POLITECNICO DI TORINO

Laurea Magistrale in Ingegneria Informatica

Studio comparativo tra Cuda e Vulkan in ambito GPGPU

Relatore

Prof. Giovanni MALNATI

Candidato

Gabriele BELLUARDO

Anno Accademico
2023/2024

Sommario

L'elaborato di tesi si propone l'obiettivo di confrontare soluzioni basate su CUDA con soluzioni omologhe basate su Vulkan e Rust, per la realizzazione di applicativi GPGPU in un contesto orientato ai microservizi, e, in particolare, per accelerare la risoluzione di problemi di ottimizzazione sfruttando l'elevato parallelismo delle GPU. Viene posto il focus sulla comparazione delle performance, della *development experience*, e della presenza, o meno, di tool per lo sviluppo e il debug.

L'approccio proposto, basato su Vulkan e Rust, si pone come un'alternativa moderna, performante e portabile per lo sviluppo di applicativi su GPU, mantenendo alto il livello di astrazione dall'hardware e garantendo un'esecuzione del programma *memory safe*. Queste proprietà derivano dalle caratteristiche intrinseche degli strumenti sopra citati e dell'ecosistema su cui si basano: fornite quindi senza costo, non richiedendo allo sviluppatore particolari accortezze o revisioni aggiuntive.

Nella prima parte dell'elaborato, vengono presentati i concetti fondamentali della GPGPU. Vengono introdotte le API CUDA e Vulkan e l'adozione da parte dell'industria software. Segue una breve introduzione al linguaggio Rust e ai vantaggi che porta in termini di usabilità e produttività per lo sviluppatore, oltre che per l'utente finale.

Nella seconda parte, viene illustrata la suite di benchmark creati per comparare le performance e i risultati ottenuti. Viene, infine, fornita un'analisi dell'esperienza di sviluppo, la trattazione dei punti di forza e debolezza delle due soluzioni e l'elenco dei contesti in cui è più indicato usare l'una o l'altra soluzione.

Nella terza parte, infine, viene illustrato lo sviluppo di un risolutore di matrici QUBO, che integra i benefici di entrambi gli approcci. L'applicativo è nella forma di un microservizio, per facilitare la comunicazione *machine-to-machine*, e sfrutta, per la parte di rete, il linguaggio Rust con il framework async 'Axum', e le API CUDA, per la parte di GPU computing. La scelta di adottare questo tipo di soluzione, come si vedrà, ha portato vantaggi sia in termini di velocità sviluppo che di prestazioni. Inoltre, grazie all'ottimo supporto della FFI di Rust ai linguaggi C/C++, l'integrazione con CUDA è stata agevole, al netto di un preliminare passo di configurazione.

keywords: GPGPU, CUDA, Rust, Vulkan, Microservice, REST API, FFI

Contenuti

Elenco delle figure

Elenco delle tabelle

Listings

Capitolo 1

Introduzione

1.1 Programmazione su GPU

Sin dalla nascita delle prime CPU, l'obiettivo principale degli ingegneri è stato quello di aumentane velocità ed efficienza. Sono state sviluppate diverse tecniche per aumentare le prestazioni: come le *CPU pipelined*, che permettono di aumentare il throughput delle istruzioni, la *multilevel cache*, per nascondere la latenza della memoria principale rispetto alla velocità di clock del processore, l'*instruction level parallelism* e processori superscalari, per eseguire più istruzioni durante un singolo ciclo di clock. Quando gli ingegneri si sono accorti che queste metodologie non sarebbero bastate per tenere viva la legge di Moore [1], si è passati ad un approccio *multi-core*, che ha portato a una nuova forma di parallelismo: parti dell'applicazione possono essere eseguite in contemporanea sui diversi core della stessa CPU. L'impatto delle CPU multi-core, per accelerare le applicazioni, è stato considerevole, ma il numero limitato di core che possono essere presenti sulla singola unità CPU è uno dei maggiori ostacoli che tutt'oggi limita l'ulteriore aumento di prestazioni. In fig. 1.1 è mostrato come, grazie alla tecnologia multi-core, la legge di Moore può essere ancora considerata valida.

Le GPU sono speciali processori, la cui architettura prende a piene mani il concetto di multi-core e lo esalta, al punto da avere tantissimi core, più semplici e lenti dei core CPU, ma con un più alto throughput totale. Originariamente le GPU sono state sviluppate per scopi legati all'elaborazione grafica, in particolare, per migliorare la resa delle immagini e la grafica 3D nei videogiochi, nelle applicazioni CAD e nei software di modellazione e rendering. In seguito, le esigenze di una grafica sempre più dettagliata e complessa ha portato alla creazione di unità di elaborazione specializzate, con un'architettura che le rende energicamente più efficienti di una CPU, per algoritmi che processano grossi blocchi di dati in parallelo. Le GPU sono state quindi fruttate anche per scopi diversi dall'elaborazione grafica. Si sono rivelate

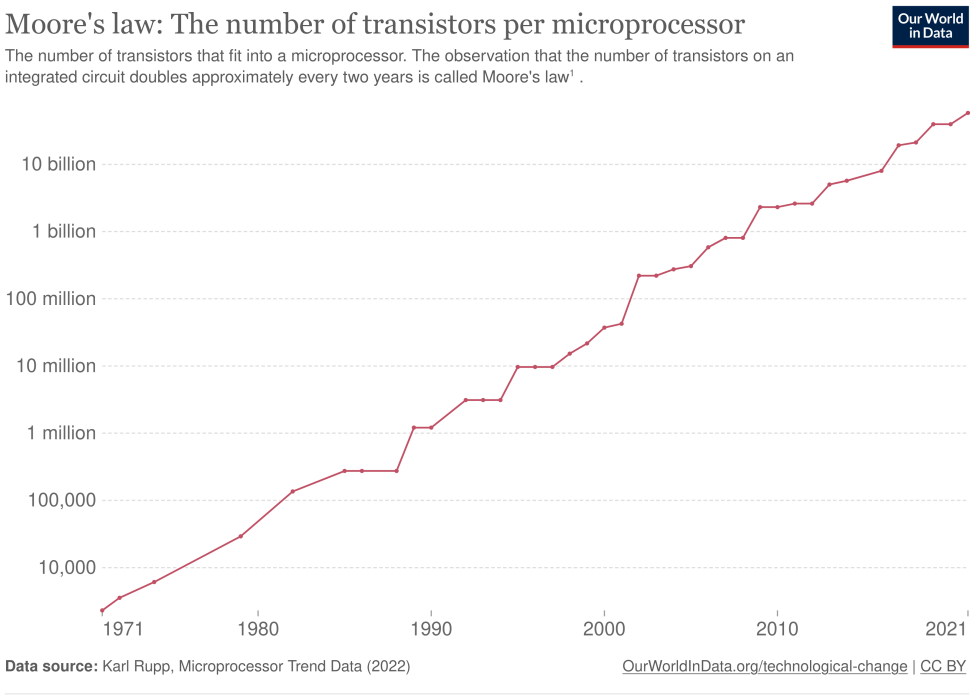

Figura 1.1: Legge di Moore [fonte: ourworldindata.org]

particolarmente utili per applicazioni scientifiche, HPC e tecniche che richiedono elaborazione intensive, come la simulazione di modelli numerici, l'analisi dei dati e il calcolo scientifico. In tempi recenti, le GPU vengono impiegate per l'addestramento e l'esecuzione di reti neurali, diventando uno strumento fondamentale per lo sviluppo dell'intelligenza artificiale.

Quando si parla di GPGPU si intende l'utilizzo delle GPU per compiti di calcolo generale, rendendole molto più che semplici dispositivi per l'elaborazione grafica. Questo è fondamentale per accelerare algoritmi di calcolo che richiedono enorme quantità di risorse, a patto che la computazione sia parallelizzabile su più processori. Calcoli che prima richiedevano l'uso di supercomputer, adesso si possono eseguire con una normale GPU desktop. Nel 2011, secondo la lista Top500, che classifica e descrive nel dettaglio i 500 sistemi informatici non distribuiti più potenti al mondo, il Tianhe-1A risultava il secondo supercomputer più potente al mondo. Grazie a un sistema basato sulle GPU NVIDIA Tesla M2050 [2] il Tianhe-1A è riuscito a raggiungere i 4.7 petaFLOPS di potenza computazionale. Solamente un anno dopo, invece, in vetta alla classifica spiccava il Titan [3], con un sistema basato sulle più performanti GPU NVIDIA Tesla K20X, raggiungendo i 17.59 petaFLOPS di potenza. Da quell'anno divenne chiaro che le GPU sarebbero diventate una componente essenziale nell'HPC tanto quanto lo erano nel desktop computing.

Dato che il supercomputing è il motore trainante di molte delle tecnologie che vediamo nei processori moderni, si è venuto a creare un circolo virtuoso: la necessità di processori sempre più veloci, per elaborare dataset sempre più grandi, porta l'industria a produrre supercomputer sempre più potenti, tramite i quali si possono progettare processori migliori. Ad oggi, si sta delineando una divisione netta nella produzione di GPU per uso desktop e per uso scientifico, i principali produttori, quali AMD, NVIDIA e Intel rilasciano prodotti specificatamente per l'uno o per l'altro segmento di mercato, con lo scopo di soddisfare i diversi requisiti di ogni settore. Da giugno 2023 il supercomputer più veloce al mondo è il Frontier, e con un sistema basato sulle GPU Radeon Instinct MI250X riesce a raggiungere i 1.67 exaFLOPS, divenendo il primo exascale supercomputer al mondo [4]. Grazie all'uso delle GPU, si riescono a incrementare le performance dei supercomputer in modo esponenziale, come si può notare in fig. 1.2. L'incremento così repentino delle performance ha portato gli esperti di settore a coniare una nuova legge empirica: la legge di Huang [5], da Jensen Huang, CEO e co-founder di NVIDIA. La legge enuncia che le performance delle GPU "più che raddoppiano ogni due anni". In pratica, la legge di Huang è analoga alla legge di Moore, ma applicata alle GPU.

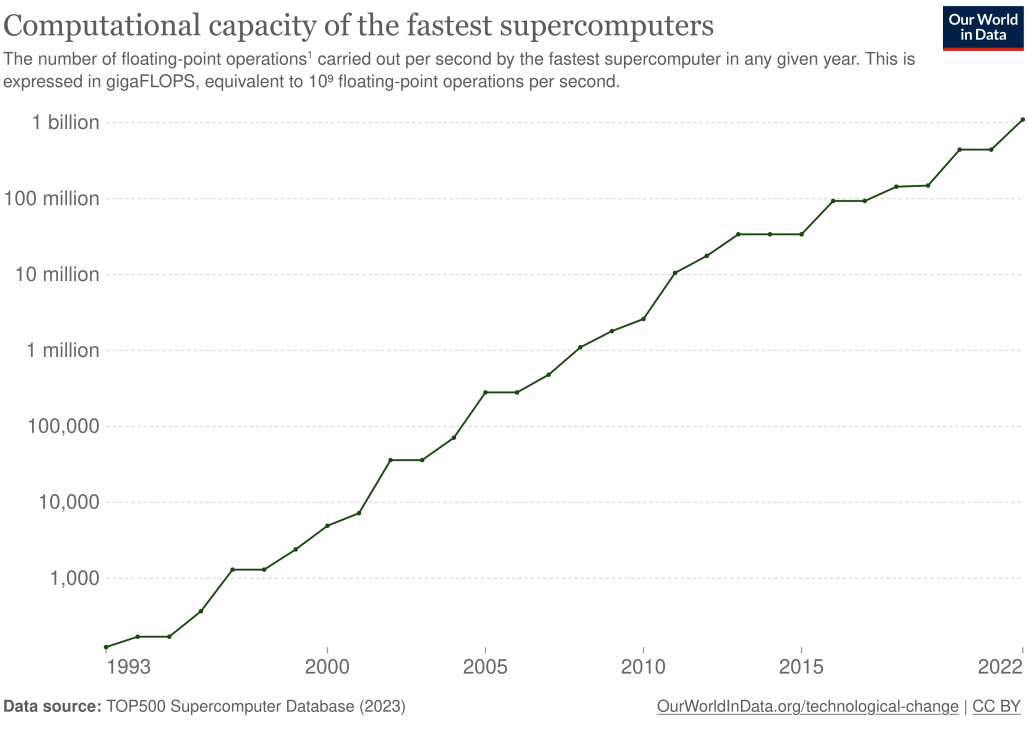

Figura 1.2: Potenza dei supercomputer negli anni [fonte: ourworldindata.org]

Per poter programmare le GPU è stato necessario sviluppare delle API che garantissero l'esecuzione del programma senza un'esplicita conversione dei dati in formato grafico. Le API oggi più utilizzate sono le CUDA API di NVIDIA [6] e le OpenCL API di Khronos Group [7]. Le API CUDA sfruttano l'omonima architettura delle GPU NVIDIA e sono in formato proprietario, mentre le API OpenCL, come suggerisce il nome, sono uno standard aperto *royalty-free* e supportano la maggior parte delle architetture GPU esistenti. Sebbene queste abbiano prestazioni di molto inferiori rispetto a CUDA, OpenCL è la prima scelta per lo sviluppo di applicazioni multi-piattaforma, proprio per la sua natura open e l'ampio supporto hardware. Nel 2015 Khronos Group annuncia le Vulkan API [8] e il formato SPIR-V [9]. Vulkan è una API grafica e di calcolo che offre un accesso ad alta efficienza e multi-piattaforma alle GPU e CPU moderne. Viene utilizzato in una vasta gamma di dispositivi, da PC e console, a telefoni cellulari e piattaforme embedded. Vulkan è stato progettato per sfruttare pesantemente il *multithreading*, permettendo la generazione di carichi di lavoro asincroni da parte di thread multipli della CPU, che eseguono il codice sulla GPU solo dopo una esplicita sottomissione. Inoltre, la sua natura *close-to-metal* permette un controllo oculato delle risorse della GPU: lo sviluppatore è responsabile della sincronizzazione, allocazione di memoria e della sottomissione del lavoro, il che risulta in un minore overhead del driver rispetto a OpenCL. SPIR-V è un formato di rappresentazione intermedia per shader, che permette di scrivere il codice una volta e poi compilarlo per diverse architetture, sia GPU che CPU. Vulkan e SPIR-V sono stati sviluppati per essere usati insieme, ma non sono strettamente legati, infatti SPIR-V può essere usato anche con OpenCL e OpenGL.

