

#include "../ff\_Pipe\_nodes/Emitter.hpp"

#include "../ff\_Pipe\_nodes/ff\_node\_acc\_t.hpp"

#include <chrono>

#include <future>

#include <iostream>

/\*\*

\* @brief Costruttore. Prende possesso del puntatore all'acceleratore.

\*/

AcceleratorPipelineRunner::AcceleratorPipelineRunner(

std::unique\_ptr<IAccelerator> accelerator)

: accelerator\_(std::move(accelerator)) {}

/\*\*

\* @brief Orchestra l'intera pipeline FastFlow per l'offloading su un acceleratore. Crea i

\* due nodi della pipeline FF (Emitter, ff\_node\_acc\_t). Riceve l'acceleratore già

\* inizializzato. Avvia la pipeline. Raccoglie e misura i tempi di esecuzione e il numero

\* di task completati.

\*/

ComputeResult AcceleratorPipelineRunner::execute(size\_t N, size\_t NUM\_TASKS) {

// Dati per ottenere il conteggio finale dei task processati.

StatsCollector stats;

std::future<size\_t> count\_future = stats.count\_promise.get\_future();

// Creazione della pipeline FF e dei suoi due nodi (Emitter, ff\_node\_acc\_t),

// il cui secondo nodo incapsula una pipeline interna a 2 thread (producer, consumer).

Emitter emitter(N, NUM\_TASKS);

ff\_node\_acc\_t accNode(accelerator\_.get(), &stats);

ff\_Pipe<> pipe(&emitter, &accNode);

std::cout << "[Main] Starting FF pipeline execution...\n";

auto t0 = std::chrono::steady\_clock::now();

// Avvio della pipeline e attesa del completamento.

if (pipe.run\_and\_wait\_end() < 0) {

std::cerr << "[ERROR] Main: Pipeline execution failed.\n";

exit(EXIT\_FAILURE);

}

auto t1 = std::chrono::steady\_clock::now();

std::cout << "[Main] FF Pipeline execution finished.\n";

// Raccolta dei risultati.

ComputeResult res;

res.tasks\_completed = count\_future.get();

res.elapsed\_ns = std::chrono::duration\_cast<std::chrono::nanoseconds>(t1 - t0).count();

res.computed\_ns = stats.computed\_ns.load();

res.total\_InNode\_time\_ns = stats.total\_InNode\_time\_ns.load();

res.inter\_completion\_time\_ns = stats.inter\_completion\_time\_ns.load();

return res;

}

// src/ff\_pipes\_nodes/Emitter.hpp

#pragma once

#include "../../include/ff\_includes.hpp"

#include "../common/Task.hpp"

/\*\*

\* @brief Nodo sorgente della pipeline FastFlow.

\*

\* Il nodo Emitter genera i Task da far processare al nodo ff\_node\_acc\_t.

\* Inizializza i dati di input una sola volta, poi crea dinamicamente un nuovo

\* oggetto Task per ogni richiesta dalla pipeline.

\*/

class Emitter : public ff\_node {

public:

/\*\*

\* @param n Dimensione dei vettori da processare.

\* @param num\_tasks Il numero totale di task da generare.

\*/

explicit Emitter(size\_t n, size\_t num\_tasks)

: tasks\_to\_send(num\_tasks), tasks\_sent(0) {

// Init dei vettori con i dati di input.

a.resize(n);

b.resize(n);

c.resize(n);

// ? Usiamo 2 vettori con dati diversi cosi un compilatore estremamente intelligente

// non ? bara e non trasforma la somma in una moltiplicazione (2 \* a[i]).

for (size\_t i = 0; i < n; ++i) {

a[i] = int(i);

b[i] = int(2 \* i);

}

a\_ptr\_ = a.data();

b\_ptr\_ = b.data();

c\_ptr\_ = c.data();

n\_ = n;

}

/\*\*

\* @brief Genera un nuovo Task fino al raggiungimento del numero totale.

\* @return Un puntatore a un nuovo Task, o FF\_EOS al termine.

\*/

void \*svc(void \*) override {

if (tasks\_sent < tasks\_to\_send) {

tasks\_sent++;

return new Task{a\_ptr\_, b\_ptr\_, c\_ptr\_, n\_, tasks\_sent};

}

return FF\_EOS; // Tutti i task inviati -> fine stream

}

private:

size\_t tasks\_to\_send; // Numero totale di task da inviare

size\_t tasks\_sent; // Numero di task già inviati

std::vector<int> a, b, c; // Vettori di input/output

int \*a\_ptr\_, \*b\_ptr\_, \*c\_ptr\_; // Puntatori ai dati di input/output

size\_t n\_; // Dimensione dei vettori

};

// src/ff\_pipes\_nodes/ff\_node\_acc\_t.hpp

#pragma once

#include "../../include/ff\_includes.hpp"

#include "../common/BlockingQueue.hpp"

#include "../common/StatsCollector.hpp"

#include "../common/Task.hpp"

#include "../strategy\_accelerator/accelerator/IAccelerator.hpp"

#include <atomic>

#include <future>

#include <iostream>

#include <memory>

#include <thread>

/\*\*

\* @brief Nodo FastFlow che orchestra l'offloading su un acceleratore.

\*

\* Implementa una pipeline interna a 2 stadi gestita da 2 thread:

\* 1. Producer (Upload+Launch): Trasferisce i dati dall'host al device e avvia

\* l'esecuzione del kernel.

\* 2. Consumer (Download): Trasferisce i risultati dal device all'host.

\*

\* Permette di sovrapporre le operazioni di I/O con il calcolo, nella pipeline il task 'n'

\* è in esecuzione, mentre i dati per 'n+1' vengono caricati e i risultati di 'n-1'

\* vengono scaricati.

\*/

class ff\_node\_acc\_t : public ff\_node {

public:

explicit ff\_node\_acc\_t(IAccelerator \*acc, StatsCollector \*stats);

~ff\_node\_acc\_t() override;

protected:

int svc\_init() override;

void \*svc(void \*t) override;

void svc\_end() override;

private:

// Sentinella usata per segnalare la fine dello stream di dati nelle code interne ai

// thread.

static void \*const SENTINEL;

// Loops dei 2 stadi della pipeline interna.

void producerLoop();

void consumerLoop();

// Puntatori all'acceleratore e all'oggetto per le statistiche.

IAccelerator \*accelerator\_;

StatsCollector \*stats\_;

// Code per i task in ingresso dalla pipeline FF e per i task pronti per il download

// dal device all'host.

BlockingQueue<void \*> inQ\_;

BlockingQueue<void \*> readyQ\_;

std::thread producerTh\_, consumerTh\_;

};

// src/ff\_pipes\_nodes/ff\_node\_acc\_t.cpp

#include "ff\_node\_acc\_t.hpp"

/\*\*

\* @brief Implementazione del nodo FastFlow che orchestra l'offloading.

\*

\* Il nodo incapsula una pipeline interna a 2 stadi, gestita da due thread:

\* 1. Producer (Upload+Launch): Trasferisce i dati dall'host al device e

\* avvia l'esecuzione del kernel.

\* 2. Consumer (Download): Trasferisce i risultati dal device all'host.

\*/

// Sentinella usata per segnalare la fine dello stream di dati alla pipeline interna.

static char sentinel\_obj;

void \*const ff\_node\_acc\_t::SENTINEL = &sentinel\_obj;

/\*\*

\* @brief Costruttore del nodo.

\*

\* @param acc Puntatore a un'implementazione di IAccelerator.

\* @param stats Puntatore all'oggetto per le statistiche finali.

\*/

ff\_node\_acc\_t::ff\_node\_acc\_t(IAccelerator \*acc, StatsCollector \*stats)

: accelerator\_(acc), stats\_(stats) {}

ff\_node\_acc\_t::~ff\_node\_acc\_t() = default;

/\*\*

\* @brief Metodo di inizializzazione del nodo

\*/

int ff\_node\_acc\_t::svc\_init() {

std::cerr << "[Accelerator Node] Initializing...\n";

// Trova il tipo di acceleratore, crea il contesto e la coda di comandi

// OpenCL, legge il sorgente del kernel, lo compila e prepara l'oggetto

// kernel, inizializza il pool di buffer e la coda degli indici liberi.

if (!accelerator\_ || !accelerator\_->initialize()) {

std::cerr << "[ERROR] Accelerator setup failed.\n";

return -1;

}

// Avvia i due thread.

producerTh\_ = std::thread(&ff\_node\_acc\_t::producerLoop, this);

consumerTh\_ = std::thread(&ff\_node\_acc\_t::consumerLoop, this);

std::cerr << "[Accelerator Node] Internal 2-stage pipeline started.\n\n";

return 0;

}

/\*\*

\* @brief Metodo principale del nodo, chiamato da FF per ogni task.

\* Rice un task, lo inserisce nella inQ\_ e ritorna FF\_GO\_ON per indicare che è

\* pronto a ricevere un altro task.

\*/

void \*ff\_node\_acc\_t::svc(void \*task) {

// Se il task è un EOS, propaga la sentinella alla pipeline interna.

if (task == FF\_EOS) {

inQ\_.push(SENTINEL);

return FF\_EOS;

}

// Imposta l'ora di arrivo del task nel nodo.

static\_cast<Task \*>(task)->arrival\_time = std::chrono::steady\_clock::now();

inQ\_.push(task);

return FF\_GO\_ON;

}

/\*\*

\* @brief Loop per il 1° stadio della pipeline: Producer (Upload + Launch).

\*/

void ff\_node\_acc\_t::producerLoop() {

while (true) {

// Attende un task dalla coda di input.

void \*ptr = inQ\_.pop();

// Se riceve la sentinella, la propaga e termina.

if (ptr == SENTINEL) {

readyQ\_.push(SENTINEL);

break;

}

auto \*task = static\_cast<Task \*>(ptr);

// Acquisisce un buffer set, invia i dati sul device e avvia il kernel.

task->buffer\_idx = accelerator\_->acquire\_buffer\_set();

accelerator\_->send\_data\_to\_device(task);

accelerator\_->execute\_kernel(task);

readyQ\_.push(task);

}

}

/\*\*

\* @brief Loop per il 2° stadio della pipeline: Consumer (Download).

\*/

void ff\_node\_acc\_t::consumerLoop() {

// Memorizza l'ora di completamento del task precedente.

std::chrono::steady\_clock::time\_point last\_completion\_time;

bool first\_task = true;

while (true) {

// Prende un task pronto dalla coda.

void \*ptr = readyQ\_.pop();

if (ptr == SENTINEL) {

// La pipeline è vuota. Comunica il conteggio finale.

stats\_->count\_promise.set\_value(stats\_->tasks\_processed.load());

break;

}

auto \*task = static\_cast<Task \*>(ptr);

long long current\_task\_ns = 0;

// Attende il completamento del kernel e scarica i risultati sull'host.

accelerator\_->get\_results\_from\_device(task, current\_task\_ns);

auto end\_time = std::chrono::steady\_clock::now();

// Calcola il tempo nel nodo per questo task.

auto inNode\_duration = std::chrono::duration\_cast<std::chrono::nanoseconds>(

end\_time - task->arrival\_time);

// Calcola il tempo dall'ultimo completamento.

if (!first\_task) {

auto inter\_completion\_duration =

std::chrono::duration\_cast<std::chrono::nanoseconds>(end\_time -

last\_completion\_time);

stats\_->inter\_completion\_time\_ns += inter\_completion\_duration.count();

} else

first\_task = false;

last\_completion\_time = end\_time;

// Aggiorna le statistiche.

stats\_->computed\_ns += current\_task\_ns;

stats\_->total\_InNode\_time\_ns += inNode\_duration.count();

stats\_->tasks\_processed++;

accelerator\_->release\_buffer\_set(task->buffer\_idx);

delete task;

}

}

/\*\*

\* @brief Metodo di terminazione, chiamato da FF. Invia la sentinella ai

\* thread interni e attende la loro terminazione.

\*/

void ff\_node\_acc\_t::svc\_end() {

inQ\_.push(SENTINEL);

if (producerTh\_.joinable())

producerTh\_.join();

if (consumerTh\_.joinable())

consumerTh\_.join();

std::cerr << "\n[Accelerator Node] Shutdown complete.\n";

}

// src/strategy\_accelerator/BufferManager.hpp

#pragma once

#include <condition\_variable>

#include <mutex>

#include <queue>

#include <vector>

#ifdef \_\_APPLE\_\_

#include <OpenCL/opencl.h>

#else

#include <CL/cl.h>

#endif

/\*\*

\* @brief Gestisce un pool di set di buffer OpenCL. Incapsula la logica per

\* l'acquisizione, il rilascio e la riallocazione dei buffer di memoria sul

\* device.

\*/

class BufferManager {

public:

explicit BufferManager(cl\_context context);

~BufferManager();

// Set di buffer, 2 per input e 1 per l'output.

struct BufferSet {

cl\_mem bufferA{nullptr};

cl\_mem bufferB{nullptr};

cl\_mem bufferC{nullptr};

};

// Metodi per l'acquisizione e il rilascio dei buffer.

size\_t acquire\_buffer\_set();

void release\_buffer\_set(size\_t index);

// Viene chiamata la prima volta o quando un task arriva con una

// dimensione di dati diversa da quella per cui i buffer sono stati allocati.

bool reallocate\_buffers\_if\_needed(size\_t required\_size\_bytes);

// Restituisce un riferimento a un set di buffer specifico.

BufferSet &get\_buffer\_set(size\_t index);

private:

cl\_context context\_; // Contesto OpenCL per creare i buffer

// Dati per il pool di buffer nel device e per la gestione della concorrenza.

const size\_t POOL\_SIZE =

3; // ! Pool size OTTIMALE perchè con N = 7.449.999, i buffer per i vettori di input o

// ! output richiedono ~30MB l'uno, un set di buffer richiede quindi 90MB => sto

// ! già allocando POOL\_SIZE x 90 = 270MB di VRAM, se aumentassi il pool size

// ! rischierei di rallentare l'OS o potrebber fallire l'alloc su FPGA, inoltre non

// ! aumenterebbe il throughput. Se usassi POOL\_SIZE = 100, dovrei allocare 9GB di

// ! VRAM su FPGA. Con POOL\_SIZE = 3 ho un buon compromesso fra performance e

// ! minimo utilizzo di memoria.

std::vector<BufferSet> buffer\_pool\_;

std::queue<size\_t> free\_buffer\_indices\_;

std::mutex pool\_mutex\_;

std::condition\_variable buffer\_available\_cond\_;

// Dimensione attualmente allocata per i buffer nel pool.

size\_t allocated\_size\_bytes\_{0};

};

// src/strategy\_accelerator/BufferManager.cpp

#include "BufferManager.hpp"

#include <iostream>

/\*\*

\* @brief Costruttore: inizializza il pool di buffer.

\*/

BufferManager::BufferManager(cl\_context context) : context\_(context) {

buffer\_pool\_.resize(POOL\_SIZE);

for (size\_t i = 0; i < POOL\_SIZE; ++i)

free\_buffer\_indices\_.push(i);

}

/\*\*

\* @brief Distruttore: rilascia tutti i buffer di memoria nel pool.

\*/

BufferManager::~BufferManager() {

for (auto &buffer\_set : buffer\_pool\_) {

if (buffer\_set.bufferA) {

clReleaseMemObject(buffer\_set.bufferA);

buffer\_set.bufferA = nullptr;

}

if (buffer\_set.bufferB) {

clReleaseMemObject(buffer\_set.bufferB);

buffer\_set.bufferB = nullptr;

}

if (buffer\_set.bufferC) {

clReleaseMemObject(buffer\_set.bufferC);

buffer\_set.bufferC = nullptr;

}

if (buffer\_set.bufferA != nullptr && buffer\_set.bufferB != nullptr &&

buffer\_set.bufferC != nullptr)

std::cerr

<< "[BufferManager] Warning: Some buffers were not released properly.\n";

}

}

/\*\*

\* @brief Acquisisce un indice di buffer dal pool. Se nessun buffer è

\* disponibile per un thread da acquisire, attende in modo non bloccante.

\*/

size\_t BufferManager::acquire\_buffer\_set() {

std::unique\_lock<std::mutex> lock(pool\_mutex\_);

// Attende finché non c'è un buffer libero.

buffer\_available\_cond\_.wait(lock, [this] { return !free\_buffer\_indices\_.empty(); });

// Thread risvegliato. Estrae e restituisce l'indice del buffer libero.

size\_t index = free\_buffer\_indices\_.front();

free\_buffer\_indices\_.pop();

return index;

}

/\*\*

\* @brief Rilascia un indice di buffer nel pool e notifica i thread in attesa.

\*/

void BufferManager::release\_buffer\_set(size\_t index) {

{

std::lock\_guard<std::mutex> lock(pool\_mutex\_);

free\_buffer\_indices\_.push(index);

}

buffer\_available\_cond\_.notify\_one();

}

/\*\*

\* @brief Helper per allocare o riallocare la memoria per tutti i buffer del

\* pool. Viene invocata al primo utilizzo o quando un task richiede una

\* dimensione di dati differente da quella corrente.

\*/

bool BufferManager::reallocate\_buffers\_if\_needed(size\_t required\_size\_bytes) {

if (allocated\_size\_bytes\_ == required\_size\_bytes)

return true; // Nessuna riallocazione necessaria

std::cerr << " [BufferManager - DEBUG] Allocating pool buffers for "

<< required\_size\_bytes << " bytes\n";

// Rilascia eventuali buffer esistenti.

for (auto &buffer\_set : buffer\_pool\_) {

if (buffer\_set.bufferA)

clReleaseMemObject(buffer\_set.bufferA);

if (buffer\_set.bufferB)

clReleaseMemObject(buffer\_set.bufferB);

if (buffer\_set.bufferC)

clReleaseMemObject(buffer\_set.bufferC);

}

// Alloca nuovi buffer.

cl\_int ret;

for (size\_t i = 0; i < POOL\_SIZE; ++i) {

buffer\_pool\_[i].bufferA =

clCreateBuffer(context\_, CL\_MEM\_READ\_ONLY, required\_size\_bytes, NULL, &ret);

buffer\_pool\_[i].bufferB =

clCreateBuffer(context\_, CL\_MEM\_READ\_ONLY, required\_size\_bytes, NULL, &ret);

buffer\_pool\_[i].bufferC =

clCreateBuffer(context\_, CL\_MEM\_WRITE\_ONLY, required\_size\_bytes, NULL, &ret);

if (ret != CL\_SUCCESS) {

std::cerr << "[ERROR] BufferManager: Failed to allocate buffer pool. If on "

"FPGA, maxium N "

"usable is 7449999.\n";

return false;

}

}

allocated\_size\_bytes\_ = required\_size\_bytes;

return true;

}

BufferManager::BufferSet &BufferManager::get\_buffer\_set(size\_t index) {

return buffer\_pool\_[index];

}

// src/strategy\_accelerator/Gpu\_OpenCL\_Accelerator.hpp

#pragma once

#include "BufferManager.hpp"

#include "IAccelerator.hpp"

#include <string>

#ifdef \_\_APPLE\_\_

#include <OpenCL/opencl.h>

#else

#include <CL/cl.h>

#endif

/\*\*

\* @brief Implementazione di IAccelerator che gestisce l'offloading su GPU.

\*

\* La pipeline interna al nodo ff\_node\_acc\_t è composta da 2 thread:

\* - Il thread Producer esegue gli stadi di Upload e Execute, utilizzando le funzioni qui

\* dichiarate send\_data\_to\_device() e execute\_kernel().

\* - Il thread Consumer esegue lo stadio di Download, utilizzando la funzione qui

\* dichiarata get\_results\_from\_device().

\*/

class Gpu\_OpenCL\_Accelerator : public IAccelerator {

public:

Gpu\_OpenCL\_Accelerator(const std::string &kernel\_path, const std::string &kernel\_name);

~Gpu\_OpenCL\_Accelerator() override;

// Esegue tutte le operazioni di setup una volta sola (creare contesto,

// coda comandi, compilare kernel, inizializzare pool buffer).

bool initialize() override;

// Metoodi utili per i thread della pipeline interna.

void send\_data\_to\_device(void \*task\_context) override;

void execute\_kernel(void \*task\_context) override;

void get\_results\_from\_device(void \*task\_context, long long &computed\_ns) override;

// Metodi per l'acquisizione e il rilascio dei buffer.

size\_t acquire\_buffer\_set() override;

void release\_buffer\_set(size\_t index) override;

private:

cl\_context context\_{nullptr}; // Il contesto OpenCL

cl\_command\_queue queue\_{nullptr}; // La coda di comandi OpenCL

cl\_program program\_{nullptr}; // Il programma OpenCL (kernel compilato)

cl\_kernel kernel\_{nullptr}; // Il kernel OpenCL (func da eseguire)

// Incapsula la logica per l'acquisizione, il rilascio e la riallocazione dei buffer di

// memoria sul device.

std::unique\_ptr<BufferManager> buffer\_manager\_;

std::string kernel\_path\_;

std::string kernel\_name\_;

};

// src/strategy\_accelerator/Gpu\_OpenCL\_Accelerator.cpp

#include "Gpu\_OpenCL\_Accelerator.hpp"

#include <chrono>

#include <filesystem>

#include <fstream>

#include <iostream>

#include <thread>

#include <vector>

/\*\*

\* @brief Implementazione della classe Gpu\_OpenCL\_Accelerator per l'offloading su GPU.

\*/

// Macro per il controllo degli errori OpenCL.

#define OCL\_CHECK(err\_code, call, on\_error\_action) \

do { \

err\_code = (call); \

if (err\_code != CL\_SUCCESS) { \

std::cerr << "[ERROR] OpenCL call `" #call "` failed with code " << err\_code \

<< " at " << \_\_FILE\_\_ << ":" << \_\_LINE\_\_ << std::endl; \

on\_error\_action; \

} \

} while (0)

/\*\*

\* @brief Il costruttrore prende in input il nome della funzione kernel e il suo path.

\*/

Gpu\_OpenCL\_Accelerator::Gpu\_OpenCL\_Accelerator(const std::string &kernel\_path,

const std::string &kernel\_name)

: kernel\_path\_(kernel\_path), kernel\_name\_(kernel\_name) {}

/\*\*

\* @brief Il distruttore si occupa di rilasciare in ordine inverso tutte le risorse OpenCL

\* allocate. La pulizia dei buffer è gestita automaticamente dal distruttore di

\* buffer\_manager\_.

\*/

Gpu\_OpenCL\_Accelerator::~Gpu\_OpenCL\_Accelerator() {

if (kernel\_)

clReleaseKernel(kernel\_);

if (program\_)

clReleaseProgram(program\_);

if (queue\_)

clReleaseCommandQueue(queue\_);

if (context\_)

clReleaseContext(context\_);

std::cerr << "[Gpu\_OpenCL\_Accelerator] Destroyed and resources released.\n";

}

/\*\*

\* @brief Esegue tutte le operazioni di setup una volta sola. Trova il dispositivo, crea

\* il contesto, la coda di comandi, legge il sorgente del kernel, lo compila e prepara

\* l'oggetto kernel, inizializza il pool di buffer e la coda degli indici liberi.

\*/

bool Gpu\_OpenCL\_Accelerator::initialize() {

cl\_int ret; // Codice di ritorno delle chiamate OpenCL

cl\_platform\_id platform\_id = NULL;

cl\_device\_id device\_id = NULL;

// Trova una piattaforma OpenCL e un dispositivo di tipo GPU.

OCL\_CHECK(ret, clGetPlatformIDs(1, &platform\_id, NULL), return false);

OCL\_CHECK(ret, clGetDeviceIDs(platform\_id, CL\_DEVICE\_TYPE\_GPU, 1, &device\_id, NULL), {

std::cerr << "[FATAL] GPU not found.\n";

exit(EXIT\_FAILURE);

});

// Crea un contesto OpenCL.

context\_ = clCreateContext(NULL, 1, &device\_id, NULL, NULL, &ret);

if (!context\_ || ret != CL\_SUCCESS) {

std::cerr << "[ERROR] Gpu\_OpenCL\_Accelerator: Failed to create OpenCL context.\n";

return false;

}

// Crea la coda di comandi.

queue\_ = clCreateCommandQueue(context\_, device\_id, 0, &ret);

if (!queue\_ || ret != CL\_SUCCESS) {

std::cerr << "[ERROR] Gpu\_OpenCL\_Accelerator: Failed to create command queue.\n";

return false;

}

// Chiama il costruttore di BufferManager che iniializza il pool di buffer.

buffer\_manager\_ = std::make\_unique<BufferManager>(context\_);

// Legge il kernel OpenCL e verifica che il percorso sia un file valido.

std::ifstream kernelFile(kernel\_path\_);

// Controllo che il file sia stato aperto correttamente e che abbia estensione .cl, poi

// lo leggo.

if (!kernelFile.is\_open() || !std::filesystem::is\_regular\_file(kernel\_path\_) ||

kernel\_path\_.rfind(".cl") == std::string::npos) {

std::cerr << "[ERROR] Gpu\_OpenCL\_Accelerator: Could not open kernel file: "

<< kernel\_path\_ << "\n";

exit(EXIT\_FAILURE);

}

std::string kernelSource((std::istreambuf\_iterator<char>(kernelFile)),

(std::istreambuf\_iterator<char>()));

const char \*source\_str = kernelSource.c\_str();

size\_t source\_size = kernelSource.length();

// Crea il programma OpenCL.

program\_ = clCreateProgramWithSource(context\_, 1, &source\_str, &source\_size, &ret);

if (!program\_ || ret != CL\_SUCCESS) {

std::cerr << "[ERROR] Gpu\_OpenCL\_Accelerator: Failed to create program.\n";

exit(EXIT\_FAILURE);

}

// Compila il programma OpenCL.

ret = clBuildProgram(program\_, 1, &device\_id, NULL, NULL, NULL);

if (ret != CL\_SUCCESS) {

std::cerr << "[ERROR] Gpu\_OpenCL\_Accelerator: Kernel "

"compilation failed.\n";

size\_t log\_size;

clGetProgramBuildInfo(program\_, device\_id, CL\_PROGRAM\_BUILD\_LOG, 0, NULL,

&log\_size);

std::vector<char> log(log\_size);

clGetProgramBuildInfo(program\_, device\_id, CL\_PROGRAM\_BUILD\_LOG, log\_size,

log.data(), NULL);

exit(EXIT\_FAILURE);

}

// Crea l'oggetto kernel.

kernel\_ = clCreateKernel(program\_, kernel\_name\_.c\_str(), &ret);

if (!kernel\_ || ret != CL\_SUCCESS) {

std::cerr << "[ERROR] Gpu\_OpenCL\_Accelerator: Failed to create kernel object.\n";

exit(EXIT\_FAILURE);

}

std::cerr << "[Gpu\_OpenCL\_Accelerator] Initialization successful.\n";

return true;

}

/\*\*

\* @brief Stadio 1 (Upload).

\* Fa l'upload dei dati di input A e B dall'host alla device memory. L' evento per la

\* sincronizzazione (`task->event`) viene generato solodall'ultima operazione, garantendo

\* che lo stadio successivo attenda ilcompletamento di entrambi i trasferimenti.

\*/

void Gpu\_OpenCL\_Accelerator::send\_data\_to\_device(void \*task\_context) {

cl\_int ret; // Codice di ritorno delle chiamate OpenCL

auto \*task = static\_cast<Task \*>(task\_context);

std::cerr << "[Gpu\_OpenCL\_Accelerator - START] Processing task " << task->id

<< " with N=" << task->n << "...\n";

// Se la dimensione richiesta è diversa da quella allocata, rialloca

// tutti i buffer del pool e ottieni il set di buffer.

size\_t required\_size\_bytes = sizeof(int) \* task->n;

buffer\_manager\_->reallocate\_buffers\_if\_needed(required\_size\_bytes);

auto &current\_buffers = buffer\_manager\_->get\_buffer\_set(task->buffer\_idx);

// Scrive i due input sulla device memory.

OCL\_CHECK(ret,

clEnqueueWriteBuffer(queue\_, current\_buffers.bufferA, CL\_FALSE, 0,

required\_size\_bytes, task->a, 0, NULL, NULL),

return);

OCL\_CHECK(ret,

clEnqueueWriteBuffer(queue\_, current\_buffers.bufferB, CL\_FALSE, 0,

required\_size\_bytes, task->b, 0, NULL, &task->event),

return);

}

/\*\*

\* @brief Stadio 2 (Execute).

\* Imposta gli argomenti del kernel e accoda la sua esecuzione, rilasciando l'evento del

\* completamento del trasferimento dati e ottenendo un nuovo eventoche rappresenta il

\* completamento del kernel.

\*/

void Gpu\_OpenCL\_Accelerator::execute\_kernel(void \*task\_context) {

cl\_int ret; // Codice di ritorno delle chiamate OpenCL.

auto \*task = static\_cast<Task \*>(task\_context);

auto &current\_buffers = buffer\_manager\_->get\_buffer\_set(task->buffer\_idx);

cl\_event previous\_event = task->event;

// Imposta gli argomenti del kernel.

OCL\_CHECK(ret, clSetKernelArg(kernel\_, 0, sizeof(cl\_mem), &current\_buffers.bufferA),

return);

OCL\_CHECK(ret, clSetKernelArg(kernel\_, 1, sizeof(cl\_mem), &current\_buffers.bufferB),

return);

OCL\_CHECK(ret, clSetKernelArg(kernel\_, 2, sizeof(cl\_mem), &current\_buffers.bufferC),

return);

OCL\_CHECK(ret, clSetKernelArg(kernel\_, 3, sizeof(unsigned int), &(task->n)), return);

// Accoda l'esecuzione del kernel.

size\_t global\_work\_size = task->n;

OCL\_CHECK(ret,

clEnqueueNDRangeKernel(queue\_, kernel\_, 1, NULL, &global\_work\_size, NULL, 1,

&previous\_event, &task->event),

return);

// Rilascia l'evento precedente.

if (previous\_event)

clReleaseEvent(previous\_event);

}

/\*\*

\* @brief Stadio 3 (Download).

\* Punto di sincronizzaione. Recupera i risultati dalla device memory alla memoria host,

\* aspettando che l'upload e l'esecuzione del kernel siano completati. È l'unica funzione

\* bloccante della pipeline.

\*/

void Gpu\_OpenCL\_Accelerator::get\_results\_from\_device(void \*task\_context,

long long &computed\_ns) {

cl\_int ret; // Codice di ritorno delle chiamate OpenCL

auto \*task = static\_cast<Task \*>(task\_context);

size\_t required\_size\_bytes = sizeof(int) \* task->n;

auto &current\_buffers = buffer\_manager\_->get\_buffer\_set(task->buffer\_idx);

cl\_event previous\_event = task->event;

auto t0 = std::chrono::steady\_clock::now();

// Recupera i risultati dalla device memory alla memoria host.