1.2 Problema e Contributi

L'obiettivo di questo lavoro di tesi è valutare i potenziali vantaggi derivanti dall'utilizzo di Vulkan e Rust per lo sviluppo di applicazioni di computing, rispetto all'approccio tradizionale basato su CUDA. Nonostante Vulkan sia largamente usato nell'industria dei videogiochi, usarlo in ambito computing è una sfida non banale, a causa delle API verbose e del controllo meticoloso sulle strutture dati. Un'idea per ridurre il carico intellettivo per lo sviluppatore, e rendere la *development experience* più fluida e appagante, può essere quella gestire la memoria in modo automatico. Solitamente, quando si parla di gestione automatica della memoria, si fa riferimento a *garbage collector* o contatori di riferimenti. Per quanto valido sia questo approccio, il risultato non sempre è performante e ad alta efficienza, requisito fondamentale in ambito computing. Per mantenere le prestazioni quanto più vicine possibile a un'implementazione *low level*, senza sacrificare l'ergonomicità di un approccio ad alto livello, si è scelto di coniugare i vantaggi del linguaggio

Rust, quali, appunto, la gestione automatica della memoria senza strumenti di garbage collections, con quelli che offre Vulkan.

Per poter capire se Vulkan e Rust sono una valida alternativa a CUDA per il computing, è necessario rispondere alle seguenti domande:

- Qual è il livello di performance che Vulkan riesce a raggiungere rispetto a CUDA?
- Quanto è difficile sviluppare un'applicazione per il computing in Vulkan?
- In un ambiente a microservizi, quanto è facile sviluppare e mantenere un'applicazione Vulkan scritta in Rust?
- Qual è il livello di maturità dell'ecosistema Vulkan? E quello di Rust?

Per rispondere a queste domande si è agito nel seguente modo:

- Si è scelto di implementare algoritmi altamente parallelizzabili su GPU, quali la somma di vettori e moltiplicazione di matrici, da usare come banco di prova per comparare le performance di implementazioni in CUDA e Vulkan. In particolare, si è preso in considerazione sia il tempo dell'esecuzione dell'algoritmo, che il tempo di trasferimento dei dati in memoria.
- Si è implementato gli algoritmi in Vulkan per valutare il potenziale delle sue API *close-to-metal* e quanto sia ergonomico sviluppare in Rust. Dato che sviluppare in Vulkan richiede degli step di inizializzazione, quali la creazione di un *logical device*, una *pipeline*, la compilazione del codice dello shader e l'allocazione di memoria, si è usata la libreria Rust Vulkano [10], che implementa delle interfacce *safe* all'implementazione standard di Vulkan.
- Si è poi reimplementato i medesimi algoritmi in CUDA, per comparare sia le performance che il diverso approccio alla scrittura dei kernel, come estensione del linguaggio C++ tramite il compilatore NVCC [11], osservando che GLSL [12] usato in Vulkan offre più funzioni orientate alla grafica che al computing.
- Infine, considerando l'esperienza di sviluppo e i risultati relativi ai benchmark ottenuti, si è sviluppato un microservizio *proof of concept*, che coniuga i vantaggi di Rust in ambito web con quelli di CUDA in ambito computing. Si è inoltre valutato anche altri possibili approcci alla soluzione del problema.

1.3 Struttura della tesi

L'elaborato è strutturato nei seguenti capitoli:

- **Capitolo 1**: Introduzione e storia delle GPU e del HPC.
- **Capitolo 2**: Introduzione alla GPGPU e alle API di programmazione, con particolare attenzione a Vulkan e CUDA, e una breve descrizione di Rust.
- **Capitolo 3**: Descrizione del problema, e analisi dei requisiti del progetto finale, con valutazione dei possibili approcci al problema con relativi pro e contro.
- **Capitolo 4**: Descrizione dell'architettura usata per i benchmark e esposizione dei dati raccolti.
- **Capitolo 5**: Analisi dei dati raccolti dai benchmark e considerazioni sullo sviluppo degli applicativi.
- **Capitolo 6**: Implementazione del microservizio e valutazione delle possibili implementazioni future.
- **Capitolo 7**: Conclusioni e considerazioni finali.

Il lavoro è stato progettato, sviluppato e testato con il supporto del team di Quantum Computing di Data Reply con sede a Torino.

Capitolo 2

Background Tecnico

L'obiettivo di questo capitolo è presentare i concetti fondamentali dei sistemi eterogenei e della programmazione GPU, con particolare attenzione ai framework CUDA e Vulkan. Ottenere una visione approfondita di tali concetti contribuirà a una migliore la comprensione delle scelte tecnologiche e dei dettagli implementativi discussi nel capitolo del progetto finale.

Il capitolo è diviso in due sezioni: la prima si concentra sul modello di programmazione GPGPU, mentre la seconda fornirà un'analisi dettagliata di CUDA e Vulkan per lo sviluppo di software e verrà giustificata la scelta del linguaggio Rust per la programmazione con Vulkan.

2.1 General-purpose computing su GPU

Sebbene CPU e GPU siano entrambi processori che eseguono istruzioni macchina, differiscono nell'approccio usato per eseguire programmi paralleli: mentre le CPU *multi-core* possono adottare un approccio di tipo MIMD o SIMD, le GPU seguono un modello di istruzioni di tipo SIMT, in cui il carico di lavoro è suddiviso in sotto-problemi risolvibili da singoli thread indipendenti. Lo sviluppo delle GPU si è storicamente concentrato nell'ambito delle industria dei videogiochi, il cui impegno tecnologico è sempre stato finalizzato a offrire esperienze virtuali il più realistiche possibili. Per raggiungere questo obiettivo, sono richiesti un numero considerevole di calcoli geometrici (tipicamente rotazioni e traslazioni) per ogni pixel dello schermo, sotto forma di operazioni in virgola mobile. Le architetture delle GPU sono strutturate seguendo il modello *many-thread* anziché *multi-core*, mettendo quindi l'accento sull'ottimizzazione del trattamento dei dati piuttosto che sulla memorizzazione e sul controllo del flusso. Entrambi i modelli sono rappresentati in fig. 2.1.

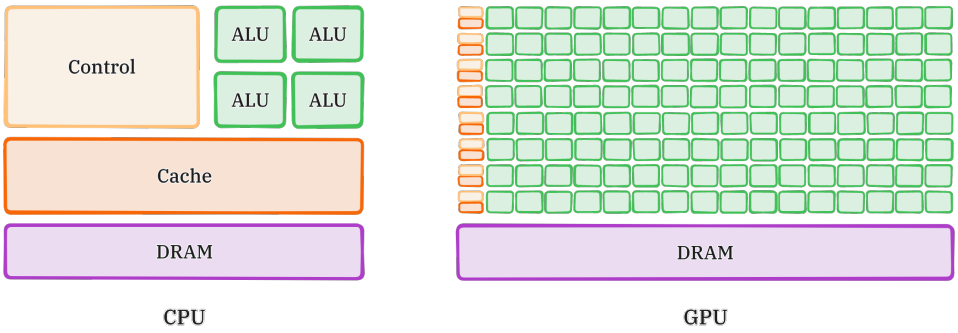

Figura 2.1: Confronto fra l'architettura CPU e GPU

Il paradigma di GPGPU consente di sfruttare la potenza di calcolo delle GPU non solo per la grafica dei videogiochi, ma anche per eseguire calcoli generici di natura scientifica ed ingegneristica, come l'ottimizzazione nella simulazione di sistemi fisici reali. Grazie alla GPGPU, il trasferimento di dati tra CPU e GPU diventa bidirezionale e, di conseguenza, in sistemi che richiedono numerose operazioni su grandi quantità di dati, si può osservare un notevole miglioramento delle prestazioni. A un'implementazione intelligente del paradigma SIMD su GPU può conseguire un aumento di velocità fino a 100 volte rispetto a un'implementazione sequenziale su un singolo core CPU. Secondo la tassonomia di Flynn [13], un sistema è classificato come SIMD se esegue una singola istruzione, o anche un piccolo insieme di istruzioni, in parallelo su un vasto numero di dati. Il paradigma SIMT è simile, e affida l'esecuzione di un ristretto insieme di istruzioni a un singolo thread, sarà poi compito del programmatore aggregare i risultati di ogni thread GPU e ottenere il risultato finale della computazione.

In genere, un sistema non può essere parallelizzato in tutte le sue parti, limitando l'accelerazione delle prestazioni a una piccola porzione del codice. Pertanto, è necessario progettare sistemi in cui, alcune parti sono eseguite in parallelo, mentre altre in modo sequenziale. Per questo motivo, nel modello GPGPU, CPU e GPU collaborano tra loro, dando origine a un modello di elaborazione eterogenea in cui la CPU gestisce la parte sequenziale, mente la GPU esegue in parallelo la parte computazionalmente onerosa. La fig. 2.2 mostra il modello di un sistema eterogeneo: il carico di lavoro parallelizzabile, è affidato alla GPU, mentre il compito di gestire i processi e la suddivisione in sotto-problemi è affidata alla CPU.

Inizialmente, le GPU e i linguaggi di programmazione parallela erano destinati a un mercato molto diverso rispetto a quello delle CPU. Nella programmazione CPU, la compatibilità tra diverse versioni dello stesso software è stata fin dall'inizio un requisito fondamentale. L'innovazione per migliorare le prestazioni GPU, invece, ha spesso comportato cambiamenti drastici nell'hardware. Queste evoluzioni

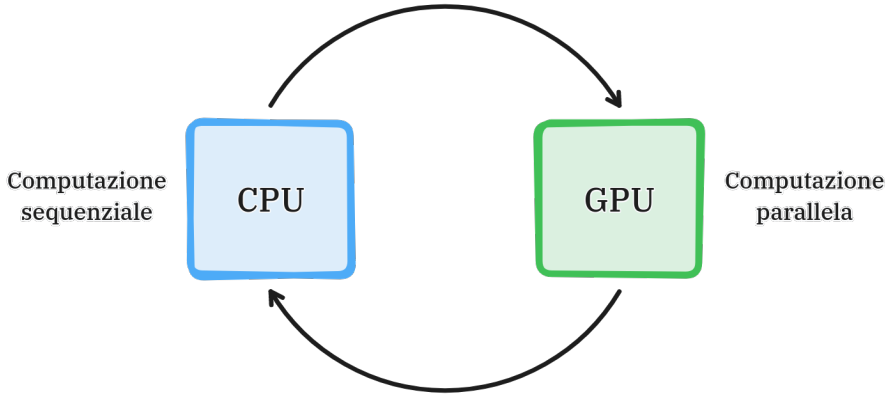

Figura 2.2: Modello di un sistema eterogeneo

tecnologiche hanno portato a una perdita di portabilità tra i diversi modelli: l'introduzione di nuove tecnologie hardware ha portato a proposte di architetture GPU completamente nuove, con significative differenze tra loro. Di conseguenza, queste nuove architetture richiedevano quasi sempre una ridefinizione completa dei relativi codici. Si è rilevato quindi necessario stabilire degli standard operativi per poter, quantomeno, mantenere un modello logico coerente tra le diverse architetture. I principali standard per l'elaborazione parallela sono MPI, OpenMP.

- **MPI**: progettato per sistemi di memoria distribuita, utilizzato in ambienti di HPC in cui più processori o nodi comunicano scambiandosi messaggi. Solitamente viene usato per l'architettura di supercomputer, nei quali migliaia di nodi sono connessi attraverso una rete dedicata. Ogni problema viene suddiviso in diversi sotto-problemi, ognuno dei quali è risolto da un nodo specifico. Questo modello è poco flessibile e oneroso di risorse e soprattutto necessita di una rete dedicata molto veloce che si possa far carico dei messaggi tra i nodi.
- **OpenMP**: offre un insieme di direttive del compilatore e routine di libreria per il *multiprocessing* su singole macchine a memoria condivisa, comunemente usato per parallelizzare cicli e altre regioni di codice per sfruttare i processori *multi-core*. Dato che il software deve essere eseguito sulla stessa macchina questo modello è relativamente più semplice del precedente, ma limitato dal punto di vista prestazionale.

Il modello OpenMP consente al programmatore di raggiungere un alto livello di parallelizzazione, specificando semplicemente le porzioni specifiche del codice da ottimizzare. D'altra parte, in MPI i nodi non condividono la memoria e tutti

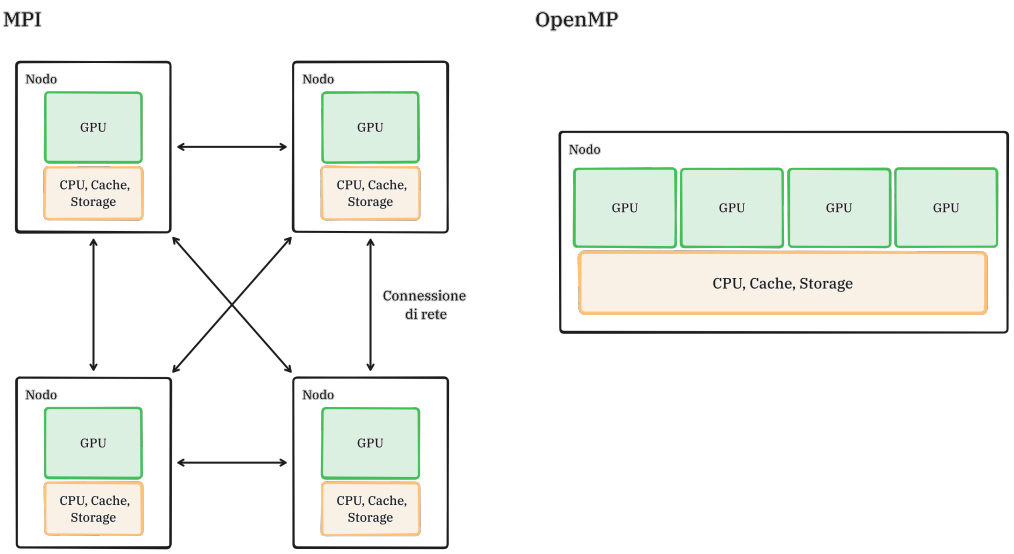

Figura 2.3: Sistema MPI e OpenMP a confronto

i dati sono condivisi attraverso messaggi, garantendo elevata scalabilità anche su centinaia di miglia di nodi; implementarlo, però, può risultare difficile proprio per la mancanza di memoria condivisa tra i nodi. A causa della loro natura intrinsecamente diversa, è raro utilizzare contemporaneamente questi standard. In fig. 2.3 è mostrata la sostanziale differenza tra i due modelli.