OCL\_CHECK(ret,

clEnqueueReadBuffer(queue\_, current\_buffers.bufferC, CL\_TRUE, 0,

required\_size\_bytes, task->c, 1, &previous\_event, NULL),

return);

// Rilascia l'evento precedente.

if (previous\_event)

clReleaseEvent(previous\_event);

task->event = nullptr;

// Calcola il tempo impiegato

auto t1 = std::chrono::steady\_clock::now();

computed\_ns = std::chrono::duration\_cast<std::chrono::nanoseconds>(t1 - t0).count();

std::cerr << "[Gpu\_OpenCL\_Accelerator - END] Task " << task->id << " finished.\n";

}

// ------------------------------------------------------------------------

// Metodi per l'acquisizione e il rilascio dei buffer

// ------------------------------------------------------------------------

size\_t Gpu\_OpenCL\_Accelerator::acquire\_buffer\_set() {

return buffer\_manager\_->acquire\_buffer\_set();

}

void Gpu\_OpenCL\_Accelerator::release\_buffer\_set(size\_t index) {

buffer\_manager\_->release\_buffer\_set(index);

}

// src/strategy\_accelerator/Gpu\_Metal\_Accelerator.hpp

#pragma once

#include "IAccelerator.hpp"

#include <memory>

#include <string>

class MetalBufferManager;

/\*\*

\* @brief Implementazione di IAccelerator che gestisce l'offloading su GPU Apple

\* tramite il framework nativo Metal.

\*/

class Gpu\_Metal\_Accelerator : public IAccelerator {

public:

Gpu\_Metal\_Accelerator(const std::string &kernel\_path, const std::string &kernel\_name);

~Gpu\_Metal\_Accelerator() override;

// Esegue tutte le operazioni di setup una volta sola (trovare device,

// creare coda comandi, compilare kernel .metal).

bool initialize() override;

// Metoodi utili per i thread della pipeline interna.

void send\_data\_to\_device(void \*task\_context) override;

void execute\_kernel(void \*task\_context) override;

void get\_results\_from\_device(void \*task\_context, long long &computed\_ns) override;

// Metodi per l'acquisizione e il rilascio dei buffer.

size\_t acquire\_buffer\_set() override;

void release\_buffer\_set(size\_t index) override;

private:

// --- Oggetti Metal ---

void \*device\_{nullptr}; // Puntatore al device Metal (id<MTLDevice>).

void \*command\_queue\_{nullptr}; // Puntatore alla coda di comandi (id<MTLCommandQueue>).

void \*library\_{

nullptr}; // Puntatore alla libreria dei kernel compilati (id<MTLLibrary>).

void \*kernel\_function\_{

nullptr}; // Puntatore alla funzione kernel specifica (id<MTLFunction>).

void \*pipeline\_state\_{nullptr}; // Puntatore allo stato della pipeline di calcolo

// (id<MTLComputePipelineState>).

// La logica del pool di buffer è incapsulata in un oggetto specifico per Metal.

std::unique\_ptr<MetalBufferManager> buffer\_manager\_;

std::string kernel\_path\_;

std::string kernel\_name\_;

};

// src/strategy\_accelerator/ Gpu\_Metal\_Accelerator.mm

#import "Gpu\_Metal\_Accelerator.hpp"

#include "../../common/Task.hpp"

#import <Metal/Metal.h>

#include <chrono>

#include <condition\_variable>

#include <filesystem>

#include <fstream>

#include <iostream>

#include <mutex>

#include <queue>

#include <vector>

/\*\*

\*

\* File contente GPU Metal Accelerator e Buffer Manager specifico per il framework Metal.

\*

\*/

// =======================================================================

// BufferManager specifico per Metal.

// Gestisce il pool di buffer di memoria sulla GPU.

// =======================================================================

class MetalBufferManager {

public:

// Set di buffer, 2 per input e 1 per l'output.

struct BufferSet {

id<MTLBuffer> bufferA{nullptr};

id<MTLBuffer> bufferB{nullptr};

id<MTLBuffer> bufferC{nullptr};

};

/\*\*

\* Costruttore: inizializza il pool di buffer.

\*/

explicit MetalBufferManager(id<MTLDevice> device) : device\_(device) {

buffer\_pool\_.resize(POOL\_SIZE);

for (size\_t i = 0; i < POOL\_SIZE; ++i)

free\_buffer\_indices\_.push(i);

}

// ARC di Objective-C rilascia automaticamente gli oggetti MTLBuffer.

~MetalBufferManager() {}

/\*\*

\* Acquisisce un indice di buffer dal pool. Se nessun buffer è

\* disponibile per un thread da acquisire, attende in modo non bloccante.

\*/

size\_t acquire\_buffer\_set() {

std::unique\_lock<std::mutex> lock(pool\_mutex\_);

// Attende finché non c'è un buffer libero.

buffer\_available\_cond\_.wait(lock, [this] { return !free\_buffer\_indices\_.empty(); });

// Th risvegliato. Estrae e restituisce l'indice del buffer libero.

size\_t index = free\_buffer\_indices\_.front();

free\_buffer\_indices\_.pop();

return index;

}

/\*\*

\* Rilascia un indice di buffer nel pool e notifica i thread in attesa.

\*/

void release\_buffer\_set(size\_t index) {

{

std::lock\_guard<std::mutex> lock(pool\_mutex\_);

free\_buffer\_indices\_.push(index);

}

buffer\_available\_cond\_.notify\_one();

}

bool reallocate\_buffers\_if\_needed(size\_t required\_size\_bytes) {

if (allocated\_size\_bytes\_ == required\_size\_bytes)

return true;

allocated\_size\_bytes\_ = required\_size\_bytes;

// Su Apple Silicon la memoria è condivisa tra CPU e GPU, quindi possiamo accedere

// agli stessi dati senza copie esplicite sul bus PCIe.

MTLResourceOptions options = MTLResourceStorageModeShared;

for (size\_t i = 0; i < POOL\_SIZE; ++i) {

buffer\_pool\_[i].bufferA = [device\_ newBufferWithLength:required\_size\_bytes

options:options];

buffer\_pool\_[i].bufferB = [device\_ newBufferWithLength:required\_size\_bytes

options:options];

buffer\_pool\_[i].bufferC = [device\_ newBufferWithLength:required\_size\_bytes

options:options];

if (!buffer\_pool\_[i].bufferA || !buffer\_pool\_[i].bufferB ||

!buffer\_pool\_[i].bufferC) {

std::cerr << "[ERROR] MetalBufferManager: Failed to allocate "

"buffer pool.\n";

return false;

}

}

std::cerr << " [MetalBufferManager - DEBUG] Allocating pool buffers for "

<< required\_size\_bytes << " bytes\n";

return true;

}

BufferSet &get\_buffer\_set(size\_t index) { return buffer\_pool\_[index]; }

private:

id<MTLDevice> device\_; // Riferimento al device Metal.

// Dati per il pool di buffer nel device e per la gestione della concorrenza.

const size\_t POOL\_SIZE = 3; // ! POOL\_SIZE ottimale.

std::vector<BufferSet> buffer\_pool\_;

std::queue<size\_t> free\_buffer\_indices\_;

std::mutex pool\_mutex\_;

std::condition\_variable buffer\_available\_cond\_;

// Dimensione attualmente allocata per i buffer nel pool.

size\_t allocated\_size\_bytes\_{0};

};

// =======================================================================

// =======================================================================

// Gpu\_Metal\_Accelerator

// =======================================================================

Gpu\_Metal\_Accelerator::Gpu\_Metal\_Accelerator(const std::string &kernel\_path,

const std::string &kernel\_name)

: kernel\_path\_(kernel\_path), kernel\_name\_(kernel\_name) {}

/\*\*

\* Il distruttore usa \_\_bridge\_transfer per passare la proprietà dei puntatori C di nuovo

\* ad ARC, che li rilascerà correttamente.

\*/

Gpu\_Metal\_Accelerator::~Gpu\_Metal\_Accelerator() {

if (pipeline\_state\_)

id<MTLComputePipelineState> pso =

(\_\_bridge\_transfer id<MTLComputePipelineState>)pipeline\_state\_;

if (kernel\_function\_)

id<MTLFunction> func = (\_\_bridge\_transfer id<MTLFunction>)kernel\_function\_;

if (library\_)

id<MTLLibrary> lib = (\_\_bridge\_transfer id<MTLLibrary>)library\_;

if (command\_queue\_)

id<MTLCommandQueue> queue = (\_\_bridge\_transfer id<MTLCommandQueue>)command\_queue\_;

if (device\_)

id<MTLDevice> dev = (\_\_bridge\_transfer id<MTLDevice>)device\_;

buffer\_manager\_.reset();

std::cerr << "[Gpu\_Metal\_Accelerator] Destroyed and resources released.\n";

}

/\*\*

\* @brief Esegue tutte le operazioni di setup.

\*/

bool Gpu\_Metal\_Accelerator::initialize() {

// Trova un device che supporta Metal.

device\_ = (\_\_bridge\_retained void \*)MTLCreateSystemDefaultDevice();

if (!device\_) {

std::cerr << "[FATAL] Metal GPU not found.\n";

exit(EXIT\_FAILURE);

}

// Crea la coda di comandi.

id<MTLDevice> dev = (\_\_bridge id<MTLDevice>)device\_;

command\_queue\_ = (\_\_bridge\_retained void \*)[dev newCommandQueue];

if (!command\_queue\_) {

std::cerr << "[ERROR] Gpu\_Metal\_Accelerator: Failed to create command queue.\n";

exit(EXIT\_FAILURE);

}

// Chiama il costruttore di MetalBufferManager che inializza il pool di buffer.

buffer\_manager\_ = std::make\_unique<MetalBufferManager>(dev);

// Legge il kernel Metal e verifica che il percorso sia un file valido.

std::ifstream kernelFile(kernel\_path\_);

if (!kernelFile.is\_open() || !std::filesystem::is\_regular\_file(kernel\_path\_)) {

std::cerr << "[ERROR] Gpu\_Metal\_Accelerator: Could not open kernel file: "

<< kernel\_path\_ << "\n";

exit(EXIT\_FAILURE);

}

std::string kernelSource((std::istreambuf\_iterator<char>(kernelFile)),

std::istreambuf\_iterator<char>(nullptr));

NSError \*error = nil;

// Compila il sorgente del kernel .metal in una libreria.

id<MTLLibrary> lib =

[dev newLibraryWithSource:[NSString stringWithUTF8String:kernelSource.c\_str()]

options:nil

error:&error];

if (!lib) {

std::cerr << "\n\n------- ERRORE DI COMPILAZIONE KERNEL METAL -------\n";

std::cerr << [error.localizedDescription UTF8String] << "\n";

std::cerr << "--------------------------------------------------\n\n";

std::cerr << "[ERROR] Gpu\_Metal\_Accelerator: Kernel library compilation "

"failed. Check kernel file type.\n";

exit(EXIT\_FAILURE);

}

library\_ = (\_\_bridge\_retained void \*)lib;

// Ottiene la funzione kernel.

id<MTLFunction> func =

[lib newFunctionWithName:[NSString stringWithUTF8String:kernel\_name\_.c\_str()]];

if (!func) {

std::cerr << "[ERROR] Gpu\_Metal\_Accelerator: Failed to find kernel function '"

<< kernel\_name\_ << "'.\n";

exit(EXIT\_FAILURE);

}

kernel\_function\_ = (\_\_bridge\_retained void \*)func;

// Crea lo stato della pipeline di calcolo (oggetto che rappresenta il kernel

// compilato).

id<MTLComputePipelineState> pso = [dev newComputePipelineStateWithFunction:func

error:&error];

if (!pso) {

std::cerr << "[ERROR] Gpu\_Metal\_Accelerator: Failed to create pipeline state "

"object.\n";

if (error) {

std::cerr << "Reason: " << [[error localizedDescription] UTF8String] << "\n";

}

exit(EXIT\_FAILURE);

}

pipeline\_state\_ = (\_\_bridge\_retained void \*)pso;

std::cerr << "[Gpu\_Metal\_Accelerator] Initialization successful.\n";

return true;

}

/\*\*

\* @brief Stadio 1 (Upload).

\*/

void Gpu\_Metal\_Accelerator::send\_data\_to\_device(void \*task\_context) {

auto \*task = static\_cast<Task \*>(task\_context);

std::cerr << "[Gpu\_Metal\_Accelerator - START] Processing task " << task->id

<< " with N=" << task->n << "...\n";

// Se la dimensione richiesta è diversa da quella allocata, rialloca tutti i buffer del

// pool e ottieni il set di buffer.

size\_t required\_size\_bytes = sizeof(int) \* task->n;

buffer\_manager\_->reallocate\_buffers\_if\_needed(required\_size\_bytes);

auto &current\_buffers = buffer\_manager\_->get\_buffer\_set(task->buffer\_idx);

// Grazie alla memoria unificata, copia i dati direttamente.

memcpy([current\_buffers.bufferA contents], task -> a, required\_size\_bytes);

memcpy([current\_buffers.bufferB contents], task -> b, required\_size\_bytes);

}

/\*\*

\* @brief Stadio 2 (Execute).

\*/

void Gpu\_Metal\_Accelerator::execute\_kernel(void \*task\_context) {

auto \*task = static\_cast<Task \*>(task\_context);

auto &current\_buffers = buffer\_manager\_->get\_buffer\_set(task->buffer\_idx);

// "Prende in prestito" i puntatori agli oggetti Metal per usarli in questa funzione.

id<MTLCommandQueue> queue = (\_\_bridge id<MTLCommandQueue>)command\_queue\_;

id<MTLComputePipelineState> pso =

(\_\_bridge id<MTLComputePipelineState>)pipeline\_state\_;

// Crea un contenitore per i comandi da inviare alla GPU.

id<MTLCommandBuffer> command\_buffer = [queue commandBuffer];

// Crea un "codificatore" per scrivere i comandi di calcolo.

id<MTLComputeCommandEncoder> encoder = [command\_buffer computeCommandEncoder];

// Imposta il kernel e i suoi argomenti (i buffer).

[encoder setComputePipelineState:pso];

[encoder setBuffer:current\_buffers.bufferA offset:0 atIndex:0];

[encoder setBuffer:current\_buffers.bufferB offset:0 atIndex:1];

[encoder setBuffer:current\_buffers.bufferC offset:0 atIndex:2];

unsigned int n\_uint = task->n;

[encoder setBytes:&n\_uint length:sizeof(unsigned int) atIndex:3];

// Definisce la griglia di calcolo (quanti thread lanciare).

MTLSize grid\_size = MTLSizeMake(task->n, 1, 1);

NSUInteger thread\_group\_width = [pso maxTotalThreadsPerThreadgroup];

if (thread\_group\_width > task->n) {

thread\_group\_width = task->n;

}

MTLSize thread\_group\_size = MTLSizeMake(thread\_group\_width, 1, 1);

// Accoda il comando di esecuzione del kernel.

[encoder dispatchThreads:grid\_size threadsPerThreadgroup:thread\_group\_size];

// Finalizza la codifica dei comandi.

[encoder endEncoding];

// Invia il command buffer alla GPU per l'esecuzione asincrona.

[command\_buffer commit];

// Salva il command buffer nel task per la sincronizzazione successiva.

task->sync\_handle = (\_\_bridge\_retained void \*)command\_buffer;

}

/\*\*

\* @brief Stadio 3 (Download).

\*/

void Gpu\_Metal\_Accelerator::get\_results\_from\_device(void \*task\_context,

long long &computed\_ns) {

auto \*task = static\_cast<Task \*>(task\_context);

// Recupera il command buffer dal task e riprende la sua proprietà.

id<MTLCommandBuffer> command\_buffer =

(\_\_bridge\_transfer id<MTLCommandBuffer>)task->sync\_handle;

auto t0 = std::chrono::steady\_clock::now();

// Attende il completamento del kernel (op. bloccante ma va bene perchè il consumerLoop

// ha un solo e unico scopo: aspettare che la GPU finisca e poi copiare i dati).

[command\_buffer waitUntilCompleted];

auto t1 = std::chrono::steady\_clock::now();

computed\_ns = std::chrono::duration\_cast<std::chrono::nanoseconds>(t1 - t0).count();

// Copia i risultati indietro nella memoria host.

size\_t required\_size\_bytes = sizeof(int) \* task->n;

auto &current\_buffers = buffer\_manager\_->get\_buffer\_set(task->buffer\_idx);

memcpy(task->c, [current\_buffers.bufferC contents], required\_size\_bytes);

std::cerr << "[Gpu\_Metal\_Accelerator - END] Task " << task->id << " finished.\n";

}

// ------------------------------------------------------------------------

// Metodi per l'acquisizione e il rilascio dei buffer

// ------------------------------------------------------------------------

size\_t Gpu\_Metal\_Accelerator::acquire\_buffer\_set() {

return buffer\_manager\_->acquire\_buffer\_set();

}

void Gpu\_Metal\_Accelerator::release\_buffer\_set(size\_t index) {

buffer\_manager\_->release\_buffer\_set(index);

}

// src/strategy\_accelerator/Fpga\_Accelerator.hpp

#pragma once

#include "BufferManager.hpp"

#include "IAccelerator.hpp"

#include <string>

#ifdef \_\_APPLE\_\_

#include <OpenCL/opencl.h>

#else

#include <CL/cl.h>

#endif

/\*\*

\* @brief Implementazione di IAccelerator che gestisce l'offloading su FPGA.

\*

\* La pipeline interna al nodo ff\_node\_acc\_t è composta da 2 thread:

\* - Il thread Producer esegue gli stadi di Upload e Execute, utilizzando le

\* funzioni qui dichiarate send\_data\_to\_device() e execute\_kernel().

\* - Il thread Consumer esegue lo stadio di Download, utilizzando la

\* funzione qui dichiarata get\_results\_from\_device().

\*/

class Fpga\_Accelerator : public IAccelerator {

public:

Fpga\_Accelerator(const std::string &kernel\_path, const std::string &kernel\_name);

~Fpga\_Accelerator() override;

// Esegue tutte le operazioni di setup una volta sola (creare contesto,

// coda comandi, compilare kernel, inizializzare pool buffer).

bool initialize() override;

// Metoodi utili per i thread della pipeline interna.

void send\_data\_to\_device(void \*task\_context) override;

void execute\_kernel(void \*task\_context) override;

void get\_results\_from\_device(void \*task\_context, long long &computed\_ns) override;

// Metodi per l'acquisizione e il rilascio dei buffer.

size\_t acquire\_buffer\_set() override;

void release\_buffer\_set(size\_t index) override;

private:

cl\_context context\_{nullptr}; // Il contesto OpenCL

cl\_command\_queue queue\_{nullptr}; // La coda di comandi OpenCL

cl\_program program\_{nullptr}; // Il programma OpenCL (kernel compilato)

cl\_kernel kernel\_{nullptr}; // Il kernel OpenCL (func da eseguire)

// Incapsula la logica per l'acquisizione, il rilascio e la riallocazione dei

// buffer di memoria sul device.

std::unique\_ptr<BufferManager> buffer\_manager\_;

std::string kernel\_path\_;

std::string kernel\_name\_;

};

// src/strategy\_accelerator/ Fpga\_Accelerator.cpp

#include "Fpga\_Accelerator.hpp"

#include <chrono>

#include <filesystem>

#include <fstream>

#include <iostream>

#include <thread>

#include <vector>

/\*\*

\* @brief Implementazione della classe Fpga\_Accelerator per l'offloading su FPGA.

\*/

// Macro per il controllo degli errori OpenCL.

#define OCL\_CHECK(err\_code, call, on\_error\_action) \

do { \

err\_code = (call); \

if (err\_code != CL\_SUCCESS) { \

std::cerr << "[ERROR] OpenCL call `" #call "` failed with code " << err\_code \

<< " at " << \_\_FILE\_\_ << ":" << \_\_LINE\_\_ << std::endl; \

on\_error\_action; \

} \

} while (0)

/\*\*

\* @brief Il costruttrore prende in input il nome della funzione kernel e il suo

\* path.

\*/

Fpga\_Accelerator::Fpga\_Accelerator(const std::string &kernel\_path,

const std::string &kernel\_name)

: kernel\_path\_(kernel\_path), kernel\_name\_(kernel\_name) {}

/\*\*

\* @brief Il distruttore si occupa di rilasciare in ordine inverso tutte le

risorse OpenCL allocate. La pulizia dei buffer è gestita automaticamente dal

distruttore di buffer\_manager\_.

\*/

Fpga\_Accelerator::~Fpga\_Accelerator() {

if (kernel\_)

clReleaseKernel(kernel\_);

if (program\_)

clReleaseProgram(program\_);

if (queue\_)

clReleaseCommandQueue(queue\_);

if (context\_)

clReleaseContext(context\_);

std::cerr << "[Fpga\_Accelerator] Destroyed and resources released.\n";

}

/\*\*

\* @brief Esegue tutte le operazioni di setup una volta sola. Trova il

\* dispositivo, crea il contesto, la coda di comandi, legge il sorgente del

\* kernel, lo compila e prepara l'oggetto kernel, inizializza il pool di buffer

\* e la coda degli indici liberi.

\*/

bool Fpga\_Accelerator::initialize() {

cl\_int ret; // Codice di ritorno delle chiamate OpenCL.

cl\_platform\_id platform\_id = NULL;

cl\_device\_id device\_id = NULL;

// Trova una piattaforma OpenCL e un dispositivo di tipo ACCELERATOR.

OCL\_CHECK(ret, clGetPlatformIDs(1, &platform\_id, NULL), return false);

OCL\_CHECK(

ret, clGetDeviceIDs(platform\_id, CL\_DEVICE\_TYPE\_ACCELERATOR, 1, &device\_id, NULL), {

std::cerr << "[FATAL] FPGA not found.\n";

exit(EXIT\_FAILURE);

});

// Crea un contesto

context\_ = clCreateContext(NULL, 1, &device\_id, NULL, NULL, &ret);

if (!context\_) {

std::cerr << "[ERROR] Fpga\_Accelerator: Failed creating OpenCL context.\n";

return false;

}

// Crea la coda di comandi

queue\_ = clCreateCommandQueue(context\_, device\_id, 0, &ret);

if (!queue\_) {

std::cerr << "[ERROR] Fpga\_Accelerator: Failed to create command queue.\n";

return false;

}

// Chiama il costruttore di BufferManager che iniializza il pool di buffer.

buffer\_manager\_ = std::make\_unique<BufferManager>(context\_);

// Caricamento del file binario dell'FPGA (.xclbin).

std::ifstream binaryFile(kernel\_path\_, std::ios::binary);

// Controllo che il file sia stato aperto correttamente e che abbia

// estensione .xclbin, poi lo leggo.

if (!binaryFile.is\_open() || !std::filesystem::is\_regular\_file(kernel\_path\_) ||

kernel\_path\_.rfind(".xclbin") == std::string::npos) {

std::cerr << "[ERROR] Fpga\_Accelerator: Could not open kernel file: "

<< kernel\_path\_ << "\n";

exit(EXIT\_FAILURE);

}

binaryFile.seekg(0, binaryFile.end);

size\_t binarySize = binaryFile.tellg();

binaryFile.seekg(0, binaryFile.beg);

std::vector<unsigned char> kernelBinary(binarySize);

binaryFile.read(reinterpret\_cast<char \*>(kernelBinary.data()), binarySize);

const unsigned char \*binaries[] = {kernelBinary.data()};

const size\_t binary\_sizes[] = {binarySize};

// Crea il programma con il binario xclbin caricato.

program\_ = clCreateProgramWithBinary(context\_, 1, &device\_id, binary\_sizes, binaries,

NULL, &ret);

if (!program\_ || ret != CL\_SUCCESS) {

std::cerr << "[ERROR] Fpga\_Accelerator: Failed to create program from binary.\n";

exit(EXIT\_FAILURE);

}

// NON è necessaria la chiamata a clBuildProgram(), il programma è già

// compilato => l'inizializzazione dell'FPGA è molto più veloce di

// quella della GPU.

// Crea il kernel.

kernel\_ = clCreateKernel(program\_, kernel\_name\_.c\_str(), &ret);

if (!kernel\_ || ret != CL\_SUCCESS) {

std::cerr << "[ERROR] Fpga\_Accelerator: Failed to create kernel.\n";

exit(EXIT\_FAILURE);

}

std::cerr << "[Fpga\_Accelerator] Initialization successful.\n";

return true;

}

/\*\*

\* @brief Stadio 1 (Upload).

\* Fa l'upload dei dati di input A e B dall'host alla device memory.

\* L'evento per la sincronizzazione (`task->event`) viene generato solo

\* dall'ultima operazione, garantendo che lo stadio successivo attenda il

\* completamento di entrambi i trasferimenti.

\*/

void Fpga\_Accelerator::send\_data\_to\_device(void \*task\_context) {

cl\_int ret; // Codice di ritorno delle chiamate OpenCL

auto \*task = static\_cast<Task \*>(task\_context);

std::cerr << "[Fpga\_Accelerator - START] Processing task " << task->id

<< " with N=" << task->n << "...\n";

// Se la dimensione richiesta è diversa da quella allocata, rialloca

// tutti i buffer del pool e ottieni il set di buffer.

size\_t required\_size\_bytes = sizeof(int) \* task->n;

buffer\_manager\_->reallocate\_buffers\_if\_needed(required\_size\_bytes);

auto &current\_buffers = buffer\_manager\_->get\_buffer\_set(task->buffer\_idx);

// Scrive i due input sulla device memory.

OCL\_CHECK(ret,

clEnqueueWriteBuffer(queue\_, current\_buffers.bufferA, CL\_FALSE, 0,

required\_size\_bytes, task->a, 0, NULL, NULL),

return);

OCL\_CHECK(ret,

clEnqueueWriteBuffer(queue\_, current\_buffers.bufferB, CL\_FALSE, 0,

required\_size\_bytes, task->b, 0, NULL, &task->event),

return);

}

/\*\*

\* @brief Stadio 2 (Execute).

\* Imposta gli argomenti del kernel e accoda la sua esecuzione, rilasciando

\* l'evento del completamento del trasferimento dati e ottenendo un nuovo evento

\* che rappresenta il completamento del kernel.

\*/

void Fpga\_Accelerator::execute\_kernel(void \*task\_context) {

cl\_int ret; // Codice di ritorno delle chiamate OpenCL.

auto \*task = static\_cast<Task \*>(task\_context);

auto &current\_buffers = buffer\_manager\_->get\_buffer\_set(task->buffer\_idx);

cl\_event previous\_event = task->event;

// Imposta gli argomenti del kernel.

OCL\_CHECK(ret, clSetKernelArg(kernel\_, 0, sizeof(cl\_mem), &current\_buffers.bufferA),

return);

OCL\_CHECK(ret, clSetKernelArg(kernel\_, 1, sizeof(cl\_mem), &current\_buffers.bufferB),

return);

OCL\_CHECK(ret, clSetKernelArg(kernel\_, 2, sizeof(cl\_mem), &current\_buffers.bufferC),

return);

OCL\_CHECK(ret, clSetKernelArg(kernel\_, 3, sizeof(int), &(task->n)), return);

// Accoda l'esecuzione del kernel.

OCL\_CHECK(ret, clEnqueueTask(queue\_, kernel\_, 1, &previous\_event, &task->event),

return);

// Rilascia l'evento precedente.

if (previous\_event)

clReleaseEvent(previous\_event);

}

/\*\*

\* @brief Stadio 3 (Download).

\* Punto di sincronizzaione. Recupera i risultati dalla device memory alla

\* memoria host, aspettando che l'upload e l'esecuzione del kernel siano

\* completati. È l'unica funzione bloccante della pipeline.

\*/

void Fpga\_Accelerator::get\_results\_from\_device(void \*task\_context,

long long &computed\_ns) {

cl\_int ret; // Codice di ritorno delle chiamate OpenCL

auto \*task = static\_cast<Task \*>(task\_context);

size\_t required\_size\_bytes = sizeof(int) \* task->n;

auto &current\_buffers = buffer\_manager\_->get\_buffer\_set(task->buffer\_idx);

cl\_event previous\_event = task->event;

auto t0 = std::chrono::steady\_clock::now();

// Recupera i risultati dalla device memory alla memoria host.

OCL\_CHECK(ret,

clEnqueueReadBuffer(queue\_, current\_buffers.bufferC, CL\_TRUE, 0,

required\_size\_bytes, task->c, 1, &previous\_event, NULL),

return);

// Rilascia l'evento precedente.

if (previous\_event)

clReleaseEvent(previous\_event);

task->event = nullptr;

// Calcola il tempo impiegato.

auto t1 = std::chrono::steady\_clock::now();

computed\_ns = std::chrono::duration\_cast<std::chrono::nanoseconds>(t1 - t0).count();

std::cerr << "[Fpga\_Accelerator - END] Task " << task->id << " finished.\n";

}

// ------------------------------------------------------------------------

// Metodi per l'acquisizione e il rilascio dei buffer

// ------------------------------------------------------------------------

size\_t Fpga\_Accelerator::acquire\_buffer\_set() {

return buffer\_manager\_->acquire\_buffer\_set();

}

void Fpga\_Accelerator::release\_buffer\_set(size\_t index) {

buffer\_manager\_->release\_buffer\_set(index);

}

// src/strategy\_cpu/ AbstractCpuRunner.hpp

#pragma once

#include "../common/ComputeResult.hpp"

#include "../common/IDeviceRunner.hpp"

#include <chrono>

#include <cmath>

#include <cstdlib>

#include <iostream>

#include <string>

#include <vector>

/\*\*

\* @brief Classe base astratta per tutte le strategie di esecuzione su CPU.

\* Utilizza il "Template Method Pattern". Il metodo 'execute' contiene lo scheletro

\* dell'algoritmo, valido per tutte le CPU (validazione, init dati, timing, loop sui

\* task). Viene lasciato un buco (il metodo astratto 'execute\_parallel\_loop') che le

\* sottoclassi (CpuOmpRunner, CpuFfRunner) devono implementare con la loro logica di

\* parallelizzazione specifica.

\*/

class AbstractCpuRunner : public IDeviceRunner {

public:

/\*\*

\* @param kernel\_name Nome del kernel da eseguire.

\* @param runner\_tag Stringa per i log (es. "CPU OpenMP").