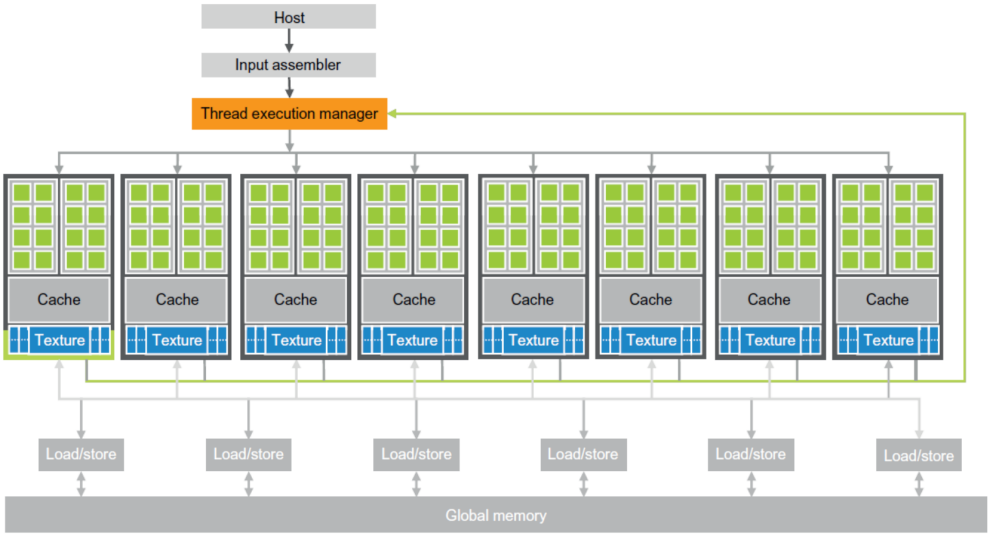

Figura 2.4: Architettura di una GPU CUDA-compatibile. Fonte: [14] pag. 20

Con l'introduzione di CUDA, NVIDIA ha fornito agli sviluppatori un framework in grado di combinare i vantaggi dei due standard. Con l'acronimo CUDA, che sta per *Compute Unified Device Architecture*, si intende non solo il linguaggio di programmazione, ma anche l'architettura hardware. Per sfruttare questo framework, è necessaria una GPU compatibile con CUDA e, dato che è sviluppato da NVIDIA, tutte le loro GPU recenti lo supportano. L'architettura tipica di una GPU NVIDIA compatibile con CUDA è illustrata in fig. 2.4. Le differenze tra i vari modelli possono essere negli *Streaming Multiprocessors* e nei *Stream Processors*, ma non ci si addentrerà ulteriormente in questi concetti. Ogni GPU recente è dotata di gigabyte di memoria dedicata GDDR, un tipo di memoria SDRAM, usata per memorizzare temporaneamente informazioni necessarie alla computazione, evitando di doverle recuperare nella memoria centrale del sistema, operazione che richiederebbe tempo e risorse elevati.

2.2 Framework di programmazione su GPU

Con rilascio di CUDA, nel 2007, NVIDIA commercializza le prime GPU che riservano aree specifiche di silicio per semplificare la programmazione parallela. Nei chip G80 e nei loro successori dedicati al calcolo parallelo, i programmi CUDA non attraversano più l'interfaccia grafica, ma una nuova interfaccia di programmazione parallela ad uso generale, che gestisce le richieste. Questa interfaccia espande notevolmente i tipi di applicazioni che è possibile sviluppare per le GPU. Inoltre, anche tutti gli altri strati software sono stati ridisegnati, consentendo ai programmatori di utilizzare i familiari strumenti di programmazione C/C++. Sebbene sia ancora possibile usare la vecchia interfaccia di programmazione basata su OpenGL, l'avvento di CUDA e il supporto al computing da parte di NVIDIA ha reso molto più semplice sviluppare applicazioni parallele, eliminando la necessità di utilizzare API grafiche e di esprimere i calcoli sotto forma di funzioni che colorano pixel.

Solamente un anno dopo Khronos Group, un consorzio industriale nato da Apple, che gestisce lo sviluppo e la promozione di standard aperti, rilascia OpenCL. OpenCL fornisce una piattaforma aperta e standardizzata per la programmazione parallela su sistemi eterogenei. Così come CUDA, anche OpenCL mira a fornire una piattaforma unificata per lo sviluppo di applicazioni che possono sfruttare le capacità di elaborazione parallela, ma, a differenza di CUDA, OpenCL supporta varie architetture hardware e non ha bisogno di chip dedicati. Un aspetto fondamentale di OpenCL è la sua natura di standard aperto e multi-piattaforma: ciò significa che gli sviluppatori possono scrivere codice OpenCL che può essere eseguito su più dispositivi, indipendentemente dal produttore. OpenCL ha subito diverse revisioni nel corso degli anni per migliorare le funzionalità e la flessibilità.

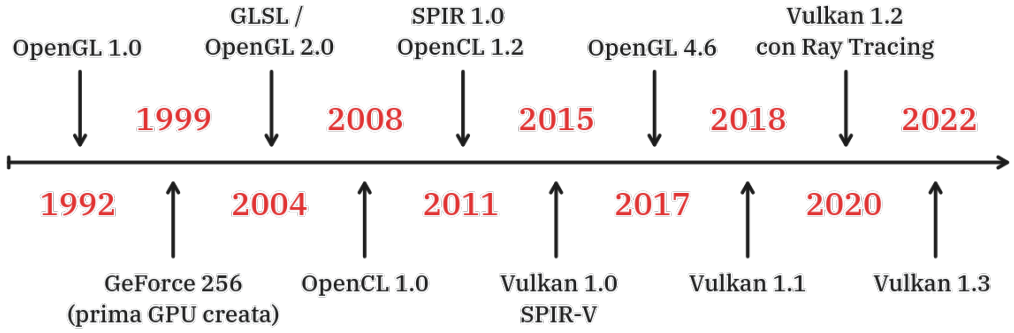

Figura 2.5: Contributi di Khronos Group

Col tempo, Khronos Group ha riconosciuto la necessità di unire le caratteristiche di grafica e calcolo in un'unica API. Vulkan è stata sviluppata con l'obiettivo di superare le limitazioni di OpenGL e di fornire un accesso più diretto alle risorse hardware, consentendo una maggiore parallelizzazione e un maggiore controllo da parte degli sviluppatori. Vulkan offre una serie di funzionalità che lo rendono più efficiente e flessibile rispetto ad OpenGL e OpenCL, inclusa una migliore gestione delle risorse, un controllo più diretto sulla GPU, una maggiore parallelizzazione e un minore overhead dei driver. Vulkan è progettata per essere più adatta agli sviluppatori che richiedono il massimo delle prestazioni all'hardware, come nei videogiochi e nelle applicazioni di realtà virtuale. In sintesi, Vulkan rappresenta una diretta evoluzione di OpenGL e OpenCL, cercando di unire le migliori caratteristiche di entrambe per fornire un'API più potente, efficiente e adatta alle moderne architetture hardware. In fig. 2.5 è visibile l'evoluzione, nel tempo, delle API rilasciate da Khronos Group. Sia OpenGL che OpenCL sopravvivono tutt'oggi per la loro semplicità di sviluppo e per l'ampio supporto hardware, sebbene siano meno performanti rispetto a CUDA o Vulkan.

2.2.1 CUDA

La struttura di un programma CUDA riflette la coesistenza, nella stessa macchina, di un *host*, rappresentato dalla CPU, e uno o più *device* GPU. Ogni file sorgente CUDA può contenere sia codice *host* che codice *device*. Da questo punto di vista, si può considerare un qualsiasi programma C/C++ come un programma CUDA che contiene solo codice *host*. Le dichiarazioni di funzioni o dati per il *device* sono chiaramente contrassegnate con particolari keyword di CUDA. Il codice deve quindi

essere compilato da un compilatore che riconosce e comprende queste dichiarazioni aggiuntive: il compilatore CUDA ufficiale è NVCC prodotto da NVIDIA.

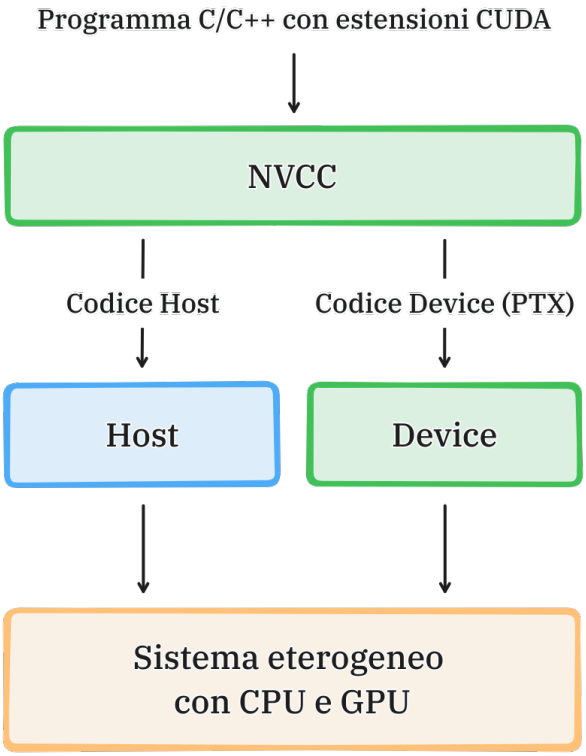

Figura 2.6: Processo di compilazione di un programma CUDA

NVCC elabora il programma, utilizzando le keyword CUDA per separare il codice *host* e il codice *device*. Il codice *host* è normale codice ANSI C, che viene ulteriormente compilato dai compilatori standard C/C++ dell'*host* e viene eseguito come un processo CPU tradizionale. Il codice *device* è contrassegnato con le keyword CUDA `__device__` o `__global__` per etichettare le funzioni di parallelismo dei dati, chiamate kernel, e le relative strutture di dati associate. Il codice *device* viene ulteriormente compilato da un componente runtime di NVCC ed eseguito su un *device* GPU. L'esecuzione inizia sull'*host* e quando una funzione kernel viene chiamata, essa viene eseguita parallelamente dai thread del *device*. L'insieme di thread generati da un avvio del kernel viene chiamato *blocco*, l'insieme dei blocchi è chiamato *griglia*.

Listing 2.1: Somma di vettori in CUDA

```
__global__ void kernelFn(int *vec_a, int *vec_b, int *res, uint n) {
  uint i = threadIdx.x + blockDim.x * blockIdx.x;

  if (i < n) {
    res[i] = vec_a[i] + vec_b[i];
  }
}

int main(int argc, char **argv) {
  uint size = len * sizeof(int);
  int *h_vec_a = reinterpret_cast<int *>(malloc(size));
  int *h_vec_b = reinterpret_cast<int *>(malloc(size));
  int *h_res_vec = reinterpret_cast<int *>(malloc(size));

  int *vec_a, *vec_b, *res_vec;

  cudaMalloc(&vec_a, size);
  cudaMemcpy(vec_a, h_vec_a, size, cudaMemcpyHostToDevice);
  cudaMalloc(&vec_b, size);
  cudaMemcpy(vec_b, h_vec_b, size, cudaMemcpyHostToDevice);

  cudaMalloc(&res_vec, size);

  kernelFn<<<ceil(len / 64.0), 64>>>(vec_a, vec_b, res_vec, len);
  cudaDeviceSynchronize();

  cudaMemcpy(h_res_vec, res_vec, size, cudaMemcpyDeviceToHost);

  ...

  cudaFree(res_vec);
  cudaFree(vec_b);
  cudaFree(vec_a);
  free(h_res_vec);
  free(h_vec_b);
  free(h_vec_a);

  return 0;
}
```

Il codice in 2.1 mostra un programma CUDA che somma due vettori in parallelo su GPU. La keyword `__global__` indica che la funzione è un kernel CUDA che verrà eseguita in `ceil(len / 64.0)` blocchi da `64` thread ciascuno, nella griglia della GPU. I vettori vengono creati nella memoria *host*, copiati nella memoria *device*, elaborati e il risultato della computazione salvato nel vettore `res_vec` in memoria *device*, che dovrà essere ricopiato in memoria *host* per potervi accedere

successivamente dal codice *host*. Le funzioni `cudaMalloc`, `cudaFree` e `cudaMemcpy` sono le equivalenti di `malloc`, `free` e `memcpy` presenti in C, e vengono usate per gestire l'allocazione di memoria *device* e la copia di memoria tra *host* e *device* e viceversa.

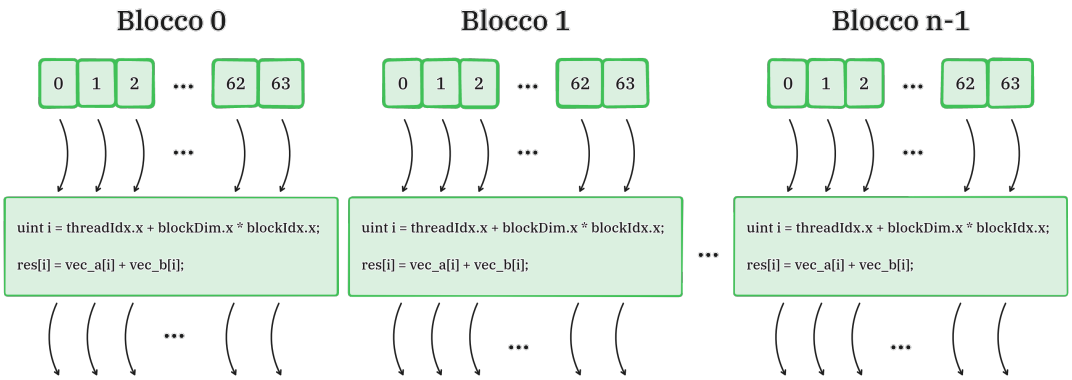

Figura 2.7: Esecuzione di programma CUDA

Il codice kernel viene eseguito parallelamente da ogni thread della GPU seguendo il paradigma SPMD, che differisce dal SIMD per l'assenza del vincolo che ogni istruzione debba essere eseguita in contemporanea su ogni thread. Il thread esegue tre operazioni: lettura dei dati dai vettori di input, somma dai dati e scrittura del dato sul vettore di output. Ogni thread esegue queste stesse identiche istruzioni, la differenza sta nelle aree di memoria su cui esse vengono eseguite: `uint i = threadIdx.x + blockDim.x * blockIdx.x` indica l'indice del vettore, e quindi l'aera di memoria sui cui lavorare, partendo dai parametri `threadIdx`, `blockDim` e `blockIdx`. Come illustrato in fig 2.7 la griglia sarà quindi composta da `n` blocchi da 64 thread ciascuno, il numero effettivo dei blocchi dipende dalla lunghezza `len` dei vettori, poiché si è deciso che 64 thread fissi devono essere presenti in ogni blocco.