\*/

AbstractCpuRunner(const std::string &kernel\_name, const std::string &runner\_tag)

: kernel\_name\_(kernel\_name), runner\_tag\_(runner\_tag) {}

virtual ~AbstractCpuRunner() = default;

/\*\*

\* @brief Esegue l'intero carico di lavoro su CPU, orchestrando la logica comune a

\* tutte le strategie di esecuzione su CPU (OpenMP, FastFlow).

\*/

ComputeResult execute(size\_t N, size\_t NUM\_TASKS) override {

// Validazione del kernel.

if (kernel\_name\_ != "vecAdd" && kernel\_name\_ != "polynomial\_op" &&

kernel\_name\_ != "heavy\_compute\_kernel") {

std::cerr << "[ERROR] " << runner\_tag\_ << ": Unknown kernel name '"

<< kernel\_name\_ << "'.\n"

<< " --> Supported kernels are: 'vecAdd', 'polynomial\_op', "

"'heavy\_compute\_kernel'.\n";

exit(EXIT\_FAILURE);

}

std::cout << "[" << runner\_tag\_

<< "] Running tasks in PARALLEL on all CPU cores.\n\n";

// Inizializzazione dei dati.

a\_.resize(N);

b\_.resize(N);

c\_.resize(N);

for (size\_t i = 0; i < N; ++i) {

a\_[i] = int(i);

b\_[i] = int(2 \* i);

}

size\_t tasks\_completed = 0;

auto t0 = std::chrono::steady\_clock::now();

// Esegue NUM\_TASKS volte il calcolo parallelo in modo sequenziale.

for (size\_t task\_num = 0; task\_num < NUM\_TASKS; ++task\_num) {

std::cerr << "[" << runner\_tag\_ << " - START] Processing task " << task\_num + 1

<< " with N=" << N << "...\n";

// Chiamata al metodo che esegue il calcolo parallelo (definito nelle

// sottoclassi).

execute\_parallel\_loop(0, N);

std::cerr << "[" << runner\_tag\_ << " - END] Task " << task\_num + 1

<< " finished.\n";

tasks\_completed++;

}

// Calcolo del tempo totale e ritorno del risultato.

ComputeResult res;

auto t1 = std::chrono::steady\_clock::now();

res.elapsed\_ns =

std::chrono::duration\_cast<std::chrono::nanoseconds>(t1 - t0).count();

res.tasks\_completed = tasks\_completed;

return res;

}

protected:

/\*\*

\* @brief Metodo astratto definito dalle sottoclassi per eseguire il calcolo parallelo.

\*/

virtual void execute\_parallel\_loop(long start, long end) = 0;

/\*\*

\* @brief Logica di calcolo del kernel. Viene chiamato N volte nel loop parallelo dalle

\* sottoclassi.

\*/

void execute\_kernel\_work(long i) {

if (kernel\_name\_ == "vecAdd") {

// --------------------------------------------------------------

// SOMMA VETTORIALE

// --------------------------------------------------------------

c\_[i] = a\_[i] + b\_[i];

} else if (kernel\_name\_ == "polynomial\_op") {

// --------------------------------------------------------------

// OPERAZIONE POLINOMIALE (Calcolo 2a² + 3a³ - 4b² + 5b⁵)

// --------------------------------------------------------------

long long val\_a = a\_[i], val\_b = b\_[i];

long long a2 = val\_a \* val\_a, a3 = a2 \* val\_a;

long long b2 = val\_b \* val\_b, b4 = b2 \* b2, b5 = b4 \* val\_b;

c\_[i] = (int)((2 \* a2) + (3 \* a3) - (4 \* b2) + (5 \* b5));

} else if (kernel\_name\_ == "heavy\_compute\_kernel") {

// --------------------------------------------------------------

// COMPUTAZIONE MOLTO PESANTE (for interno e fz. trigonometriche)

// --------------------------------------------------------------

double val\_a = (double)a\_[i], val\_b = (double)b\_[i], result = 0.0;

for (int j = 0; j < 5; ++j)

result += std::sin(val\_a + j) \* std::cos(val\_b - j);

c\_[i] = (int)result;

}

}

protected:

std::vector<int> a\_, b\_, c\_; // Vettori di dati input/output

std::string kernel\_name\_;

std::string runner\_tag\_; // Device name per i log

};

// src/strategy\_cpu/ Cpu\_FF\_Runner.hpp

#pragma once

#include "../../include/ff\_includes.hpp"

#include "AbstractCpuRunner.hpp"

#include <string>

/\*\*

\* @brief Strategia concreta che esegue il calcolo su CPU utilizzando il parallel\_for di

\* FastFlow.

\*/

class Cpu\_FF\_Runner : public AbstractCpuRunner {

public:

explicit Cpu\_FF\_Runner(const std::string &kernel\_name);

virtual ~Cpu\_FF\_Runner() = default;

protected:

/\*\*

\* @brief Implementa il loop parallelo usando ff::ParallelFor.

\*/

void execute\_parallel\_loop(long start, long end) override;

private:

ff::ParallelFor pf\_;

};

// src/strategy\_cpu/ Cpu\_FF\_Runner.cpp

#include "Cpu\_FF\_Runner.hpp"

Cpu\_FF\_Runner::Cpu\_FF\_Runner(const std::string &kernel\_name)

: AbstractCpuRunner(kernel\_name, "CPU Parallel FF") {}

/\*\*

\* @brief Implementazione del loop parallelo con FastFlow. Questa funzione viene chiamata

\* dal metodo execute() della classe base AbstractCpuRunner.

\*/

void Cpu\_FF\_Runner::execute\_parallel\_loop(long start, long end) {

// Parallelizza il calcolo usando ff\_parallel\_for che gestisce il parallelismo a dati

// su CPU distribuendo le iterazioni del loop sui core disponibili.

pf\_.parallel\_for(start, end, 1, 0, [&](const long i) {

// Chiama l'helper della classe base che contiene la logica di calcolo del kernel.

this->execute\_kernel\_work(i);

});

}

// src/strategy\_cpu/ Cpu\_OMP\_Runner.hpp

#pragma once

#include "AbstractCpuRunner.hpp"

#include <string>

/\*\*

\* @brief Strategia concreta che esegue il calcolo del kernel su CPU utilizzando le

\* direttive OpenMP.

\*/

class Cpu\_OMP\_Runner : public AbstractCpuRunner {

public:

explicit Cpu\_OMP\_Runner(const std::string &kernel\_name);

virtual ~Cpu\_OMP\_Runner() = default;

protected:

/\*\*

\* @brief Implementa il loop parallelo usando la direttiva #pragma omp.

\*/

void execute\_parallel\_loop(long start, long end) override;

};

// src/strategy\_cpu/ Cpu\_OMP\_Runner.cpp

#include "Cpu\_OMP\_Runner.hpp"

#include <omp.h>

Cpu\_OMP\_Runner::Cpu\_OMP\_Runner(const std::string &kernel\_name)

: AbstractCpuRunner(kernel\_name, "CPU OpenMP") {}

/\*\*

\* @brief Implementazione del loop parallelo con OpenMP. Questa funzione viene chiamata

\* dal metodo execute() della classe base AbstractCpuRunner.

\*/

void Cpu\_OMP\_Runner::execute\_parallel\_loop(long start, long end) {

// Dice al compilatore di parallelizzare il ciclo for distribuendolo tra i thread

// disponibili.

#pragma omp parallel for

for (long i = start; i < end; ++i) {

// Chiama l'helper della classe base che contiene la logica del kernel

this->execute\_kernel\_work(i);

}

}

// src/common/Task.hpp

#pragma once

#include <chrono>

#include <cstddef>

#ifdef \_\_APPLE\_\_

#include <OpenCL/opencl.h>

#else

#include <CL/cl.h>

#endif

/\*\*

\* Struttura che rappresenta un singolo task di calcolo.

\*/

struct Task {

int \*a, \*b, \*c; // Puntatori ai vettori di input/output

size\_t n; // Dimensione dei vettori

size\_t id{0}; // ID del task

size\_t buffer\_idx{0}; // Index del buffer set che il task sta usando

// Ultimo evento OpenCL generato (usato con GPU\_openCL e FPGA).

cl\_event event{nullptr};

// Handle generico per la sincronizzazione con GPU\_Metal.

void \*sync\_handle{nullptr};

// Tempo di arrivo del task nel nodo.

std::chrono::steady\_clock::time\_point arrival\_time;

};

// src/common/BlockingQueue.hpp

#pragma once

#include <condition\_variable>

#include <mutex>

#include <queue>

/\*\*

\* @brief Coda bloccante e thread-safe per la comunicazione tra stadi.

\*

\* Mette i thread consumer a dormire quando la coda è vuota e li risveglia

\* quando un nuovo elemento è disponibile, evitando l'attesa attiva.

\*/

template <typename T> class BlockingQueue {

public:

void push(T value) {

{

std::lock\_guard<std::mutex> lock(mutex\_);

queue\_.push(std::move(value));

}

notEmptyCondition\_.notify\_one();

}

T pop() {

std::unique\_lock<std::mutex> lock(mutex\_);

notEmptyCondition\_.wait(lock, [this] { return !queue\_.empty(); });

T item = std::move(queue\_.front());

queue\_.pop();

return item;

}

private:

std::queue<T> queue\_;

std::mutex mutex\_;

std::condition\_variable notEmptyCondition\_;

};

// src/common/ComputeResult.hpp

#pragma once

#include <cstddef>

/\*\*

\* @brief Struct dati generica per i risultati di qualsiasi strategia.

\*

\* Viene restituita dal metodo 'execute' di IDeviceRunner e contiene tutte le metriche di

\* performance raccolte durante l'esecuzione, sia essa su CPU o su acceleratore. Trasporta

\* i dati grezzi dalla singola strategia (es. CPU\_OMP\_Runner, AcceleratorPipelineRunner,

\* ecc) al main.

\*/

struct ComputeResult {

long long elapsed\_ns =

0; // Tempo totale misurato dall'inizio alla fine dell'esecuzione dei task.

size\_t tasks\_completed = 0; // Numero totale di task effettivamente completati

long long computed\_ns = 0; // Tempo effettivo di calcolo del kernel

long long total\_InNode\_time\_ns =

0; // Tempo totale trascorso dai task dentro il nodo accelerato

long long inter\_completion\_time\_ns =

0; // Tempo medio tra il completamento di due task consecutivi.

};

// src/common/ PerformanceData.hpp

#pragma once

#include <cstddef>

/\*\*

\* @brief Struttura usata per contenere le metriche di performance calcolate a partire dai

\* dati raccolti in StatsCollector.

\*/

struct PerformanceData {

double avg\_service\_time\_ms = 0.0;

double avg\_InNode\_time\_ms = 0.0;

double avg\_computed\_ms = 0.0;

double avg\_overhead\_ms = 0.0;

double throughput = 0.0;

double elapsed\_s = 0.0;

};

// src/common/StatsCollector.hpp

#pragma once

#include <atomic>

#include <future>

/\*\*

\* @brief Struttura usata per raccogliere risultati generati dai 2 thread interni del nodo

\* ff\_node\_acc\_t e passarli al thread principale di FF in nella strategia di esecuzione

\* AcceleratorPipelineRunner.

\*/

struct StatsCollector {

std::atomic<size\_t> tasks\_processed{0};

std::promise<size\_t> count\_promise;

std::atomic<long long> computed\_ns{0};

std::atomic<long long> total\_InNode\_time\_ns{0};

std::atomic<long long> inter\_completion\_time\_ns{0};

};

// src/common/device\_types.hpp

#pragma once

/\*\*

\* @brief Definisce i nomi stringa costanti per i tipi di device

\* di calcolo supportati dal sistema.

\*/

namespace device {

inline constexpr const char \*CPU\_FF = "cpu\_ff";

inline constexpr const char \*CPU\_OMP = "cpu\_omp";

inline constexpr const char \*GPU\_CL = "gpu\_opencl";

inline constexpr const char \*GPU\_MTL = "gpu\_metal";

inline constexpr const char \*FPGA = "fpga";

} // namespace device

// src/helpers/Helpers.hpp

#pragma once

#include "../common/ComputeResult.hpp"

#include "../common/PerformanceData.hpp"

#include <cstddef>

#include <string>

/\*\*

\* Funzione per il parsing e il setting di default degli argomenti della riga di comando.

\*/

void parse\_args(int argc, char \*argv[], size\_t &N, size\_t &NUM\_TASKS,

std::string &device\_type, std::string &kernel\_path,

std::string &kernel\_name);

/\*\*

\* Stampa la configurazione attuale della computazione in base agli argomenti inseriti da

\* riga di comando.

\*/

void print\_configuration(size\_t N, size\_t NUM\_TASKS, const std::string &device\_type,

const std::string &kernel\_path, const std::string &kernel\_name);

/\*\*

\* Stampa le istruzioni d'uso.

\*/

void print\_usage(const char \*prog\_name);

/\*\*

\* @brief Calcola le metriche di performance finali a partire dai dati grezzi.

\*/

PerformanceData calculate\_metrics(const ComputeResult &results);

/\*\*

\* Stampa le statistiche finali del benchmark.

\*/

void print\_metrics(size\_t N, size\_t NUM\_TASKS, const std::string &device\_type,

const std::string &kernel\_name, const PerformanceData &metrics,

size\_t final\_count);

// src/helpers/Helpers.cpp

#include "Helpers.hpp"

#include "../common/device\_types.h"

#include <algorithm>

#include <iostream>

/\*\*

\* Helper interno per estrarre il nome del file da un percorso, senza

\* estensione. Usata per estrapolare il nome della funzione kernel dal kernel

\* file.

\*/

static std::string extractKernelName(const std::string &path) {

if (path.empty())

return "";

size\_t last\_slash\_pos = path.find\_last\_of("/\\");

std::string filename =

(last\_slash\_pos == std::string::npos) ? path : path.substr(last\_slash\_pos + 1);

size\_t dot\_pos = filename.find\_first\_of('.');

if (dot\_pos == std::string::npos)

return filename;

return filename.substr(0, dot\_pos);

}

/\*\*

\* Helper interno per il parsing rigoroso di un argomento numerico.

\*/

static size\_t parse\_numeric\_arg(const char \*arg\_str) {

std::string s(arg\_str);

size\_t pos = 0;

unsigned long long value = std::stoull(s, &pos, 10);

if (pos != s.length())

throw std::invalid\_argument("L'argomento '" + s +

"' contiene caratteri non numerici.");

return static\_cast<size\_t>(value);

}

/\*\*

\* Funzione per il parsing e il setting di default degli argomenti della riga di comando.