Listing 2.2: Kernel CUDA di somma di vettori

```
__global__ void kernelFn(int *vec_a, int *vec_b, int *res, uint n) {
  uint i = threadIdx.x + blockDim.x * blockIdx.x;

  if (i < n) {
    res[i] = vec_a[i] + vec_b[i];
  }
}
```

Il codice in 2.2 è schematizzato in fig. 2.8. Supponiamo adesso che la dimensione dei vettori sia 132, non multiplo di 64 (`blockDim`). Il blocco condizionale `if (i<n) {...}` indica che i thread nel blocco con `blockIdx=2`, che si occuperanno di sommare gli elementi con indici `i=128,129,130,131`, devono eseguire le operazioni solo se, appunto, `i` è minore della lunghezza massima del vettore `n`. Sebbene per questo specifico esempio non sia necessario al fine della corretta esecuzione del programma, è buona prassi aggiungere sempre questo tipo di condizione sui dati per evitare letture o scritture su aree di memoria non allocate.

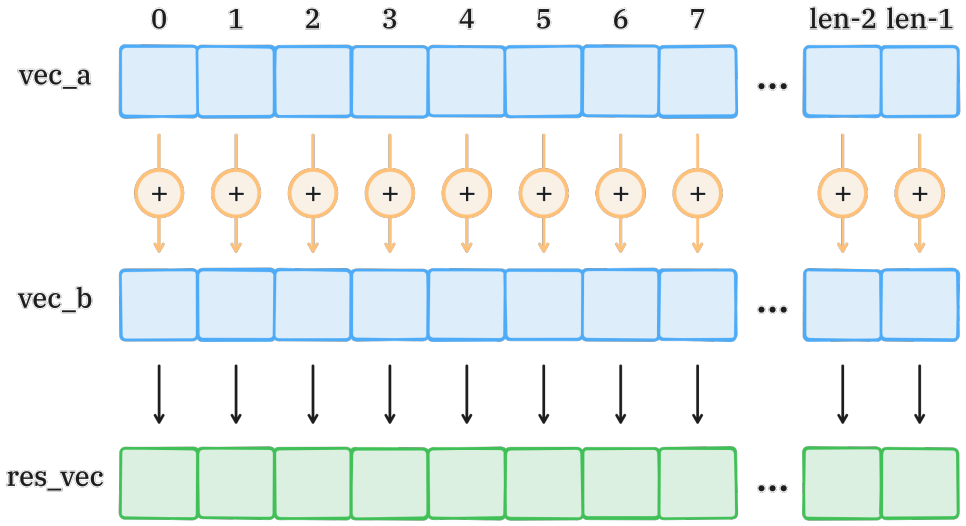

Figura 2.8: Somma parallela di vettori

Come si è illustrato, in poco più di 30 righe di codice si è in grado di accelerare, in CUDA, una semplice operazione di somma di vettori. Nella prossima sezione si illustrerà lo stesso programma, ma utilizzando le API Vulkan tramite il linguaggio di programmazione Rust.

2.2.2 Vulkan

Vulkan è una rivoluzionaria API per grafica e calcolo 3D ad alte prestazioni progettata per le moderne architetture a pipeline delle GPU, mantenendo il supporto multi-piattaforma tipico di OpenGL. Nonostante sia il successore dell'API OpenGL e OpenCL, è stato adottato un approccio completamente nuovo, per soddisfare le esigenze degli sviluppatori e lavorare in modo più efficiente con l'hardware della GPU. Vulkan è un'API esplicita, in grado di controllare le impostazioni dell'hardware per sfruttare al massimo la potenza del calcolo parallelo, sia in ambito grafico che di computing. Il *driver layer* è minimale e pone maggiori responsabilità

sul programmatore dell'applicazione, che deve gestire le risorse, la gestione della memoria, la sincronizzazione, oltre all'applicazione stessa. Questa natura esplicita di Vulkan lo rende particolarmente verboso, soprattutto se confrontato a CUDA.

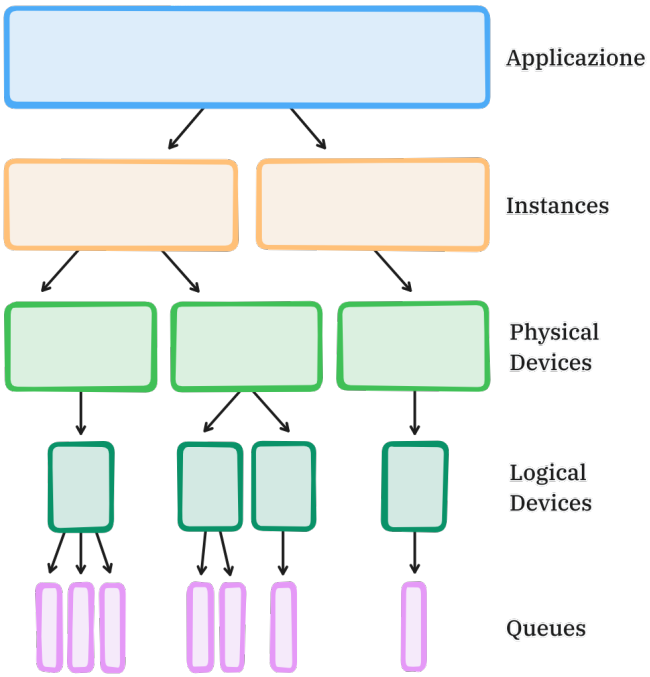

Figura 2.9: Schema di un'applicazione Vulkan

In fig. 2.9 è mostrato lo schema logico di un'applicazione Vulkan. Per scrivere un'applicazione Vulkan, per prima cosa va creata una *instance*, che ha il compito di inizializzare la libreria Vulkan installata nel sistema e tenere traccia dello stato dell'applicazione. Una volta completata questa fase, si recuperano le informazioni sui *physical device* disponibili nel sistema: si possono selezionare uno o più *device*. Per *physical device* si intende l'hardware (CPU o GPU) su cui l'applicazione può essere eseguita. Se il nostro sistema dispone di molti *physical device*, dato che ogni *device* ha determinate proprietà, si selezioneranno solo i *device* adatti all'applicazione. Un *physical device* ha molte *queue*, che sono categorizzate in gruppi chiamati *queue family*, in base alle loro proprietà. Ogni *queue family* supporta una o più operazioni, ad esempio grafiche, di calcolo o di trasferimento di memoria. Poiché lo scopo di questo elaborato è sviluppare un'applicazione per il computing, verrà utilizzato un *device* appartenente alle *queue family computing*, che quindi supporta operazioni di calcolo. Una volta selezionato il *device* è necessario creare un *logical device*, astrazione logica che rappresenta le connessioni al *physical device* ed

è usato dall'applicazione per interagire con l'hardware del dispositivo. Si possono anche creare più *logical device* per lo stesso *physical device*. Come ultima cosa, deve essere creata una *queue*, della famiglia che ci interessa, a cui verrà sottoposto il carico di lavoro.

Sebbene le API Vulkan rilasciate da Khronos Group siano scritte in C++, esistono vari implementazioni per altri linguaggi. Come si è già accennato, in questo lavoro verrà usata la libreria Vulkano in Rust, in particolare la sua versione *v0.33*. Vulkano offre dei bindings all'implementazione C++ ufficiale, ed espone tutte le strutture dati necessarie per scrivere un'applicazione Vulkan implementando il pattern RAII [15] e gestendo i riferimenti in memoria in safe Rust. Questo approccio permette di concentrarsi maggiormente sulla logica del programma piuttosto che sulla gestione di riferimenti in memoria, che è affidata al borrow checker [16] del compilatore di Rust. I benefici dell'usare Rust piuttosto che un altro linguaggio di programmazione verranno illustrati nella prossima sezione.

Listing 2.3: Inizializzazione di Vulkan in Rust

```
fn main() -> Result<()> {
    let instance = Instance::new(
        VulkanLibrary::new()?,
        InstanceCreateInfo::application_from_cargo_toml(),
    )?;
    let device_extensions = DeviceExtensions {
        khr_storage_buffer_storage_class: true,
        khr_shader_float_controls: true,
        nv_compute_shader_derivatives: true,
        ..DeviceExtensions::empty()
    };
    let device_features = Features {
        shader_float64: true,
        ..Features::empty()
    };

    let (physical_device, queue_family_index) = instance
        .enumerate_physical_devices()?
        .filter(|p| {
            p.supported_extensions().contains(&device_extensions)
                && p.supported_features().contains(&device_features)
        })
        .filter_map(|p| {
            p.queue_family_properties()
                .iter()
                .position(|q| q.queue_flags
                                .contains(QueueFlags::COMPUTE))
                .map(|i| (p, i as u32))
        })
```

```
30          .min_by_key(|(p, _)| match p.properties().device_type {
31              PhysicalDeviceType::DiscreteGpu => 0,
32              PhysicalDeviceType::IntegratedGpu => 1,
33              PhysicalDeviceType::VirtualGpu => 2,
34              PhysicalDeviceType::Cpu => 3,
35              PhysicalDeviceType::Other => 4,
36              _ => 5,
37          })
38          .ok_or(anyhow!("No suitable physical device detected"));
39
40      let (device, mut queues) = Device::new(
41          physical_device,
42          DeviceCreateInfo {
43              enabled_features: device_features,
44              enabled_extensions: device_extensions,
45              queue_create_infos: vec![QueueCreateInfo {
46                  queue_family_index,
47                  ..Default::default()
48              }],
49              ..Default::default()
50          },
51      )?;
52
53      let queue = queues.next().unwrap();
54
55      ...
56  }
```

L'inizializzazione dell'applicazione Vulkan è illustrata in 2.3. Le variabili **`device`** e **`queue`** rappresentano, rispettivamente, il *logical device* e la *queue* su cui sottomettere il lavoro.

In Vulkan il codice eseguito dalla GPU viene chiamato *shader*, ed è scritto in GLSL. Il linguaggio GLSL nasce come linguaggio di shading di OpenGL ed ha una sintassi simile al C e CUDA. Come illustrato in 2.4 i vettori di input, il vettore risultate e il parametro **`n`**, che indica la lunghezza dei vettori, sono passati in modo esplicito prima dell'esecuzione della funzione **`main`**. Importante notare che per ogni vettore vi è un parametro **`binding`**, che, come si vedrà in seguito, indica la zona di memoria che viene passata alla GPU. Nella funzione **`main`** possiamo notare similitudini con il kernel CUDA di 2.2, unica differenza degna di nota è il parametro che indica l'indice del vettore su cui eseguire le operazioni, in questo caso indicato con **`uint i = gl_GlobalInvocationID.x`** piuttosto che sommando e moltiplicando manualmente indice di blocco e di thread, come avviene con CUDA.

Listing 2.4: Shader GLSL di somma di vettori

```
#version 450

layout(local_size_x = 64, local_size_y = 1, local_size_z = 1) in;

layout(set = 0, binding = 0) readonly buffer VecA {
    int data[];
} vec_a;

layout(set = 0, binding = 1) readonly buffer VecB {
    int data[];
} vec_b;

layout(set = 0, binding = 2) writeonly buffer VecRes {
    int data[];
} res;

layout(push_constant) uniform PushConstants {
    uint n;
};

void main() {
    uint i = gl_GlobalInvocationID.x;

    if (i < n){
        res.data[i] = vec_a.data[i] + vec_b.data[i];
    }
}
```

Il codice GLSL deve poi essere compilato in SPIR-V e infine caricato in memoria dall'applicazione Vulkan, con altre informazioni sui dettagli sull'esecuzione del codice. Questo meccanismo è illustrato in fig. 2.10. Passare per una rappresentazione binaria intermedia quale SPIR-V permette a Vulkan di avere supporto multi-piattaforma, lasciando ai driver del sistema del *device* i dettagli implementati su come eseguire il codice generato.

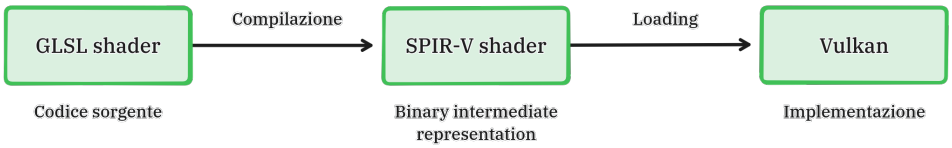

Figura 2.10: Generazione di uno shader SPIR-V e binding nell'applicazione Vulkan

Per compilare e linkare lo shader, la liberia Vulkano fornisce delle macro, illustrate in 2.5, che rendono tutto il processo automatico, agendo in fase di compilazione dell'eseguibile finale.

Listing 2.5: Compilazione e binding dello shader GLSL

```
pub mod vec_sum {
    vulkano_shaders::shader! {
        ty: "compute",
        path: "vec_sum.glsl"
    }
}
```

Una volta che Vulkan è inizializzata e lo shader è pronto per essere compilato, rimane solo allocare le risorse per l'implementazione del programma. Prima di addentrasi nei dettagli implementativi serve capire come Vulkan alloca e gestisce la memoria.

In Vulkan, considereremo due tipi di memoria: *host memory* e *device memory* schematizzate in fig. 2.11. La principale differenza tra le due è che la memoria *host* è accessibile e utilizzata dalla CPU, mentre la memoria device è accessibile solo dal dispositivo in cui viene eseguito lo shader. La memoria del dispositivo può avere alcune proprietà e tipi [17]. Ad esempio, un tipo di memoria può essere `HOST_VISIBLE`, che specifica che la memoria può essere mappata nello spazio di memoria dell'*host* e a cui si può accedere come se fosse memoria RAM; oppure può essere `DEVICE_LOCAL` che indica che la memoria è, appunto, mappata nella memoria *device*. L'applicazione è responsabile dell'allocazione della memoria con il tipo appropriato in base alle esigenze. Vulkan offre questa flessibilità per migliorare le prestazioni e l'utilizzo della memoria [18].

Nella libreria Vulkano, per indicare l'uso che si farà della memoria, si utilizzano gli `enum`: i primi due buffer infatti vengono marcati come `MemoryUsage::Upload` poiché vengo inizializzati nell'*host* e trasferiti in memoria *device*, mentre il terzo buffer viene marcato come `MemoryUsage::Download` perché, appunto, viene scritto dal device e letto dall'*host*. Il risultato è che tutti i buffer sono visibili sia dall'*host* che dal *device*, ma, mentre i primi due sono scrivibili solamente dall'*host*, il terzo è scrivibile solo dal *device*. Queste politiche di accesso in memoria non solo permettono di isolare meglio i componenti ed evitare errori di programmazione, ma permettono anche ottimizzazioni da parte del compilatore SPIR-V per un'esecuzione più performante da parte del driver GPU.

Nello shader in 2.4 i tre buffer vengono inizializzati con parametri che rispecchiano la natura e l'uso della memoria dalla prospettiva del *device*: infatti, i primi due vengono indicati col parametro `readonly`, mentre il terzo con `writeonly`.

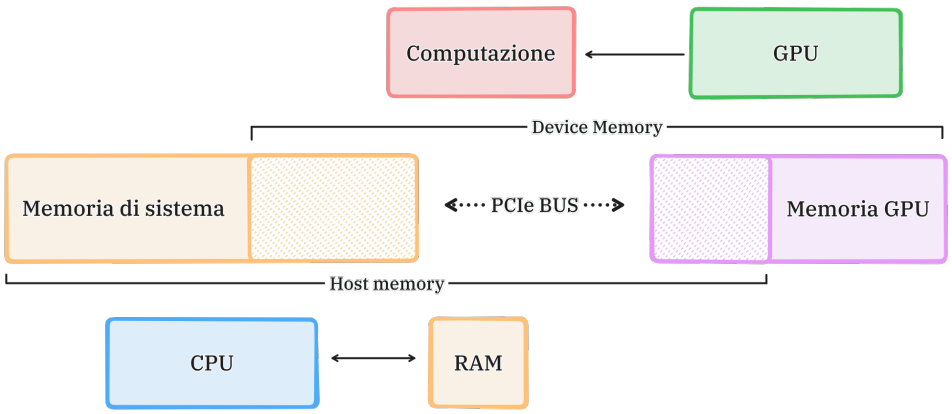

Figura 2.11: Memoria Host e Device

Listing 2.6: Inizializzazione buffer Vulkan

```
let vec_a = Vec::new();
let vec_b = Vec::new();

let memory_allocator = StandardMemoryAllocator::new_default(device.
    clone());
let command_buffer_allocator =
    StandardCommandBufferAllocator::new(device.clone(), Default::
    default());

let buf_a = Buffer::from_iter(
    &memory_allocator,
    BufferCreateInfo {
        usage: BufferUsage::STORAGE_BUFFER,
        ..Default::default()
    },
    AllocationCreateInfo {
        usage: MemoryUsage::Upload,
        ..Default::default()
    },
    vec_a.iter(),
);
let buf_b = Buffer::from_iter(
    &memory_allocator,
    BufferCreateInfo {
        usage: BufferUsage::STORAGE_BUFFER,
        ..Default::default()
    },
    AllocationCreateInfo {
        usage: MemoryUsage::Upload,
```

```
        ..Default::default()
    },
    vec_b.iter(),
);
let buf_res = Buffer::new_slice(
    &memory_allocator,
    BufferCreateInfo {
        usage: BufferUsage::STORAGE_BUFFER,
        ..Default::default()
    },
    AllocationCreateInfo {
        usage: MemoryUsage::Download,
        ..Default::default()
    },
    len as u64,
)?;
```

Marcare tutti i tre vettori come `HOST_VISIBLE` è necessario, poiché i primi due di input sono inizializzati dalla CPU in memoria *host*, mentre l'ultimo viene letto dalla CPU per conoscere il risultato dell'operazione. In caso di computazioni intermedie, in cui non è necessario conoscere il risultato dell'operazione preliminare, ma soltanto passare il buffer a un altro shader per la computazione successiva, allora avrebbe senso marcare l'area di memoria come `DEVICE_LOCAL` (utilizzando `MemoryUsage::DeviceOnly`) evitando così overhead da eventuali accessi in memoria di sistema.

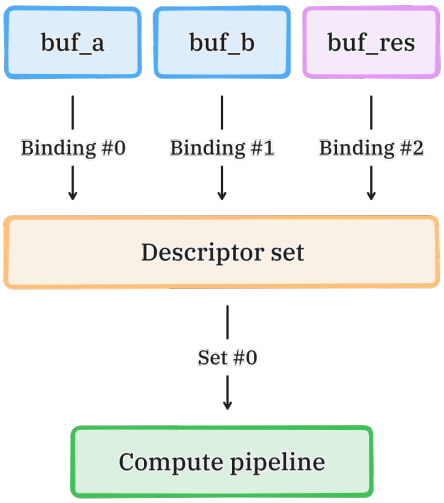

Figura 2.12: Descriptor set e compute pipeline

Dopo aver inizializzato i buffer bisogna creare dei *descriptor set* che servono a specificare e gestire i dati di input e output per il kernel di calcolo. Questi vengono passati alla *compute pipeline*, che definisce il flusso di operazioni parallele che vengono eseguite sui dati durante il processo di calcolo. In fig. 2.12 è illustrata l'inizializzazione del *descriptor set* e della *compute pipeline*.

Listing 2.7: Inizializzazione della compute pipeline

```
let descriptor_set_allocator =
    StandardDescriptorSetAllocator::new(device.clone());

let shader = loader::vec_sum::load(device)?;
let pipeline = ComputePipeline::new(
    device.clone(),
    shader
        .entry_point("main")
        .ok_or(anyhow!("No entry point in shader"))?,
    &(),
    None,
    |_| {},
)?;
let layout = pipeline.layout.set_layouts().get(0).unwrap();
let set = PersistentDescriptorSet::new(
    &descriptor_set_allocator,
    layout.clone(),
    [
        WriteDescriptorSet::buffer(0, buf_a),
        WriteDescriptorSet::buffer(1, buf_b),
        WriteDescriptorSet::buffer(2, buf_res.clone()),
    ],
)?;
```

In ultimo, si deve allocare un *command buffer*, per definire e inviare le operazioni da eseguire sul *device*. Il *command buffer* attiva la pipeline di calcolo, impostando i parametri e l'invio dei dati di input, includendo comandi per la sincronizzazione tra le diverse operazioni. Utilizzando il *command buffer* in modo efficiente è possibile ottimizzare le prestazioni del calcolo parallelo, ad esempio raggruppando le operazioni correlate in un unico buffer e inviandole in blocco al *device* GPU.

Listing 2.8: Command buffer e dispatch

```
let push_constants = PushConstants { n: len as u32 };

let mut command_buffer_builder = AutoCommandBufferBuilder::primary(
    &command_buffer_allocator,
    queue.queue_family_index(),
    CommandBufferUsage::OneTimeSubmit,
)?;
command_buffer_builder
    .bind_pipeline_compute(pipeline.clone())
    .bind_descriptor_sets(
        PipelineBindPoint::Compute,
        pipeline.layout().clone(),
        0,
        set,
    )
    .push_constants(pipeline.layout().clone(), 0, push_constants)
    .dispatch([len as u32 / 64, 1, 1])?;
let command_buffer = command_buffer_builder.build()?;

let future = sync::now(device.clone())
    .then_execute(queue.clone(), command_buffer)?
    .then_signal_fence_and_flush()?;
future.wait(None)?;

let vec_res = buf_res.to_vec();

...
```

Come si può evincere dal codice, questa versione è molto più prolissa della corrispettiva in CUDA. Il programmatore ha modo di avere più controllo sulle risorse del sistema, ma quando si devono eseguire shader semplici, categoria in cui ricadono la maggior parte di quelli per il computing, questo potrebbe essere un ostacolo alla scrittura di codice leggibile e mantenibile. Come già accennato in precedenza, si è utilizzato la libreria Vulkano che permette di scrivere l'applicazione in Rust, se si fosse scelto di scrivere l'applicazione con C++ avremmo avuto codice ancora più verboso, per via della deallocazione manuale della memoria, e sicuramente più prono ad errori, dato che la gestione della memoria sarebbe stata affidata interamente al programmatore. In questo Rust è venuto in aiuto, gestendo tutte le risorse col paradigma RAII e permettendoci di concentrarci sulla logica applicativa. Nel prossimo capitolo verrà spiegata come Rust gestisce la memoria e perché risulta più facile scrivere codice con elevato grado di correttezza e manutenibilità.

2.3 Rust

Rust è un linguaggio di programmazione di sistema progettato con un focus sulla sicurezza, sulla concorrenza e sulle prestazioni. In particolare l'obiettivo primario del linguaggio è permettere la gestione della memoria in modo automatico, ma senza ricorrere a *garbage collector* o contatori di riferimenti.

Gli errori di memoria sono una delle principali cause di vulnerabilità nel software [19], causando disagi agli utenti e soprattutto perdite economiche per aziende. Rust mitiga questo problema incorporando funzionalità avanzate che forniscono sicurezza della memoria senza compromettere le prestazioni. Al centro della strategia di gestione della memoria di Rust ci sono le semantica di *ownership* e il *borrow checker* [16], che lavorano insieme per produrre codice formalmente *safe*. Il modello di ownership di Rust ruota attorno al concetto di possesso: ogni valore nel programma ha un unico *owner* in un dato momento. La deallocazione della memoria avviene quando le variabili escono fuori dallo *scope* sintattico, come avviene col paradigma RAII. Questo approccio deterministico elimina problemi come *memory leaks* e *double free*, comuni nei linguaggi intrinsecamente *unsafe* come C e C++. Quando un valore viene assegnato a un'altra variabile o passato a una funzione, il possesso viene trasferito, garantendo che ci sia sempre un chiaro owner responsabile del rilascio della memoria. Inoltre, Rust utilizza le semantica di spostamento, impedendo l'uso accidentale di puntatori invalidati o riferimenti pendenti.

Il borrow checker è uno strumento di analisi statica integrato nel compilatore Rust, che impone le regole che governano ownership e borrowing. Il borrowing consente a più riferimenti di accedere a un valore senza trasferire il possesso. Tuttavia, il borrow checker garantisce che questi riferimenti non sopravvivano al valore che prendono in prestito o che provochino *race condition* dei dati. Rust supporta due tipi di borrowing: borrowing immutabile e mutabile. Il borrowing immutabile consente a più lettori di accedere a un valore contemporaneamente, mentre il borrowing mutabile garantisce l'accesso esclusivo, impedendo scritture concorrenti. Per facilitare il borrowing, Rust introduce il concetto di *lifetimes*, che tracciano la durata per cui i riferimenti sono validi. I lifetimes consentono al borrow checker di analizzare staticamente il codice e garantire che i riferimenti presi in prestito non sopravvivano ai valori a cui fanno riferimento.

Listing 2.9: Controllo dei puntatori in fase di compilazione

```rust
fn main(){
    let x = Vec::new();

    // y ora è l'owner del dato di y
    let y = x;

    drop(x); // errore di compilazione, y è l'owner

    let mut x = vec![1, 2, 3];
    let first = &x[0];

    // x viene spostato in y, first non è più valido
    let y = x;

    println!("{*first}"); // errore di compilazione
}
```

Il modello di ownership di Rust facilita la concorrenza sicura applicando regole rigorose sull'aliasing mutabile a tempo di compilazione. Rust garantisce che l'accesso concorrente ai dati condivisi sia privo di *race condition*, rendendo più facile scrivere codice concorrente corretto e scalabile. Nonostante le sue forti garanzie sulla sicurezza, Rust offre prestazioni comparabili, e in alcuni casi migliori, rispetto a C e C++ [20]. Sfruttando ownership e borrowing per eliminare il sovraccarico a tempo di esecuzione associato alla garbage collection e ai controlli a runtime, Rust produce codice efficiente e di basso livello adatto alla programmazione di sistema e alle applicazioni critiche in termini di prestazioni.

Listing 2.10: Fearless concurrency

```rust
use std::rc::Rc; // reference counting
use std::sync::Arc; // atomic reference counting
use std::sync::Mutex; // mutable exclusion

fn main(){
    let rc = Rc::new("not thread safe");
    std::thread::spawn(move || {
        // errore di compilazione, Rc non implementa il trait Send
        println!("{:?}", rc);
    });

    let arc = Arc::new("thread safe");
    std::thread::spawn(move || {
        // Arc implementa Send, quindi è thread safe
        println!("{:?}", arc);  // output: thread safe
    });
```

```
17
18      let mut v = Vec::new();
19      std::thread::spawn(move || {
20          v.push(1);
21      });
22      // errore di compilazione, accesso a v dopo il movimento
23      println!("{:?}", v);
24
25      // Mutex permette l'accesso concorrente in mutua esclusione
26      let v = Arc::new(Mutex::new(Vec::new()));
27      let arc_v = Arc::clone(&v);
28
29      std::thread::spawn(move || {
30          arc_v.lock().unwrap().push(1);
31      }).join().unwrap();
32
33      println!("{:?}", *v.lock().unwrap()); // output: [1]
34  }
```

L'approccio innovativo di Rust alla gestione della memoria e alla sicurezza, offre significativi vantaggi rispetto ai linguaggi tradizionali di programmazione di sistema. Combinando sicurezza della memoria, concorrenza e prestazioni, Rust permette ai programmatori di scrivere software affidabile, efficiente e scalabile senza un enorme sforzo cognitivo. Inoltre, Rust si posiziona da 8 anni come linguaggio più amato dagli sviluppatori [21], per il suo ecosistema ricco, i numerosi tool disponibili e la facilità con cui si riesce a produrre software efficiente e sicuro. L'esigenza di produrre codice sicuro, ma efficiente, non è solo del settore privato, per massimizzare il ritorno di investimento nello sviluppo software, ma premura anche del settore pubblico: lo dimostra il recente endorsement del governo USA per quanto riguarda la sicurezza del cyberspazio [22]. Rust ha tutte le premesse per ridefinire il panorama della programmazione di sistema per gli anni a venire e la scelta di usarlo, per questo tipo di applicazioni, si sposa bene con la natura verbosa e dettagliata di Vulkan.

Capitolo 3

Analisi dei Requisiti

Questo studio nasce dalla necessità dell'azienda Data Reply di aggiornare un microservizio CUDA per renderlo più moderno e mantenibile. Il microservizio espone delle REST API, tramite una libreria legacy C++ non più mantenuta, a chiamate kernel CUDA per la risoluzione di matrici QUBO.

3.1 Il problema

Una matrice QUBO è un concetto fondamentale nel campo della computazione quantistica: è una matrice che rappresenta un problema di ottimizzazione combinatoria, dove le variabili di decisione sono binarie e l'obiettivo è di massimizzare o minimizzare una funzione obiettivo quadratico. La funzione obiettivo è espressa come una combinazione di termini quadrati di variabili binarie, e la matrice QUBO rappresenta esattamente questi termini.