\*/

void parse\_args(int argc, char \*argv[], size\_t &N, size\_t &NUM\_TASKS,

std::string &device\_type, std::string &kernel\_path,

std::string &kernel\_name) {

N = 1000000;

NUM\_TASKS = 20;

device\_type = device::CPU\_FF;

kernel\_path = "";

kernel\_name = "";

if (argc > 1 && (std::string(argv[1]) == "-h" || std::string(argv[1]) == "--help")) {

print\_usage(argv[0]);

exit(0);

}

if (argc > 5)

std::cerr << "[WARNING] Too many arguments provided. Ignoring extras.\n";

try {

if (argc > 1)

N = parse\_numeric\_arg(argv[1]);

if (argc > 2)

NUM\_TASKS = parse\_numeric\_arg(argv[2]);

} catch (const std::invalid\_argument &e) {

std::cerr

<< "\n[ERROR] Not valid args for N or NUM\_TASKS. Integer values required.\n";

print\_usage(argv[0]);

exit(EXIT\_FAILURE);

}

if (N == 0 || NUM\_TASKS == 0) {

std::cerr << "\n[ERROR] The size of vectors (N) or the number of tasks (NUM\_TASKS) "

"cannot be 0.\n";

print\_usage(argv[0]);

exit(EXIT\_FAILURE);

}

if (argc > 3)

device\_type = argv[3];

if (argc > 4)

kernel\_path = argv[4];

// Per GPU e FPGA, se non specifico un kernel di default imposta polynomial\_op.

if (kernel\_path.empty()) {

if (device\_type == device::GPU\_CL)

kernel\_path = "kernels/gpu/polynomial\_op.cl";

else if (device\_type == device::GPU\_MTL)

kernel\_path = "kernels/gpu/polynomial\_op.metal";

else if (device\_type == device::FPGA)

kernel\_path = "kernels/fpga/krnl\_polynomial\_op.xclbin";

}

// Per GPU e FPGA, estraggo il nome del kernel dal percorso specificato.

if (device\_type == device::GPU\_CL || device\_type == device::FPGA ||

device\_type == device::GPU\_MTL)

kernel\_name = extractKernelName(kernel\_path);

// Per CPU, se non specifico un kernel imposta polynomial\_op, altrimenti lo estrae dal

// nome.

if (kernel\_path.empty() &&

(device\_type == device::CPU\_FF || device\_type == device::CPU\_OMP))

kernel\_name = "polynomial\_op";

else

kernel\_name = extractKernelName(kernel\_path);

}

/\*\*

\* Funzione per stampare la configurazione di esecuzione del programma in base agli

\* argomenti inseriti da riga di comando.

\*/

void print\_configuration(size\_t N, size\_t NUM\_TASKS, const std::string &device\_type,

const std::string &kernel\_path, const std::string &kernel\_name) {

std::cout << "\nConfiguration: N=" << N << ", NUM\_TASKS=" << NUM\_TASKS

<< ", Device=" << device\_type;

if (device\_type == "cpu\_ff" || device\_type == "cpu\_omp")

std::cout << ", Kernel=" << kernel\_name;

if (device\_type == "gpu\_opencl" || device\_type == "gpu\_metal" || device\_type == "fpga")

std::cout << ", Using " << kernel\_path;

std::cout << "\n\n";

}

/\*\*

\* Funzione per stampare le istruzioni d'uso.

\*/

void print\_usage(const char \*prog\_name) {

std::cerr

<< "\nUsage: " << prog\_name << " N NUM\_TASKS DEVICE [KERNEL]\n\n"

<< " (Gli argomenti sono posizionali e devono essere forniti in questo "

"ordine,\n gli argomenti fra [] sono opzionali)\n\n"

<< " N : Size of the vectors (default: 1,000,000)\n"

<< " NUM\_TASKS : Number of tasks to run (default: 20)\n"

<< " DEVICE : 'cpu\_ff', 'cpu\_omp', 'gpu\_opencl', 'gpu\_metal' or 'fpga' "

"(default: 'cpu\_ff').\n"

<< " KERNEL : Path to the kernel file for accelerators (.cl, .xclbin, .metal)\n"

<< " or kernel name for CPU ('vecAdd', 'polynomial\_op', etc.)\n"

<< "\nExample (GPU): " << prog\_name

<< " 16777216 100 gpu\_opencl kernels/gpu/heavy\_compute\_kernel.cl\n"

<< "Example (CPU): " << prog\_name << " 16777216 100 cpu\_ff vecAdd\n";

}

/\*\*

\* @brief Calcola le metriche di performance finali a partire dai dati grezzi.

\*/

PerformanceData calculate\_metrics(const ComputeResult &results) {

PerformanceData metrics;

if (results.tasks\_completed == 0)

return metrics;

// Tempo medio tra il completamento di due task consecutivi (in ms).

if (results.tasks\_completed > 1)

metrics.avg\_service\_time\_ms =

(results.inter\_completion\_time\_ns / (results.tasks\_completed - 1)) / 1.0e6;

// Tempo totale che la pipeline impiega per processare tutti i task (in sec).

metrics.elapsed\_s = results.elapsed\_ns / 1.0e9;

// Tempo medio per un task dall'ingresso all'uscita del nodo (in ms).

metrics.avg\_InNode\_time\_ms =

(results.total\_InNode\_time\_ns / results.tasks\_completed) / 1.0e6;

// Tempo medio del singolo calcolo sull'acceleratore, senza overhead (in ms).

metrics.avg\_computed\_ms = (results.computed\_ns / results.tasks\_completed) / 1.0e6;

// Costo medio di gestione: trasferimento dati, uso delle code, etc.

metrics.avg\_overhead\_ms = metrics.avg\_InNode\_time\_ms - metrics.avg\_computed\_ms;

// Task totali processati al secondo.

metrics.throughput =

(metrics.elapsed\_s > 0) ? (results.tasks\_completed / metrics.elapsed\_s) : 0;

return metrics;

}

/\*\*

\* Funzione per stampare le statistiche finali.

\*/

void print\_metrics(size\_t N, size\_t NUM\_TASKS, const std::string &device\_type,

const std::string &kernel\_name, const PerformanceData &metrics,

size\_t final\_count) {

if (final\_count == 0) {

std::cout << "-----------------------------------------------\n"

<< "No tasks were processed. No metrics to display.\n"

<< "-----------------------------------------------\n";

return;

}

// Trasforma in uppercase il device\_type.

std::string DEVICE\_TYPE = device\_type;

std::transform(DEVICE\_TYPE.begin(), DEVICE\_TYPE.end(), DEVICE\_TYPE.begin(),

[](unsigned char c) { return std::toupper(c); });

std::cout << "\n------------------------------------------------------------"

"------\n"

<< "PERFORMANCE METRICS on " << DEVICE\_TYPE << "\n (N=" << N

<< ", Tasks=" << final\_count;

if (device\_type == "cpu\_ff" || device\_type == "cpu\_omp") {

// Tempo medio per completare un singolo task.

double avg\_task\_time\_ms = metrics.elapsed\_s \* 1000 / final\_count;

std::cout

<< ", Kernel=" << kernel\_name

<< ")\n------------------------------------------------------------------\n"

<< "Avg Time per Task: " << avg\_task\_time\_ms << " ms/task\n"

<< " (Tempo medio per completare un singolo task in modo sequenziale)\n\n"

<< "Throughput: " << metrics.throughput << " tasks/sec\n"

<< " (Task totali processati al secondo)\n\n"

<< "Total Time Elapsed: " << metrics.elapsed\_s << " s\n"

<< "------------------------------------------------------------------\n"

<< "Tasks processed: " << final\_count << " / " << NUM\_TASKS

<< (final\_count == NUM\_TASKS ? " (SUCCESS)" : " (FAILURE)") << "\n"

<< "------------------------------------------------------------------\n";

} else

std::cout

<< ", Kernel=" << kernel\_name

<< ")\n------------------------------------------------------------------"

"\n"

<< "Avg Service Time: " << metrics.avg\_service\_time\_ms << " ms/task\n"

<< " (Tempo medio tra il completamento di due task consecutivi)\n\n"

<< "Avg In\_Node Time: " << metrics.avg\_InNode\_time\_ms << " ms/task\n"

<< " (Tempo medio per un task dall'ingresso all'uscita del nodo)\n\n"

<< "Avg Pure Compute Time: " << metrics.avg\_computed\_ms << " ms/task\n"

<< " (Tempo medio di un singolo calcolo sull'acceleratore, senza "

"overhead)\n\n"

<< "Avg Overhead Time: " << metrics.avg\_overhead\_ms << " ms/task\n"

<< " (Costo medio di gestione: trasferimento dati, uso delle code, etc.)\n\n"

<< "Throughput: " << metrics.throughput << " tasks/sec\n"

<< " (Task totali processati al secondo)\n\n"

<< "Total Time Elapsed: " << metrics.elapsed\_s << " s\n"

<< "------------------------------------------------------------------\n"

<< "Tasks processed: " << final\_count << " / " << NUM\_TASKS

<< (final\_count == NUM\_TASKS ? " (SUCCESS)" : " (FAILURE)") << "\n"

<< "------------------------------------------------------------------\n";

}

// include/ff\_includes.hpp

#pragma once

// DO NOT CHANGE ORDER OF INCLUDES

#include <ff/pipeline.hpp>

#include <ff/farm.hpp>

#include <ff/all2all.hpp>

#include <ff/graph\_utils.hpp>

#include <ff/ubuffer.hpp>

#include <ff/parallel\_for.hpp>

using namespace ff;

// kernels/gpu/vecAdd.cl

\_\_kernel void vecAdd(\_\_global const int\* a,

\_\_global const int\* b,

\_\_global int\* c,

const uint n) {

uint i = get\_global\_id(0);

if (i < n) c[i] = a[i] + b[i];

}

// kernels/gpu/vecAdd.metal

#include <metal\_stdlib>

using namespace metal;

/\*\*

\* @brief Esegue la somma elemento per elemento di due vettori.

\*

\* @param a Puntatore al primo vettore di input [attributo buffer(0)].

\* @param b Puntatore al secondo vettore di input [attributo buffer(1)].

\* @param c Puntatore al vettore di output [attributo buffer(2)].

\* @param n Il numero totale di elementi nei vettori [attributo buffer(3)].

\* @param gid L'ID globale del thread, fornito dalla GPU [attributo thread\_position\_in\_grid].

\*/

kernel void vecAdd(device const int\* a [[buffer(0)]],

device const int\* b [[buffer(1)]],

device int\* c [[buffer(2)]],

constant uint& n [[buffer(3)]],

uint gid [[thread\_position\_in\_grid]])

{

// Esegue un controllo per assicurarsi che il thread non acceda a memoria fuori dai limiti del vettore.

if (gid >= n) {

return;

}

// Esegue la somma elemento per elemento.

c[gid] = a[gid] + b[gid];

}

// kernels/gpu/polynomial\_op.cl

/\*\*

\* @brief Esegue un'operazione polinomiale complessa su due vettori di input.

\*

\* Per ogni elemento i, calcola:

\* c[i] = (2 \* a[i]^2) + (3 \* a[i]^3) - (4 \* b[i]^2) + (5 \* b[i]^5)

\*

\* @param a Puntatore al primo vettore di input in memoria globale.

\* @param b Puntatore al secondo vettore di input in memoria globale.

\* @param c Puntatore al vettore di output in memoria globale.

\* @param n Il numero totale di elementi nei vettori.

\*/

\_\_kernel void polynomial\_op(\_\_global const int\* a,

\_\_global const int\* b,

\_\_global int\* c,

const unsigned int n) {

const int i = get\_global\_id(0);

if (i < n) {

int val\_a = a[i];

int val\_b = b[i];

int a2 = val\_a \* val\_a;

int a3 = a2 \* val\_a;

int b2 = val\_b \* val\_b;

int b4 = b2 \* b2;

int b5 = b4 \* val\_b;

c[i] = (2 \* a2) + (3 \* a3) - (4 \* b2) + (5 \* b5);

}

}

// kernels/gpu/polynomial\_op.metal

#include <metal\_stdlib>

using namespace metal;

/\*\*

\* @brief Esegue un'operazione polinomiale complessa su due vettori di input.

\*

\* Versione in Metal Shading Language (MSL) del kernel.

\* Per ogni elemento i, calcola:

\* c[i] = (2 \* a[i]^2) + (3 \* a[i]^3) - (4 \* b[i]^2) + (5 \* b[i]^5)

\*

\* @param a Puntatore al primo vettore di input [attributo buffer(0)].

\* @param b Puntatore al secondo vettore di input [attributo buffer(1)].

\* @param c Puntatore al vettore di output [attributo buffer(2)].

\* @param n Il numero totale di elementi nei vettori [attributo buffer(3)].

\* @param gid L'ID globale del thread, fornito dalla GPU [attributo thread\_position\_in\_grid].

\*/

kernel void polynomial\_op(device const int\* a [[buffer(0)]],

device const int\* b [[buffer(1)]],

device int\* c [[buffer(2)]],

constant uint& n [[buffer(3)]],

uint gid [[thread\_position\_in\_grid]])

{

// Boundary check per evitare accessi a memoria non valida.

if (gid >= n) {

return;

}

// Usa 'long' (64-bit) per i calcoli intermedi per prevenire l'overflow.

long val\_a = a[gid];

long val\_b = b[gid];

// Calcolo delle potenze tramite moltiplicazioni esplicite.

long a2 = val\_a \* val\_a;

long a3 = a2 \* val\_a;

long b2 = val\_b \* val\_b;

long b4 = b2 \* b2;

long b5 = b4 \* val\_b;

// Esegue il calcolo polinomiale finale.

long result = (2 \* a2) + (3 \* a3) - (4 \* b2) + (5 \* b5);

// Assegna il risultato finale, riconvertendolo a int.

c[gid] = (int)result;

}

// kernels/gpu/heavy\_compute\_kernel.cl

/\*\*

\* @brief Esegue un calcolo computazionalmente intensivo (compute-bound).

\*

\* Per ogni elemento, esegue un ciclo di 5 iterazioni di calcoli

\* trigonometrici (sin, cos) per stressare le unità di calcolo.

\*

\* @param a Puntatore al primo vettore di input in memoria globale.

\* @param b Puntatore al secondo vettore di input in memoria globale.

\* @param c Puntatore al vettore di output in memoria globale.

\* @param n Il numero totale di elementi nei vettori.

\*/

\_\_kernel void heavy\_compute\_kernel(\_\_global const int\* a,

\_\_global const int\* b,

\_\_global int\* c,

const unsigned int n) {

const int i = get\_global\_id(0);

if (i < n) {

// Converte gli input in float per le funzioni trigonometriche

float val\_a = (float)a[i];

float val\_b = (float)b[i];

float result = 0.0f;

// Ciclo computazionalmente pesante

for (int j = 0; j < 5; ++j) {

result += sin(val\_a + j) \* cos(val\_b - j);

}

// Riconverte il risultato finale in int

c[i] = (int)result;

}

}

// kernels/gpu/heavy\_compute\_kernel.metal

#include <metal\_stdlib>

using namespace metal;

/\*\*

\* @brief Esegue un calcolo computazionalmente intensivo (compute-bound).

\*

\* Per ogni elemento, esegue un ciclo di 5 iterazioni di calcoli

\* trigonometrici (sin, cos) per stressare le unità di calcolo.

\*

\* @param a Puntatore al primo vettore di input [buffer(0)].

\* @param b Puntatore al secondo vettore di input [buffer(1)].

\* @param c Puntatore al vettore di output [buffer(2)].

\* @param n Il numero totale di elementi [buffer(3)].

\* @param gid L'ID globale del thread.

\*/

kernel void heavy\_compute\_kernel(device const int\* a [[buffer(0)]],

device const int\* b [[buffer(1)]],

device int\* c [[buffer(2)]],

constant uint& n [[buffer(3)]],

uint gid [[thread\_position\_in\_grid]])

{

if (gid >= n) {

return;

}

// Converte gli input in float per le funzioni trigonometriche

float val\_a = (float)a[gid];

float val\_b = (float)b[gid];

float result = 0.0f;

// Ciclo computazionalmente pesante

for (int j = 0; j < 5; ++j) {

result += sin(val\_a + j) \* cos(val\_b - j);

}

// Riconverte il risultato finale in int

c[gid] = (int)result;

}

// kernels/fpga/krnl\_vadd.cpp (.xclbin n.d.)

/\*\*

\* Copyright (C) 2019-2021 Xilinx, Inc

\*

\* Licensed under the Apache License, Version 2.0 (the "License"). You may

\* not use this file except in compliance with the License. A copy of the

\* License is located at

\*

\* http://www.apache.org/licenses/LICENSE-2.0

\*

\* Unless required by applicable law or agreed to in writing, software

\* distributed under the License is distributed on an "AS IS" BASIS, WITHOUT

\* WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the

\* License for the specific language governing permissions and limitations

\* under the License.

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Description:

This example uses the load/compute/store coding style which is generally

the most efficient for implementing kernels using HLS. The load and store

functions are responsible for moving data in and out of the kernel as

efficiently as possible. The core functionality is decomposed across one

of more compute functions. Whenever possible, the compute function should

pass data through HLS streams and should contain a single set of nested

loops.

HLS stream objects are used to pass data between producer and consumer

functions. Stream read and write operations have a blocking behavior which

allows consumers and producers to synchronize with each other automatically.

The dataflow pragma instructs the compiler to enable task-level pipelining.

This is required for to load/compute/store functions to execute in a

parallel and pipelined manner. Here the kernel loads, computes and stores

NUM\_WORDS integer values per clock cycle and is implemented as below:

\_\_\_\_\_\_\_\_\_\_\_\_\_

| |<----- Input Vector 1 from

Global Memory | load\_input | \_\_

|\_\_\_\_\_\_\_\_\_\_\_\_\_|----->| |

\_\_\_\_\_\_\_\_\_\_\_\_\_ | | in1\_stream

Input Vector 2 from Global Memory --->| | |\_\_|

\_\_ | load\_input | |

| |<---|\_\_\_\_\_\_\_\_\_\_\_\_\_| |

in2\_stream | | \_\_\_\_\_\_\_\_\_\_\_\_\_ |

|\_\_|--->| |<--------

| compute\_add | \_\_

|\_\_\_\_\_\_\_\_\_\_\_\_\_|---->| |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | | out\_stream

| |<---|\_\_|

| store\_result |

|\_\_\_\_\_\_\_\_\_\_\_\_\_\_|-----> Output result to

Global Memory

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Includes

#include <hls\_stream.h>

#include <stdint.h>

#define DATA\_SIZE 4096

// TRIPCOUNT identifier

const int c\_size = DATA\_SIZE;

static void load\_input(uint32\_t \*in, hls::stream<uint32\_t> &inStream,

int size) {

mem\_rd:

for (int i = 0; i < size; i++) {

#pragma HLS LOOP\_TRIPCOUNT min = c\_size max = c\_size

inStream << in[i];

}

}

static void compute\_add(hls::stream<uint32\_t> &in1\_stream,

hls::stream<uint32\_t> &in2\_stream,

hls::stream<uint32\_t> &out\_stream, int size) {

// The kernel is operating with vector of NUM\_WORDS integers. The + operator

// performs an element-wise add, resulting in NUM\_WORDS parallel additions.

execute:

for (int i = 0; i < size; i++) {

#pragma HLS LOOP\_TRIPCOUNT min = c\_size max = c\_size

out\_stream << (in1\_stream.read() + in2\_stream.read());

}

}

static void store\_result(uint32\_t \*out, hls::stream<uint32\_t> &out\_stream,

int size) {

mem\_wr:

for (int i = 0; i < size; i++) {

#pragma HLS LOOP\_TRIPCOUNT min = c\_size max = c\_size

out[i] = out\_stream.read();

}

}

extern "C" {

/\*

Vector Addition Kernel

Arguments:

in1 (input) --> Input vector 1

in2 (input) --> Input vector 2

out (output) --> Output vector

size (input) --> Number of elements in vector

\*/

void krnl\_vadd(uint32\_t \*in1, uint32\_t \*in2, uint32\_t \*out, int size) {

#pragma HLS INTERFACE m\_axi port = in1 bundle = gmem0

#pragma HLS INTERFACE m\_axi port = in2 bundle = gmem1

#pragma HLS INTERFACE m\_axi port = out bundle = gmem0

static hls::stream<uint32\_t> in1\_stream("input\_stream\_1");

static hls::stream<uint32\_t> in2\_stream("input\_stream\_2");

static hls::stream<uint32\_t> out\_stream("output\_stream");

#pragma HLS dataflow

// dataflow pragma instruct compiler to run following three APIs in parallel

load\_input(in1, in1\_stream, size);

load\_input(in2, in2\_stream, size);

compute\_add(in1\_stream, in2\_stream, out\_stream, size);

store\_result(out, out\_stream, size);

}

}

// kernels/fpga/krnl\_polynomial\_op.cpp (.xclbin n.d.)

//

// Kernel

//

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Description:

This kernel implements a complex polynomial operation using the

load/compute/store coding style for Vitis HLS. It follows a dataflow

architecture where data is streamed between three distinct stages:

1. load\_input: Reads data from global memory into streams.

2. compute\_poly: Performs the core polynomial calculation.

3. store\_result: Writes the results from a stream back to global memory.

The operation performed is:

c[i] = (2 \* a[i]^2) + (3 \* a[i]^3) - (4 \* b[i]^2) + (5 \* b[i]^5)

This structure allows for task-level pipelining, enabling the stages

to operate in parallel for maximum throughput.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <hls\_stream.h>

#include <stdint.h>

#define DATA\_SIZE 4096

// TRIPCOUNT identifier

const int c\_size = DATA\_SIZE;

/\*\*

\* @brief Reads data from global memory and writes it into an HLS stream.

\* \* @param in Pointer to the input vector in global memory.

\* @param inStream The output HLS stream.

\* @param size The number of elements to process.

\*/

static void load\_input(int32\_t \*in, hls::stream<int32\_t> &inStream, int size) {

mem\_rd:

for (int i = 0; i < size; i++) {

#pragma HLS LOOP\_TRIPCOUNT min = c\_size max = c\_size

inStream << in[i];

}

}

/\*\*

\* @brief Reads from input streams, performs the polynomial calculation, and

\* writes to an output stream.

\* \* @param in1\_stream Stream for the first input vector 'a'.

\* @param in2\_stream Stream for the second input vector 'b'.

\* @param out\_stream The output stream for the results 'c'.

\* @param size The number of elements to process.

\*/

static void compute\_poly(hls::stream<int32\_t> &in1\_stream,

hls::stream<int32\_t> &in2\_stream,

hls::stream<int32\_t> &out\_stream, int size) {

execute:

for (int i = 0; i < size; i++) {

#pragma HLS LOOP\_TRIPCOUNT min = c\_size max = c\_size

int32\_t val\_a = in1\_stream.read();

int32\_t val\_b = in2\_stream.read();

// Calculate powers using explicit multiplications for hardware

// efficiency. Use 64-bit integers for intermediate products to prevent

// overflow, as powers (especially b^5) can exceed the range of a 32-bit

// integer.

int64\_t a2 = (int64\_t)val\_a \* val\_a;

int64\_t a3 = a2 \* val\_a;

int64\_t b2 = (int64\_t)val\_b \* val\_b;

int64\_t b4 = b2 \* b2;

int64\_t b5 = b4 \* val\_b;

// Perform the final polynomial calculation using 64-bit integers

// to ensure correctness during intermediate additions/subtractions.

int64\_t result = (2 \* a2) + (3 \* a3) - (4 \* b2) + (5 \* b5);

out\_stream << (int32\_t)result;

}

}

/\*\*

\* @brief Reads data from an HLS stream and writes it to global memory.

\* \* @param out Pointer to the output vector in global memory.

\* @param out\_stream The input HLS stream.

\* @param size The number of elements to process.

\*/

static void store\_result(int32\_t \*out, hls::stream<int32\_t> &out\_stream,

int size) {

mem\_wr:

for (int i = 0; i < size; i++) {

#pragma HLS LOOP\_TRIPCOUNT min = c\_size max = c\_size

out[i] = out\_stream.read();

}

}

extern "C" {

/\*\*

\* @brief Top-level kernel function that orchestrates the dataflow pipeline.

\*

\* @param in1 (input) --> Input vector 'a'

\* @param in2 (input) --> Input vector 'b'

\* @param out (output) --> Output vector 'c'

\* @param size (input) --> Number of elements in vectors

\*/

void krnl\_polynomial\_op(int32\_t \*in1, int32\_t \*in2, int32\_t \*out, int size) {

#pragma HLS INTERFACE m\_axi port = in1 bundle = gmem0

#pragma HLS INTERFACE m\_axi port = in2 bundle = gmem1

#pragma HLS INTERFACE m\_axi port = out bundle = gmem0

static hls::stream<int32\_t> in1\_stream("input\_stream\_1");

static hls::stream<int32\_t> in2\_stream("input\_stream\_2");

static hls::stream<int32\_t> out\_stream("output\_stream");

#pragma HLS dataflow

// dataflow pragma instructs compiler to run following three APIs in parallel

load\_input(in1, in1\_stream, size);

load\_input(in2, in2\_stream, size);

compute\_poly(in1\_stream, in2\_stream, out\_stream, size);

store\_result(out, out\_stream, size);

}

}

// kernels/fpga/krnl\_heavy\_compute.cpp (.xclbin n.d.)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Description:

This kernel implements a computationally-intensive (compute-bound)

operation using the load/compute/store coding style for Vitis HLS.

It follows a dataflow architecture where data is streamed between

three distinct stages:

1. load\_input: Reads data from global memory into streams.

2. compute\_heavy: Performs the core iterative trigonometric calculation.

3. store\_result: Writes the results from a stream back to global memory.

The operation performed is a 5-iteration loop of sin/cos calculations

for each element, designed to make the kernel compute-bound.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <hls\_math.h>

#include <hls\_stream.h>

#include <stdint.h>

#define DATA\_SIZE 4096

// TRIPCOUNT identifier

const int c\_size = DATA\_SIZE;

/\*\*

\* @brief Reads data from global memory and writes it into an HLS stream.

\*/

static void load\_input(int32\_t \*in, hls::stream<int32\_t> &inStream, int size) {

mem\_rd:

for (int i = 0; i < size; i++) {

#pragma HLS LOOP\_TRIPCOUNT min = c\_size max = c\_size

#pragma HLS PIPELINE II = 1

inStream << in[i];

}

}

/\*\*

\* @brief Reads from input streams, performs the heavy trigonometric calculation,

\* and writes to an output stream.

\*/

static void compute\_heavy(hls::stream<int32\_t> &in1\_stream, hls::stream<int32\_t> &in2\_stream,

hls::stream<int32\_t> &out\_stream, int size) {

execute:

for (int i = 0; i < size; i++) {

#pragma HLS LOOP\_TRIPCOUNT min = c\_size max = c\_size

double val\_a = (double)in1\_stream.read();

double val\_b = (double)in2\_stream.read();

double result = 0.0;

// Compute-intensive loop (5 iterations) - Vitis HLS will optimize this inner loop.

compute\_loop:

for (int j = 0; j < 5; ++j) {

#pragma HLS PIPELINE // Enable pipelining for the inner loop

result += hls::sin(val\_a + j) \* hls::cos(val\_b - j);

}

out\_stream << (int32\_t)result;

}

}

/\*\*

\* @brief Reads data from an HLS stream and writes it to global memory.

\*/

static void store\_result(int32\_t \*out, hls::stream<int32\_t> &out\_stream, int size) {

mem\_wr:

for (int i = 0; i < size; i++) {

#pragma HLS LOOP\_TRIPCOUNT min = c\_size max = c\_size

#pragma HLS PIPELINE II = 1

out[i] = out\_stream.read();

}

}

extern "C" {

/\*\*

\* @brief Top-level kernel function that orchestrates the dataflow pipeline.

\*

\* @param in1 (input) --> Input vector 'a'

\* @param in2 (input) --> Input vector 'b'

\* @param out (output) --> Output vector 'c'

\* @param size (input) --> Number of elements in vectors

\*/

void krnl\_heavy\_compute(int32\_t \*in1, int32\_t \*in2, int32\_t \*out, int size) {

#pragma HLS INTERFACE m\_axi port = in1 bundle = gmem0

#pragma HLS INTERFACE m\_axi port = in2 bundle = gmem1

#pragma HLS INTERFACE m\_axi port = out bundle = gmem0

static hls::stream<int32\_t> in1\_stream("input\_stream\_1");

static hls::stream<int32\_t> in2\_stream("input\_stream\_2");

static hls::stream<int32\_t> out\_stream("output\_stream");

#pragma HLS dataflow

// dataflow pragma instruct compiler to run following three APIs in parallel.

load\_input(in1, in1\_stream, size);

load\_input(in2, in2\_stream, size);

compute\_heavy(in1\_stream, in2\_stream, out\_stream, size);

store\_result(out, out\_stream, size);

}

}

// CMakeLists.txt

cmake\_minimum\_required(VERSION 3.18)

if(APPLE)

project(Tesi LANGUAGES CXX OBJCXX)

else()

project(Tesi LANGUAGES CXX)

endif()

set(CMAKE\_CXX\_STANDARD 17)

set(CMAKE\_CXX\_STANDARD\_REQUIRED ON)

find\_package(OpenCL REQUIRED)

add\_compile\_definitions(FF\_HEADER\_ONLY NO\_DEFAULT\_MAPPING)

# ========== Clonazione di FastFlow da GitHub =======

set(FASTFLOW\_DIR "${CMAKE\_SOURCE\_DIR}/external/fastflow")

if(NOT EXISTS ${FASTFLOW\_DIR})

message(STATUS "FastFlow not found. Cloning from GitHub...")

file(MAKE\_DIRECTORY "${CMAKE\_SOURCE\_DIR}/external")

execute\_process(

COMMAND git clone --depth 1 https://github.com/fastflow/fastflow.git ${FASTFLOW\_DIR}

WORKING\_DIRECTORY ${CMAKE\_SOURCE\_DIR}/external

RESULT\_VARIABLE GIT\_RESULT

OUTPUT\_VARIABLE GIT\_OUTPUT

ERROR\_VARIABLE GIT\_ERROR

)

if(NOT GIT\_RESULT EQUAL 0)

message(FATAL\_ERROR "Errore during the clonation of FastFlow: ${GIT\_ERROR}")

else()

message(STATUS "FastFlow cloned successfully in ${FASTFLOW\_DIR}")

endif()

else()

message(STATUS "Found FastFlow: ${FASTFLOW\_DIR}")

endif()

# ===================================================

# Lista dei file sorgente comuni a tutte le piattaforme.

set(COMMON\_SOURCES

src/main.cpp

src/ff\_Pipe\_nodes/ff\_node\_acc\_t.cpp

src/factory/DeviceRunner\_Factory.cpp

src/strategy\_cpu/Cpu\_FF\_Runner.cpp

src/strategy\_accelerator/AcceleratorPipelineRunner.cpp

src/strategy\_accelerator/accelerator/BufferManager.cpp

src/helpers/Helpers.cpp

)