Ad esempio, se avessimo un problema di ottimizzazione che coinvolge la distribuzione di risorse limitate tra diverse attività, potremmo rappresentare le variabili binarie come `0`, se l'attività non viene svolta, e `1`, se viene svolta. La funzione obiettivo potrebbe essere minimizzare il costo totale delle risorse impiegate, considerando le interazioni tra le attività. Queste interazioni quadratiche tra le variabili binarie costituirebbero i termini della matrice QUBO.

Risolvere una matrice QUBO significa trovare la combinazione di valori binari, per le variabili, che minimizza o massimizza la funzione obiettivo. Questo processo può essere complesso poiché potrebbe implicare la valutazione di tutte le possibili combinazioni di variabili. Tuttavia, l'interesse principale nelle matrici QUBO è nell'utilizzo di algoritmi di ottimizzazione, sia classici che quantistici, che possono trovare soluzioni approssimate in tempi ragionevoli. Gli algoritmi quantistici, in particolare, mostrano un potenziale significativo nel risolvere problemi QUBO in modo efficiente grazie alle proprietà intrinseche della meccanica quantistica.

Non ci addentreremo oltre per quanto riguarda la teoria dei problemi QUBO e degli algoritmi di ottimizzazione, in quanto non è strettamente necessario per comprendere il lavoro svolto. Tuttavia, è importante sottolineare che la risoluzione di matrici QUBO è un problema computazionalmente intensivo, e richiede l'uso di risorse hardware specializzate e sofisticate tecniche software per ottenere risultati in tempi ragionevoli. Le operazioni da effettuare per la risoluzione della matrice QUBO possono essere riassunte in:

- Generazione delle possibili soluzioni
- Calcolo dell'energia di ogni soluzione
- Identificazione della soluzione ottima, cioè a energia minima (o massima, a seconda del problema)

Per calcolare il costo della soluzione, è necessario moltiplicare la matrice QUBO per la soluzione e moltiplicare il risultato per la soluzione trasposta. Il costo della soluzione è anche chiamato *energia*.

Listing 3.1: Pseudocodice per la risoluzione di una matrice QUBO

```
# matrice NxN triangolare superiore
QUBO = [[...], ..., [...]]

# spazio delle soluzioni possibili,
# rappresentate come vettori di N elementi binari
sol_space = generate_sol_space(N)

solutions = list()
for vec_sol in sol_space:
  energy = mat.mul(mat.mul(vec_sol, QUBO), vec_sol.T)
  solutions.add((energy, sol))

# soluzione ottima a energia minima
min_sol = min(sol.energy for sol in solutions)

# soluzione ottima a energia massima
max_sol = max(sol.energy for sol in solutions)
```

Per una matrice QUBO di dimensione N, il numero di soluzioni possibili è 2^N, quindi il numero di operazioni da effettuare per trovare la soluzione ottima globale è esponenziale rispetto a N. Per valori di N anche relativamente piccoli, il numero di operazioni diventa rapidamente proibitivo e l'approccio *naive* in 3.1, che itera su tutto lo spazio delle soluzioni, è chiaramente infattibile. Si possono adottare diverse strategie per ridurre il numero di iterazioni necessarie, come adottare

un approccio di *hill climbing* per trovare ottimi locali, o algoritmi genetici per esplorare lo spazio delle soluzioni in modo più efficiente oppure metodi *greedy* con constraint temporali. Tuttavia, questi algoritmi richiedono comunque un numero significativo di operazioni, soprattutto per le moltiplicazioni matriciali: quindi è necessario sfruttare al massimo le risorse hardware disponibili per ottenere risultati in tempi ragionevoli. Con la computazione eterogenea si può parallelizzare la parte moltiplicativa, ottenendo un notevole incremento nelle prestazioni. Questo è il motivo per cui il microservizio in questione è stato originariamente scritto per sfruttare CUDA e le GPU NVIDIA, che sono particolarmente adatte per questo tipo di calcoli.

Un esempio di risoluzione con CUDA è mostrato in 3.2, pur adottando ancora un approccio naive per la parte CPU, l'algoritmo del kernel CUDA è ottimizzato per essere efficiente negli accessi in memoria.

Listing 3.2: CUDA moltiplicazine matrice QUBO

```
__global__ void solverKernel(const uint *sol,
        const uint *mat, uint *res, const uint dim) {
  const uint col = blockIdx.x * blockDim.x + threadIdx.x;

  if (col < dim) {
    uint pos = 0;
    uint value = 0;

    // itera solo sulla parte triangolare superiore della matrice
    for (uint i = 0; i <= col; i++) {
      value += sol[i] * mat[pos + col];
      pos += dim - i - 1;
    }
    res[col] = value * sol[col];
  }
}

void solver(const uint *h_mat, const uint dim) {
  // inizializzazione variabili
  ...

  dim3 dimGrid((dim + 32 - 1) / 32.0, 1);
  dim3 dimBlock(32, 32);

  for (uint i = 0; i < (2 << dim - 1); i++) {
    gen_vec_sol(vec, i, dim);

    // entrambe le moltiplicazioni in GPU
    solverKernel<<<dimGrid, dimBlock>>>(vec, mat, res, dim);
    cudaDeviceSynchronize();
```

```
31      // reduce del risultato in CPU
32      auto energy = 0;
33      for (auto i = 0; i < dim; i++) {
34        energy += res[i];
35      }
36
37      if (energy < min_energy) {
38        min_energy = energy;
39        std::copy(vec, vec + size_vec, sol);
40      }
41    }
42
43    // free variabili
44    ...
45  }
```

Per quanto riguarda la parte web del microservizio, è necessario scegliere librerie che siano in grado di esporre REST API e fare il parsing di oggetti di grandi dimensioni, dato che le matrici QUBO possono anche avere migliaia di righe e colonne. Nonostante esistano molte librerie di questo tipo è importante che siano testate e mantenute, in modo da garantire standard di sicurezza per non compromettere asset aziendali.

3.2 Possibili Approcci

Il problema può essere affrontato in diversi modi, in particolare, si sono considerati i seguenti approcci:

- Sostituire la sola libreria legacy C++ con un'altra nello stesso linguaggio.
- Sostituire la libreria legacy C++ usando linguaggi che supportino meglio lo sviluppo web, come Python, Scala, Java, Go o Rust e integrarvi la parte CUDA per poter essere richiamata a runtime.
- Sostituire la parte di computing CUDA con Vulkan e scegliere per la parte web linguaggi che ne supportassero facilmente l'integrazione.

Dato che la priorità era quella di deprecare la libreria legacy C++, ed eventualmente, usare Vulkan (se avesse portato benefici prestazionali), si è scelto di iniziare attuando il terzo approccio, in quanto avrebbe permesso di ottenere un microservizio più moderno, performante e mantenibile. Inoltre, dato che la parte CUDA era già scritta in modo modulare, sarebbe stato eventualmente facile passare al secondo approccio.

Un altro importante requisito era che l'intero microservizio fosse compilabile come unico binario, senza che per il deploy in produzione fosse presente un interprete runtime. Questo restringeva la scelta dei possibili linguaggi ai soli che avessero compilatori AOT, che fossero, cioè, in grado di produrre direttamente codice macchina. Tra i linguaggi che soddisfacevano tutti i requisiti e con cui avevo maggiore esperienza, si è scelto Rust. Ritengo che Rust sia il più adatto par vari motivi: il suo vasto ecosistema web (nonostante comunque meno florido di altri linguaggi come Java e Go), le performance eccellenti a carichi di lavoro intensi, e la sua capacità di interfacciarsi con librerie C/C++ tramite FFI. Inoltre, il supporto a Vulkan è molto buono, dato che ultimamente è il linguaggio scelto dagli studio per sviluppare videogiochi, con documentazione e risorse in costante crescita.

I requisiti possono essere, quindi, riassunti come segue:

Requisiti funzionali

- Il microservizio deve essere in grado di risolvere matrici QUBO
- Il microservizio deve essere esposto tramite REST API
- Il microservizio deve essere scritto in Rust
- Il microservizio deve usare Vulkan o CUDA per la parte di GPU computing

Requisiti non funzionali

- Il microservizio deve essere performante
- Il microservizio deve essere mantenibile
- Il microservizio deve essere facilmente testabile
- Il microservizio deve essere sicuro e *memory safe*
- Il microservizio deve essere resiliente a picchi di carico intensi ed essere facilmente scalabile
- Il microservizio deve essere facilmente integrabile con altri microservizi

Requisiti di sistema

- Il microservizio deve essere eseguibile su un server Linux con GPU NVIDIA
- Il microservizio deve essere un unico file eseguibile
- Il microservizio deve essere facilmente installabile e configurabile

Per quanto riguarda la compatibilità con le GPU NVIDIA, è un requisito derivato dal sistema in produzione dell'azienda. Per motivi storici e prestazionali sono presenti solo GPU NVIDIA, ed è quindi fondamentale garantirne la compatibilità. Implementando una versione con Vulkan, non solo si mantiene la compatibilità con le GPU NVIDIA, ma si estende automaticamente il supporto anche ad altri vendor GPU, come AMD o Intel.

Capitolo 4

Design dei Benchmark e Risultati

In questo capitolo verranno presentati i benchmark sviluppati per confrontare le performance delle soluzioni basate su CUDA e Vulkan. Verranno, inoltre, illustrati i risultati ottenuti e il processo di generazione dei dati.

4.1 Benchmark

Sono stati sviluppati due tipi di benchmark: uno per la somma di vettori e uno per la moltiplicazione di matrici sparse. Le misurazioni hanno tenuto conto sia del tempo di esecuzione su GPU che del tempo di trasferimento dei dati tra memoria *host* e *device*. I calcoli sono stati eseguiti per tre tipi di dato: `uint`, `float` e `double` (rispettivamente `u32`, `f32` e `f64` in Rust). Questo ha permesso di capire come codice ottimizzato in modo diverso, in base al tipo di dato, da CUDA o Vulkan, intacchi le performance.

La suite è stata eseguita, per tutte le implementazioni, su una macchina con la stessa versione dei driver, per garantire la coerenza dei risultati. In fig. 4.1 sono mostrate le caratteristiche della GPU della macchina. I componenti principali del sistema ibrido sono: CPU Intel Xeon Platinum 8259CL e GPU NVIDIA Tesla T4 Tensor da 16 GB di memoria GDDR6.

Mentre all'inizio si era pensato di includere anche una versione dei benchmark su CPU, si è poi deciso di accantonare l'idea, in quanto non sono rilevanti per il confronto tra le due soluzioni e poiché i tempi di esecuzione dell'intera suite erano già molto elevati.

La compilazione di entrambi gli eseguibili è stata ottimizzata, tramite flag dei rispettivi compilatori 4.1, in modo da generare codice con il più alto grado di performance possibile. Sebbene questa accortezza non intacchi le prestazioni

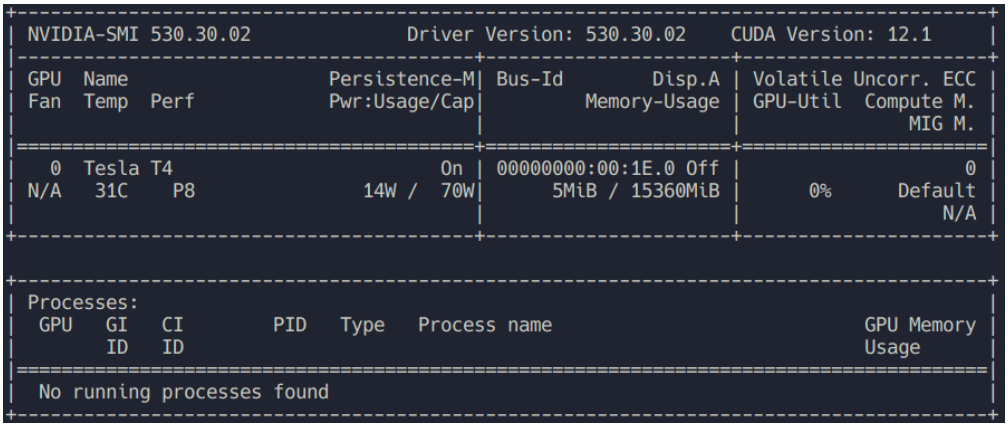

Figura 4.1: Specifiche e driver della GPU su cui sono stati eseguiti i benchmark

su GPU, è comunque necessaria per avere *throughput* massimo della CPU per il trasferimento dei dati dalla memoria *host* a quella *device* e viceversa, tramite la PCIe del sistema.

Listing 4.1: Flag di compilazione CUDA e Rust

```
# lo strip dei binari permette di ridurre la dimensione
# dei file eseguibili, in quanto vengono rimossi i simboli
# di debug non necessari per il benchmark

# Makefile CUDA
nvcc main.cu -o microbench_cuda -O3 && strip microbench_cuda

# Cargo.toml
[profile.release]
strip = true
panic = "abort"
opt-level = 3
```

I tempi di esecuzione sono stati calcolati mediante funzioni fornite dalla libreria standard di entrambi i linguaggi, e sono stati ricalcolati in più iterazioni per ottenere una media più accurata. Per Vulkan si è usata la struct `std::time::Instant` di Rust, mentre per CUDA si è sviluppato un timer che incorpora le funzioni `cudaEventCreate` e `cudaEventElapsedTime`, mostrato in 4.2, per implementare API simili a quelle usate in Rust.

I tempi risultanti sono stati poi stampati su standard output specificando il tipo di benchmark di riferimento: ad esempio `SUM VEC and NO COPY: 0.0000 ms` oppure `MUL MAT and COPY: 0.0000 ms`. Questo ha permesso di verificare che i

tempi di esecuzione fossero coerenti con le operazioni eseguite e di poter confrontare i risultati ottenuti con quelli attesi.

Listing 4.2: Timer CUDA

```
class Timer {
  cudaEvent_t start, stop;
  float elapsedTime = 0.0;

 public:
  void startTimer() {
    cudaEventCreate(&start);
    cudaEventRecord(start, 0);
  }

  void stopTimer() {
    cudaEventCreate(&stop);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
  }

  float calculateElapsed() {
    cudaEventElapsedTime(&elapsedTime, start, stop);
    return elapsedTime;
  }
};
```

Inoltre, ogni eseguibile accetta da linea di comando degli argomenti per specificare la dimensione di vettori e matrici con cui eseguire i calcoli, il numero di iterazioni e il tipo di dato da testare 4.3.