# Aggiunge i file sorgente e le librerie specifiche per ogni piattaforma.

if(APPLE)

list(APPEND COMMON\_SOURCES

src/strategy\_accelerator/accelerator/Gpu\_OpenCL\_Accelerator.cpp

src/strategy\_accelerator/accelerator/Gpu\_Metal\_Accelerator.mm

)

find\_library(METAL\_LIBRARY Metal REQUIRED)

find\_library(FOUNDATION\_LIBRARY Foundation REQUIRED)

set(PLATFORM\_LIBS ${METAL\_LIBRARY} ${FOUNDATION\_LIBRARY})

else()

list(APPEND COMMON\_SOURCES

src/strategy\_cpu/Cpu\_OMP\_Runner.cpp

src/strategy\_accelerator/accelerator/Fpga\_Accelerator.cpp

)

set(PLATFORM\_LIBS stdc++fs)

endif()

add\_executable(tesi-exec ${COMMON\_SOURCES})

# Specifica le directory dove il compilatore deve cercare gli .hpp

target\_include\_directories(tesi-exec PRIVATE

SYSTEM ${CMAKE\_SOURCE\_DIR}/external/fastflow

${CMAKE\_SOURCE\_DIR}/include

${OpenCL\_INCLUDE\_DIRS})

# Linka le librerie comuni e quelle specifiche della piattaforma.

target\_link\_libraries(tesi-exec PRIVATE

OpenCL::OpenCL

${PLATFORM\_LIBS}

)

# Aggiunge e linka OpenMP solo su sistemi non-Apple (Linux).

if(NOT APPLE)

find\_package(OpenMP)

if(OpenMP\_FOUND)

target\_link\_libraries(tesi-exec PRIVATE OpenMP::OpenMP\_CXX)

endif()

target\_link\_libraries(tesi-exec PRIVATE "-static-libstdc++")

endif()

target\_compile\_definitions(tesi-exec PRIVATE CL\_TARGET\_OPENCL\_VERSION=120)

target\_compile\_options(tesi-exec PRIVATE -Wno-deprecated-declarations)

# Tratta il file .mm come Objective-C++ e attiva ARC.

if(APPLE)

set\_source\_files\_properties(

src/strategy\_accelerator/accelerator/Gpu\_Metal\_Accelerator.mm PROPERTIES

LANGUAGE OBJCXX

COMPILE\_FLAGS "-fobjc-arc"

)

endif()

# Abilita AddressSanitizer per il debug della memoria.

# target\_compile\_options(tesi-exec PRIVATE -fsanitize=address -g -fno-omit-frame-pointer)

# target\_link\_options(tesi-exec PRIVATE -fsanitize=address)

// run\_banchmarks.sh

#!/bin/bash

############################################################################################################

# Script per eseguire tutti i benchmark e raccogliere i risultati in un file CSV in directory measurement/ #

############################################################################################################

OUTPUT\_DIR="measurement"

OUTPUT\_FILE="$OUTPUT\_DIR/Measurements.csv"

N\_VALUES=(10000 1000000 7449999)

NUM\_TASKS=100

# Controlla se il file eseguibile esiste. Se non esiste, avvia la build automatica.

EXECUTABLE="./build/tesi-exec"

if [ ! -f "$EXECUTABLE" ]; then

echo "--- Eseguibile non trovato: $EXECUTABLE ---"

echo "Avvio della build."

rm -rf build

cmake -B build && cmake --build build

if [ $? -ne 0 ]; then

echo "ERRORE: Compilazione fallita."

exit 1

fi

if [ ! -f "$EXECUTABLE" ]; then

echo "ERRORE: Compilazione riuscita ma l'eseguibile non è stato trovato in $EXECUTABLE."

exit 1

fi

echo "Compilazione completata con successo."

echo

echo "--- Inizio dei Benchmark ---"

echo

else

echo "--- Inizio dei Benchmark ---"

echo

fi

# Pulisce il file CSV precedente e scrive l'intestazione.

echo "OS,N,Tasks,Device,Kernel,Avg\_Service\_Time\_ms,Avg\_In\_Node\_Time\_ms,Avg\_Compute\_Time\_ms,Avg\_Overhead\_Time\_ms,Throughput\_tasks\_s,Total\_Time\_s,Status" > $OUTPUT\_FILE

# Funzione helper per eseguire un singolo test e fare il parsing dell'output.

run\_test() {

local N=$1

local DEVICE=$2

local KERNEL\_ARG=$3

local KERNEL\_NAME=$4

local OS\_NAME=$5

echo "Running: OS=$OS\_NAME, N=$N, Device=$DEVICE, Kernel=$KERNEL\_NAME"

# Esegue il comando e cattura sia stdout che stderr.

output=$( $EXECUTABLE $N $NUM\_TASKS $DEVICE $KERNEL\_ARG 2>&1 )

# Controlla se l'esecuzione è fallita.

if [ $? -ne 0 ]; then

echo "Run FAILED for $DEVICE, $KERNEL\_NAME, N=$N"

echo "$OS\_NAME,$N,$NUM\_TASKS,$DEVICE,$KERNEL\_NAME,,,,,,,FAILED" >> $OUTPUT\_FILE

echo "$output"

return

fi

# --- Parsing delle metriche dall'output ---

SERVICE\_TIME=$(echo "$output" | grep "Avg Service Time" | awk -F: '{print $2}' | awk '{print $1}')

IN\_NODE\_TIME=$(echo "$output" | grep "Avg In\_Node Time" | awk -F: '{print $2}' | awk '{print $1}')

COMPUTE\_TIME=$(echo "$output" | grep "Avg Pure Compute Time" | awk -F: '{print $2}' | awk '{print $1}')

OVERHEAD\_TIME=$(echo "$output" | grep "Avg Overhead Time" | awk -F: '{print $2}' | awk '{print $1}')

THROUGHPUT=$(echo "$output" | grep "Throughput" | awk -F: '{print $2}' | awk '{print $1}')

TOTAL\_TIME=$(echo "$output" | grep "Total Time Elapsed" | awk -F: '{print $2}' | awk '{print $1}')

# Scrive la riga CSV.

echo "$OS\_NAME,$N,$NUM\_TASKS,$DEVICE,$KERNEL\_NAME,$SERVICE\_TIME,$IN\_NODE\_TIME,$COMPUTE\_TIME,$OVERHEAD\_TIME,$THROUGHPUT,$TOTAL\_TIME,Success" >> $OUTPUT\_FILE

}

# Rileva il sistema operativo.

OS\_NAME=$(uname -s)

if [ "$OS\_NAME" == "Darwin" ]; then

# --- Comandi MacOS ---

OS\_NAME="MacOS"

DEVICES=("cpu\_ff" "gpu\_opencl" "gpu\_metal")

for N in "${N\_VALUES[@]}"; do

for DEVICE in "${DEVICES[@]}"; do

if [ "$DEVICE" == "cpu\_ff" ]; then

run\_test $N "cpu\_ff" "vecAdd" "vecAdd" $OS\_NAME

run\_test $N "cpu\_ff" "polynomial\_op" "polynomial\_op" $OS\_NAME

run\_test $N "cpu\_ff" "heavy\_compute\_kernel" "heavy\_compute\_kernel" $OS\_NAME

elif [ "$DEVICE" == "gpu\_opencl" ]; then

run\_test $N "gpu\_opencl" "kernels/gpu/vecAdd.cl" "vecAdd" $OS\_NAME

run\_test $N "gpu\_opencl" "kernels/gpu/polynomial\_op.cl" "polynomial\_op" $OS\_NAME

run\_test $N "gpu\_opencl" "kernels/gpu/heavy\_compute\_kernel.cl" "heavy\_compute\_kernel" $OS\_NAME

elif [ "$DEVICE" == "gpu\_metal" ]; then

run\_test $N "gpu\_metal" "kernels/gpu/vecAdd.metal" "vecAdd" $OS\_NAME

run\_test $N "gpu\_metal" "kernels/gpu/polynomial\_op.metal" "polynomial\_op" $OS\_NAME

run\_test $N "gpu\_metal" "kernels/gpu/heavy\_compute\_kernel.metal" "heavy\_compute\_kernel" $OS\_NAME

fi

done

done

elif [ "$OS\_NAME" == "Linux" ]; then

# --- Comandi Linux VM ---

OS\_NAME="Linux"

DEVICES=("cpu\_ff" "cpu\_omp" "fpga")

for N in "${N\_VALUES[@]}"; do

for DEVICE in "${DEVICES[@]}"; do

if [ "$DEVICE" == "cpu\_ff" ]; then

run\_test $N "cpu\_ff" "vecAdd" "vecAdd" $OS\_NAME

run\_test $N "cpu\_ff" "polynomial\_op" "polynomial\_op" $OS\_NAME

run\_test $N "cpu\_ff" "heavy\_compute\_kernel" "heavy\_compute\_kernel" $OS\_NAME

elif [ "$DEVICE" == "cpu\_omp" ]; then

run\_test $N "cpu\_omp" "vecAdd" "vecAdd" $OS\_NAME

run\_test $N "cpu\_omp" "polynomial\_op" "polynomial\_op" $OS\_NAME

run\_test $N "cpu\_omp" "heavy\_compute\_kernel" "heavy\_compute\_kernel" $OS\_NAME

elif [ "$DEVICE" == "fpga" ]; then

run\_test $N "fpga" "kernels/fpga/krnl\_vadd.xclbin" "vecAdd" $OS\_NAME

run\_test $N "fpga" "kernels/fpga/krnl\_polynomial\_op.xclbin" "polynomial\_op" $OS\_NAME

run\_test $N "fpga" "kernels/fpga/krnl\_heavy\_compute.xclbin" "heavy\_compute\_kernel" $OS\_NAME

fi

done

done

else

echo "Sistema operativo non supportato: $OS\_NAME"

exit 1

fi

echo "----------------------------------------"

echo "Benchmark completati."

echo "Risultati salvati in: $OUTPUT\_FILE"

echo "----------------------------------------"

// README.md

# Progettazione di un Nodo FastFlow per l'Integrazione di Acceleratori Hardware

Questo progetto di tesi esplora l'integrazione efficiente di acceleratori hardware eterogenei (GPU e FPGA) all'interno del framework di streaming parallelo \*\*FastFlow\*\*.

Il core del progetto è un nuovo nodo FastFlow (\*\*ff\_node\_acc\_t\*\*) progettato per gestire l'offloading asincrono di task computazionali su dispositivi eterogenei, nascondendo la

latenza dei task dietro una pipeline di lavoro.

---

## Architettura

Il progetto adotta un'architettura software modulare basata sui design pattern \*\*Strategy\*\*, \*\*Factory\*\* e \*\*Adapter\*\*, per garantire flessibilità e manutenibilità.

### Strategie di Esecuzione (IDeviceRunner)

Un'interfaccia comune permette di eseguire lo stesso carico di lavoro su backend diversi senza modificare il codice client.

- \*\*CPU\*\*: Strategie per l'esecuzione parallela su CPU multicore (`Cpu\_OMP\_Runner`, `Cpu\_FF\_Runner`).

- \*\*Acceleratori\*\*: Un adattatore (`AcceleratorPipelineRunner`) incapsula una pipeline FastFlow specializzata per l'offloading.

### Pipeline di Offloading

Il nodo `ff\_node\_acc\_t` implementa internamente una pipeline \*Producer–Consumer\* asincrona per massimizzare il throughput e sovrapporre la comunicazione \*Host-to-Device\* con il calcolo.

### Astrazione Hardware (IAccelerator)

Un'interfaccia unificata astrae le differenze tra le API di basso livello:

- OpenCL (FPGA / GPU NVIDIA / AMD)

- Metal (Apple Silicon)

---

## Requisiti

### Dipendenze Software

- \*\*CMake\*\* (≥ 3.18)

- Compilatore C++ con supporto \*\*C++17\*\*

- \*\*FastFlow\*\*: scaricato automaticamente da CMake

- \*\*OpenCL\*\*: per FPGA e GPU Apple M2 Pro

- \*\*OpenMP\*\*: richiesto per la strategia `cpu\_omp` su Linux Xeon

- \*\*Metal\*\*: richiesto per la strategia `gpu\_metal` su MacOS

---

## Setup Sperimentale

Le misurazioni sono state eseguite su due host di calcolo distinti:

1. \*\*MacBook M2 Pro (ambiente di sviluppo)\*\*

- Architettura: ARM64

- CPU: Apple M2 Pro, 10 core

- GPU: integrata Apple M2 Pro, 16 unità di calcolo

- Sistema operativo: macOS Sonoma 15.6.1

2. \*\*Host Linux\*\*

- CPU: Intel Xeon E5-2650 v3 @ 2.30 GHz, 20 core fisici, 40 thread logici

- Sistema operativo: Ubuntu 22.04.5 LTS

- Acceleratore: scheda Xilinx Alveo U50 (xilinx\_u50\_gen3x16\_xdma\_base\_5), connessa tramite bus PCIe

## Stack Software e Versioni

### Compilatore

- \*\*MacOS:\*\* Clang versione 20.1.7 [20] (Target: x86\_64-apple-darwin24.6.0)

- \*\*Linux:\*\* GCC 15.1.0 [21]

### Librerie e Toolchain

- \*\*FastFlow:\*\* versione 3.0.0 [10]

Utilizzata per implementare la pipeline sui nodi e per il confronto prestazionale su CPU Apple M2 Pro e CPU Intel.

- \*\*OpenMP:\*\* versione 4.5 [15]

Utilizzata per il confronto prestazionale su CPU Linux.

- \*\*Toolchain Vitis:\*\* versione v2023.1 [8]

Utilizzata per la Sintesi ad Alto Livello su FPGA.

- \*\*API OpenCL:\*\* versione 1.2 [9]

Utilizzata per l'interfacciamento con GPU Apple M2 Pro e FPGA.

- \*\*Driver XRT:\*\* versione 2.16.204 [19]

Driver di runtime Xilinx.

---

## Compilazione

Dalla directory principale del progetto:

```bash

rm -rf build; cmake -B build && cmake --build build

```

## Esecuzione

L'eseguibile ```tesi-exec``` accetta i seguenti parametri posizionali:

```bash

./build/tesi-exec <N> <NUM\_TASKS> <DEVICE\_TYPE> <KERNEL\_ARG>

```

Parametri

- N: dimensione del problema (numero elementi)

- NUM\_TASKS: numero di task da eseguire

- DEVICE\_TYPE (backend supportati):

- cpu\_ff

- cpu\_omp

- gpu\_opencl

- gpu\_metal

- fpga

- KERNEL\_ARG:

- CPU → nome kernel (vecAdd, polynomial\_op, heavy\_compute\_kernel)

- GPU/FPGA → percorso file .cl, .metal, .xclbin

<br>

Esempi

CPU (FastFlow):

```bash

./build/tesi-exec 1000000 100 cpu\_ff polynomial\_op

```

GPU (Metal - macOS):

```bash

./build/tesi-exec 1000000 100 gpu\_metal kernels/gpu/heavy\_compute\_kernel.metal

```

FPGA (OpenCL - Linux):

```bash

./build/tesi-exec 1000000 100 fpga kernels/fpga/krnl\_vadd.xclbin

```

## Benchmark Automatizzati

Lo script incluso automatizza una suite di benchmark sui kernel disponibli e genera un CSV finale in /measurements:

```bash

chmod +x run\_benchmarks.sh

./run\_benchmarks.sh

```

// .clang\_format

BasedOnStyle: LLVM

IndentWidth: 3

ContinuationIndentWidth: 3

BreakBeforeBraces: Custom

BraceWrapping:

IndentBraces: false

UseTab: Never

ColumnLimit: 90

Ringraziamenti

.