Listing 4.3: Esecuzione benchmark

```
# CUDA
./microbench_cuda $vec_exp $dim_m $dim_n $dim_k $num_run $dtype

# Rust
./microbench_vulkan $vec_exp $dim_m $dim_n $dim_k $num_run $dtype
```

Il parametro **$vec_exp** indica la grandezza dei vettori: viene eseguita la somma su due vettori di 2^{vec_exp} elementi, con **vec_exp=29** per i tipi di dato **uint** e **float**, e **vec_exp=28** per il dato **double** (tutti vettori da 2GB, dato che **double** ha il doppio dei bit di un **uint** o **float**). Durante i test ci si è accorti che aumentando la dimensione dei vettori oltre questo valore, il programma dava errore di allocazione di memoria, dovuto al fatto che raddoppiando il numero di elementi raddoppiavano

conseguentemente i thread da eseguire e quindi l'overhead dovuto al caricamento del contesto. Inoltre questa dimensione è sufficiente per testare ler performance implementative, senza aumentare eccessivamente i tempi di esecuzione.

Per le matrici, invece, i tre elementi `$dim_m, $dim_n, $dim_k`, indicano la dimensione di righe e colonne, quindi matrici di dimensione $M \times N$ e $N \times K$, che producono è una matrice grande $M \times K$. Nonostante le matrici QUBO siano quadrate e triangolari superiori, si è scelto di generalizzare il più possibile i benchmark, e quindi testare il prodotto di due matrici sparse. Quello che ci si aspetta è che, per la risoluzione di matrici QUBO di dimensioni paragonabili, i tempi d'esecuzione siano leggermente ridotti poiché, come si è visto in 3.1, si accede solo alla diagonale superiore, eseguendo però due moltiplicazioni sui dati. I valori usati per i benchmark sono `dim_m=dim_k=1024` e `dim_n=4096`. Le dimensioni sono state scelte in modo da mantenere un compromesso tra dimensioni e tempo di esecuzione, in modo da poter eseguire un numero di iterazioni sufficiente per ottenere una media accurata: durante i test ci si è accorti che, anche aumentando le dimensioni, la differenza del tempo di esecuzione tra CUDA e Vulkan rimaneva pressoché invariata, ma aumentava considerevolmente il tempo di esecuzione.

Il parametro `$num_run` indica il numero di iterazioni da eseguire per calcolare la media dei tempi di esecuzione. Infine, `$dtype` indica il tipo di dato da usare per i calcoli, e può essere `uint`, `float` o `double`.

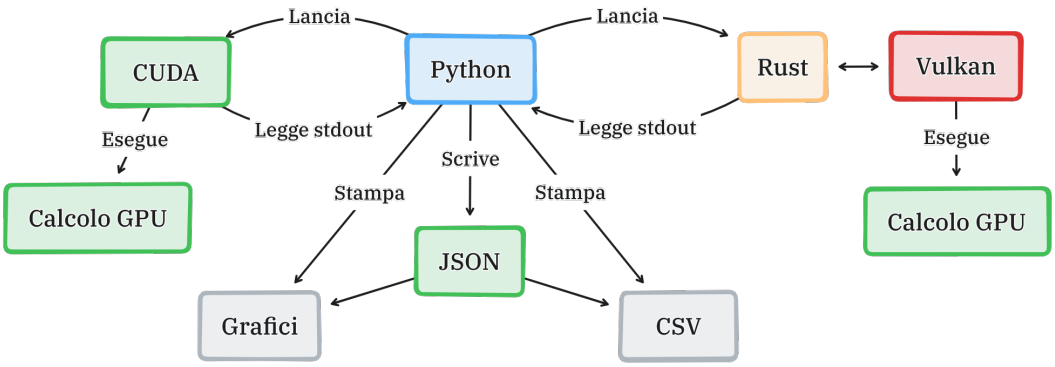

Figura 4.2: Schema della suite di benchmark

Per automatizzare il processo di benchmarking, si è scritto uno script Python che lancia come sotto-processi gli eseguibili dei benchmark e, leggendo tramite *pipe* lo *standard output* del programma, salva i risultati di ogni iterazione in un file JSON. Infine, lo script riassume i risultati su un file CSV calcolando le medie dei tempi e produce un grafico che mostra i tempi di esecuzione, evidenziando la media e gli outlier.

I dati contenuti nei vettori e nelle matrici sono stati generati casualmente a ogni esecuzione, in modo da garantire che input diversi non influenzino in alcun modo i risultati. Il risultato delle operazioni di kernel e shader viene verificato tramite funzioni `assert` presenti nei linguaggi usati. Somma di vettori e moltiplicazione di matrici vengono, quindi, eseguite anche su CPU e confrontate con i risultati ottenuti dall'esecuzione su GPU, in modo da garantire che le operazioni siano corrette. A questo proposito c'è da specificare che, per quanto riguarda la moltiplicazione di matrici di tipo `float` e `double`, la funzione di confronto è scritta in modo da eseguire su CPU una FMA, in quanto questa operazione è eseguita in modo automatico e trasparente su GPU e influenza l'approssimazione dei risultati e quindi la verifica della correttezza.

La FMA è un'operazione che consente di eseguire una moltiplicazione tra due operandi a virgola mobile e di aggiungere un terzo operando al risultato, in un unico passaggio. Questo approccio offre diversi vantaggi, inclusi una maggiore efficienza computazionale, una riduzione della latenza e un aumento della precisione. Le GPU moderne incorporano unità hardware specializzate per eseguire operazioni FMA in modo efficiente, contribuendo così alla loro capacità di elaborazione ad alte prestazioni [23]. Anche i moderni processori x86 e ARM incorporano unità FMA, ma non sempre sono in grado di eseguire operazioni FMA in modo efficiente come le GPU. Sia CUDA che Rust espongono funzioni nella libreria standard per eseguire operazioni FMA, è stato, quindi, possibile confrontare in modo preciso i risultati ottenuti su CPU con quelli ottenuti su GPU.

Per diminuire il boilerplate e la quantità di codice da scrivere per ogni benchmark, le funzioni sono state scritte usando la *programmazione generica*, in modo da poter eseguire le stesse operazioni sui diversi tipi di dato. Questo ha permesso di scrivere un'unica volta il codice per la somma di vettori e la moltiplicazione di matrici, e di poterlo eseguire su tutti e tre i tipi di dato. In CUDA si è sfruttato i `template`, come mostrato in 4.4, sia per le funzioni kernel che per le funzioni *host* che le richiamano. Le funzioni vengono generate, a tempo di compilazione, per ogni tipo di dato effettivamente usato, ed è quindi compito del compilatore generare il codice specifico per i dati.

Listing 4.4: Programmazione generica in CUDA

```
template <typename T>
__global__ void vecAddKernel(const T *vec_a, const T *vec_b,
    T *res, const uint n) { ... }

template <typename T>
__global__ void matrixMulKernel(const T *mat_a, const T *mat_b,
    T *res, const uint m, const uint n, const uint k) { ... }

template <typename T>
void vecAdd(const T *h_data_a, const T *h_data_b, const uint len) {
  ...
  vecAddKernel<<<...>>>(vec_a, vec_b, res_data, len);
  ...
}

template <typename T>
void matMul(const T *h_mat_a, const T *h_mat_b,
    const uint m, const uint n, const uint k) {
  ...
  matrixMulKernel<<<...>>>(mat_a, mat_b, res, m, n, k);
  ...
}

template <typename T>
void benchmark(const uint len, const uint dim_m,
    const uint dim_n, const uint dim_k, const uint num_run) {
  ...
  for (uint i = 0; i < num_run; i++) {
    vecAdd(vec_a, vec_b, len);
    matMul(mat_a, mat_b, dim_m, dim_n, dim_k);
  }
  ...
}

int main( ... ) {
  ...
  if (strcmp(dtype, "uint") == 0) {
    benchmark<uint>(len, dim_m, dim_n, dim_k, num_run);
  } else if (strcmp(dtype, "float") == 0) {
    benchmark<float>(len, dim_m, dim_n, dim_k, num_run);
  } else if (strcmp(dtype, "double") == 0) {
    benchmark<double>(len, dim_m, dim_n, dim_k, num_run);
  } else {
    return -1;
  }
  ...
}
```

Per Vulkan, invece, dato che GLSL non supporta la programmazione generica, si è dovuto scrivere tre versioni per ogni shader, una per ogni tipo di dato. Tramite la creazione di `enum` specifici e il *pattern matching* di Rust si è comunque riusciti a generalizzare abbastanza le funzioni dei benchmark evitando di scrivere manualmente tre versioni diverse. In 6.3 è mostrato l'approccio usato per ovviare alla mancanza di generics in GLSL, da notare che `ld` è il modulo con cui vengono caricati gli shader, tramite macro a tempo di compilazione, come mostrato in 2.5.

Listing 4.5: Loading shader con enum

```
#[allow(non_camel_case_types)]
#[derive(Clone, Copy)]
pub enum Type {
  uint,
  float,
  double,
}

pub enum Operation {
  Sum(Type),
  Mul(Type),
}

impl Operation {
  pub fn load_shader(&self, device: Arc<Device>)
    -> Result<Arc<ShaderModule>> {
    let shader = match self {
      Operation::Sum(Type::uint) => ld::sum_uint::load(device)?,
      Operation::Sum(Type::float) => ld::sum_float::load(device)?,
      Operation::Sum(Type::double) => ld::sum_double::load(device)?,
      Operation::Mul(Type::uint) => ld::mul_uint::load(device)?,
      Operation::Mul(Type::float) => ld::mul_float::load(device)?,
      Operation::Mul(Type::double) => ld::mul_double::load(device)?,
    };

    Ok(shader)
  }
}
```

Per quanto riguarda le operazioni sui dati in 4.6 è mostrato come usando il *composition pattern* in Rust, è possibile implementare il tratto `Add` per la somma dei vettori e implementare la FMA per la moltiplicazione delle matrici. Il resto della logica rimane invariata rispetto a CUDA, con una funzione `benchmark::<T>(..., Operation::...(Type::T))` che specializza il tipo di dato ed esegue tutte le operazioni necessarie.

Listing 4.6: Programmazione generica in Rust

```rust
#[allow(non_camel_case_types)]
#[derive(PartialEq, Debug, Clone, Copy)]
pub enum DataType {
  uint(u32),
  float(f32),
  double(f64),
}

impl Add for DataType {
  type Output = Self;

  fn add(self, rhs: Self) -> Self::Output {
    match (self, rhs) {
      (Self::uint(x), Self::uint(y)) => Self::uint(x + y),
      (Self::float(x), Self::float(y)) => Self::float(x + y),
      (Self::double(x), Self::double(y)) => Self::double(x + y),
      _ => unreachable!(),
    }
  }
}

impl DataType {
  pub fn fma(a: Self, b: Self, c: Self) -> Self {
    match (a, b, c) {
      (Self::uint(a), Self::uint(b), Self::uint(c)) => {
        Self::uint(a * b + c)
      }
      (Self::float(a), Self::float(b), Self::float(c)) => {
        Self::float(a.mul_add(b, c))
      }
      (Self::double(a), Self::double(b), Self::double(c)) => {
        Self::double(a.mul_add(b, c))
      }
      (Self::float(a), Self::float(b), Self::uint(c)) => {
        Self::float(a.mul_add(b, c as f32))
      }
      (Self::double(a), Self::double(b), Self::uint(c)) => {
        Self::double(a.mul_add(b, c as f64))
      }
      _ => unreachable!(),
    }
  }
}
```

I benchmark sono stati condotti utilizzando matrici vettorizzate, le quali sono memorizzate in memoria in modo tale da richiedere un solo accesso per ogni elemento. Per ottimizzare l'accesso ai dati, le matrici sono state memorizzate in modo *col-major* e *row-major*, cioè rispettivamente con le colonne o righe memorizzate in sequenza. Questa scelta è fondamentale per sfruttare al meglio la *cache* di CUDA e garantire accesso ai dati *coalesced* [14]. Le prestazioni migliorano di poco rispetto a un approccio tradizionale *row-major*, ma è comunque importante per garantire che i dati siano accessibili in modo efficiente.

Listing 4.7: Accesso ai dati in col-major e row-major

```
const uint row = blockIdx.y * blockDim.y + threadIdx.y;
const uint col = blockIdx.x * blockDim.x + threadIdx.x;

if (row < m && col < k) {
  auto value = 0;

  for (uint i = 0; i < n; i++) {
    // invece di mat_a[row * n + i] in row-major
    value += mat_a[i * m + row] * mat_b[i * k + col];
  }
  res[row * k + col] = value;
}
```

4.2 Dati

In tabella 4.1 sono riassunti i risultati ottenuti con CUDA, mentre in tabella 4.2 sono mostrati i risultati ottenuti con Vulkan. Le misure sono fornite in millisecondi e arrotondati alla terza cifra decimale.

uint			
	min	max	mean
SUM VEC and NO COPY	25.213	25.115	25.139
SUM VEC and COPY	246.253	224.671	234.274
MUL MAT and NO COPY	12.641	12.568	12.622
MUL MAT and COPY	31.283	31.048	31.132
float			
	min	max	mean
SUM VEC and NO COPY	25.212	25.113	25.14
SUM VEC and COPY	245.956	224.735	232.036
MUL MAT and NO COPY	12.649	12.575	12.626
MUL MAT and COPY	31.264	31.062	31.135
double			
	min	max	mean
SUM VEC and NO COPY	25.527	25.191	25.428
SUM VEC and COPY	212.213	188.258	194.658
MUL MAT and NO COPY	38.281	36.98	37.702
MUL MAT and COPY	99.823	97.256	99.376

Tabella 4.1: CUDA benchmark

I grafici in figg. 4.3, 4.4 e 4.5 mostrano i tempi di esecuzione di tutte le iterazioni, evidenziando graficamente la media, la distribuzione dei dati e gli eventuali outlier.

uint			
	min	max	mean
SUM VEC and NO COPY	30.664	30.147	30.361
SUM VEC and COPY	330.553	329.959	330.16
MUL MAT and NO COPY	11.101	10.903	11.045
MUL MAT and COPY	24.453	15.058	19.124
float			
	min	max	mean
SUM VEC and NO COPY	30.701	30.157	30.398
SUM VEC and COPY	330.584	329.995	330.162
MUL MAT and NO COPY	11.089	10.969	11.034
MUL MAT and COPY	24.401	15.101	17.854
double			
	min	max	mean
SUM VEC and NO COPY	30.899	30.27	30.58
SUM VEC and COPY	330.412	330.048	330.173
MUL MAT and NO COPY	40.067	38.744	39.564
MUL MAT and COPY	86.603	51.264	64.337

Tabella 4.2: Vulkan benchmark

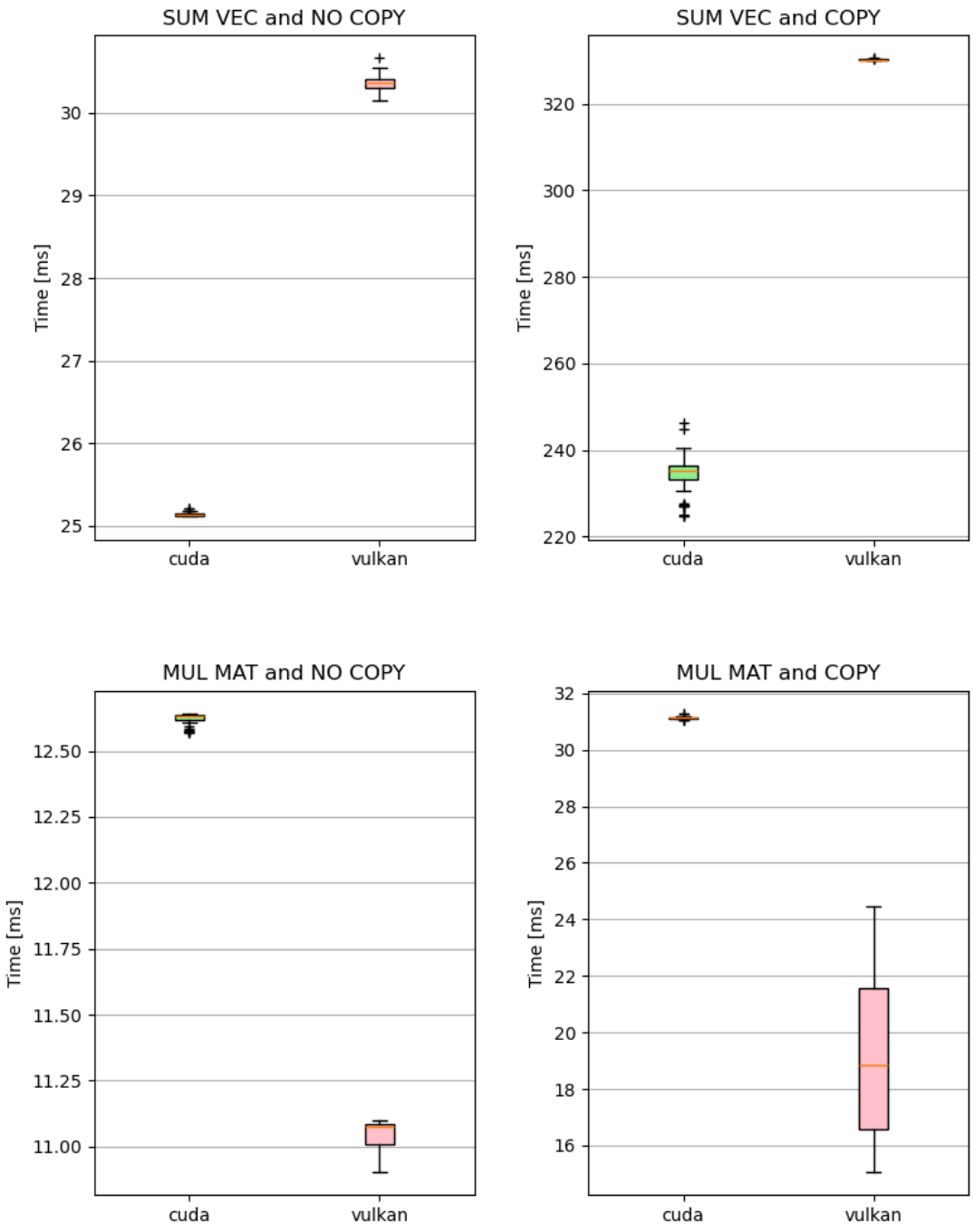

Figura 4.3: Benchmark con uint

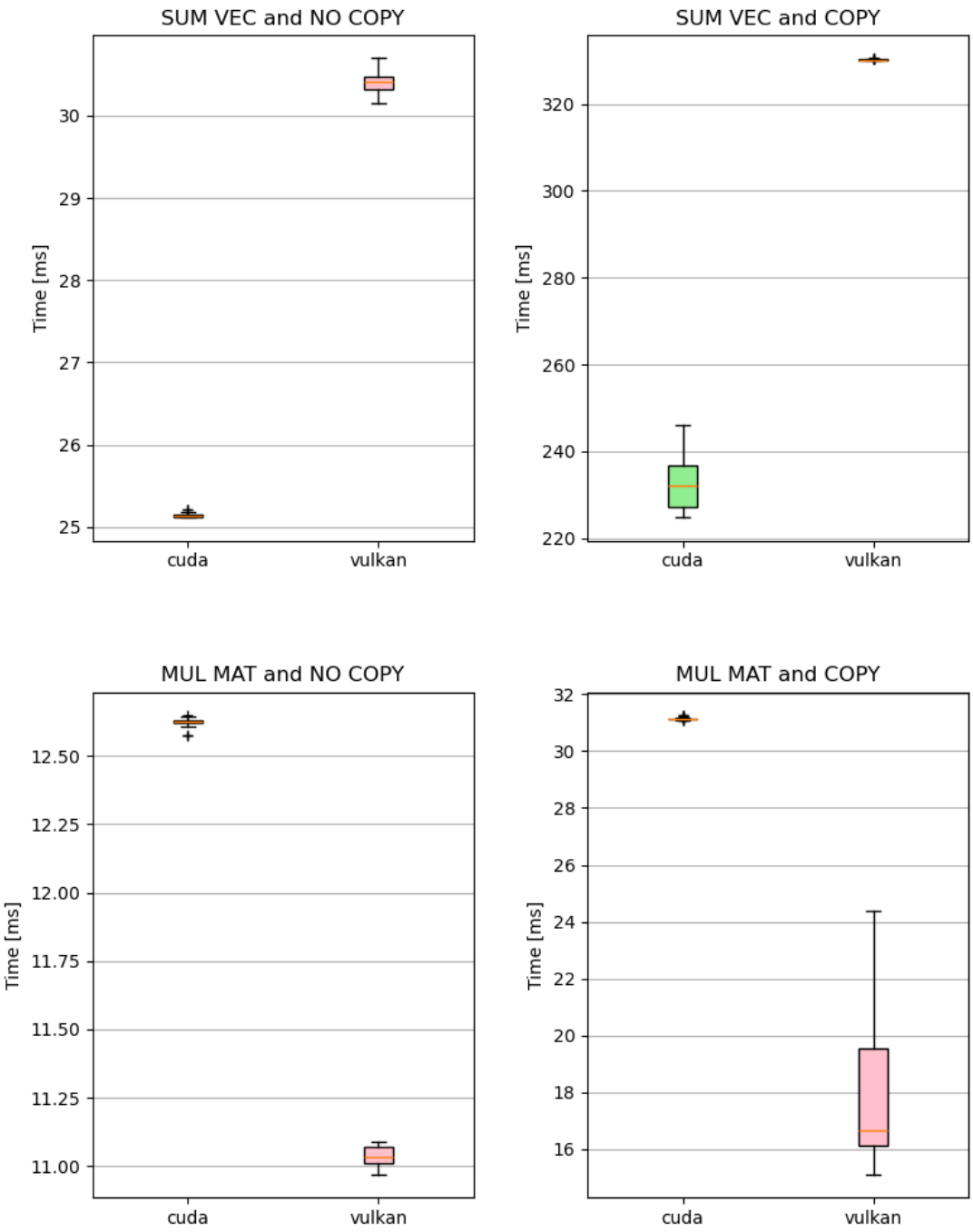

Figura 4.4: Benchmark con float

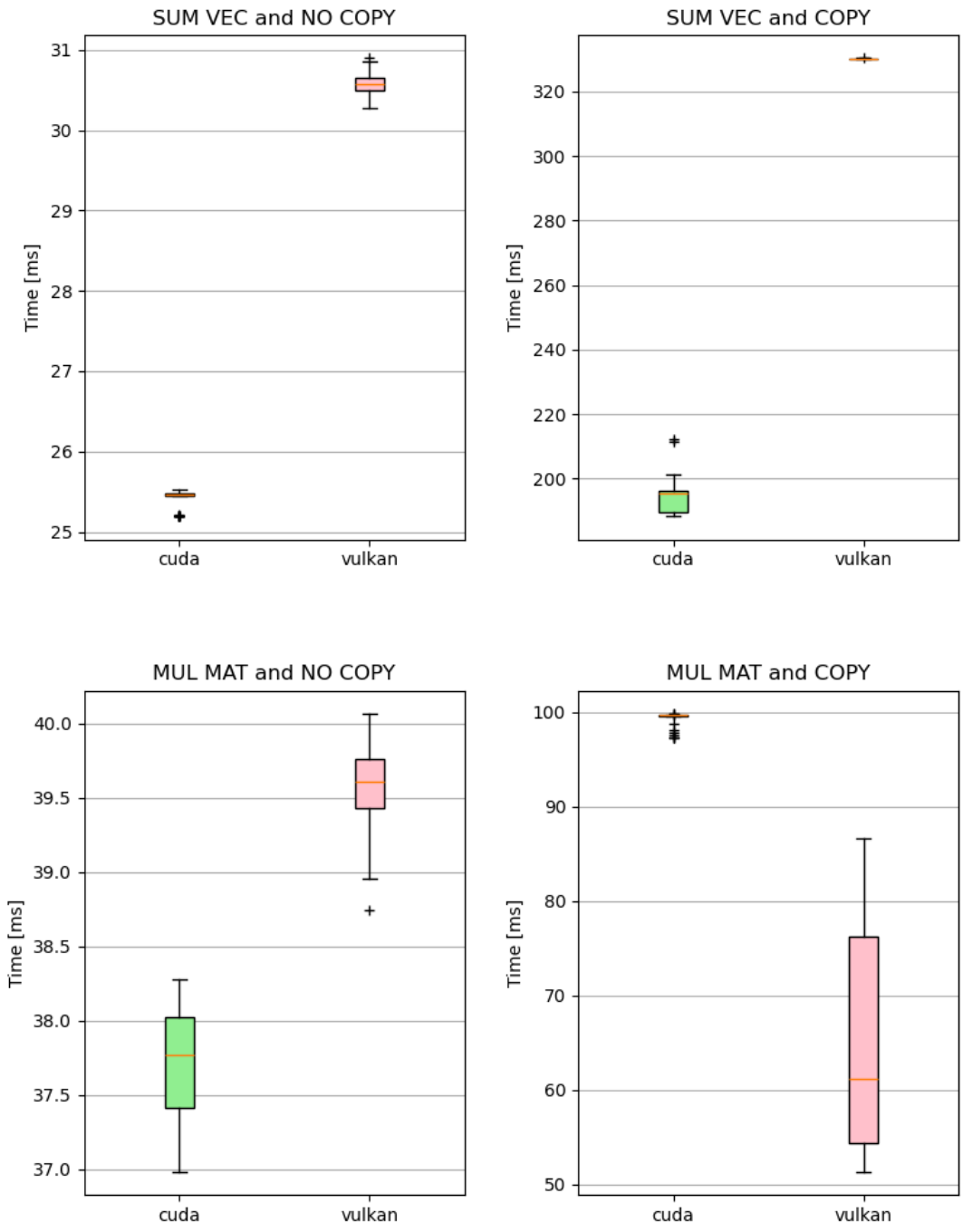

Figura 4.5: Benchmark con double

Capitolo 5

Analisi dei Benchmark

5.1 Considerazioni sui Risultati

Considerando che i benchmark sono stati eseguiti su hardware prodotto da NVIDIA e che CUDA è un framework proprietario sviluppato da NVIDIA stessa: che Vulkan il abbia prestazioni paragonabili e in alcuni casi migliori di CUDA è un dato sorprendente. Per quanto riguarda i benchmark sulle moltiplicazioni di matrici in fig. 4.5 si può notare come l'esecuzione su dati a virgola mobile con doppia precisione sia ottimizzata meglio in CUDA, anche se la differenza della media dei tempi è comunque nell'ordine del millisecondo. Se si prende in considerazione anche il trasferimento dei dati tra *host* e *device*, Vulkan risulta essere più performante su tutti i tipi di dato, dimezzando addirittura i tempi di esecuzione. Per quanto riguarda i dati sulla somma dei vettori, in tutti i benchmark Vulkan risulta meno efficiente rispetto a CUDA, ma comunque con una differenza prestazionale non marcata. Questo è dovuto alla natura stessa del benchmark, poiché il caricamento del contesto e l'inizializzazione delle risorse thread richiede tempo, data la quantità di elementi presi in esame, e tutto per eseguire solo un'operazione di somma: è indubbio che CUDA sia ottimizzato in modo migliore da questo punto di vista. In contesti reali, si sarebbe eseguito un *loop-unrolling* sul kernel per diminuire il numero dei thread rilasciati e quindi l'overhead dovuto alla gestione dei thread. Inoltre, i risultati ottenuti potrebbero essere influenzati dal fatto che i benchmark sono stati eseguiti su un singolo dispositivo e non su un cluster di GPU, quindi non si può escludere che i risultati possano variare in base al numero di GPU presenti nel sistema.

I risultati ottenuti evidenziano che Vulkan può essere considerata un'alternativa valida a CUDA per lo sviluppo di applicazioni di alto livello. Inoltre, Vulkan è un'API *open-source*, quindi non è legata a un singolo produttore e può essere utilizzata su hardware di diversi produttori. Questo è un vantaggio non trascurabile,

poiché permette di sviluppare applicazioni che possono essere eseguite su hardware di vendor diversi senza dover riscrivere il codice. Un altro elemento importante da considerare è che mentre lo scope principale di CUDA è il computing, quello di Vulkan è la grafica. Nonostante questo, i risultati ottenuti dimostrano che le ottimizzazioni di Vulkan lo rendono in grado di essere utilizzato anche per applicazioni di computing generale. Inoltre, Vulkan è in grado di estendere le funzionalità fornite tramite estensioni, per adattarsi a nuove esigenze e migliorare le prestazioni.

5.2 Considerazioni sulla Development Experience

Per quanto riguarda lo sviluppo dei benchmark, in pratica, può essere considerato come lo sviluppo di due applicazioni uguali in CUDA e in Vulkan. Come ambiente di sviluppo è stato usato *Visual Studio Code* collegato tramite *ssh* alla macchina di test, in modo da poter scrivere il codice e compilarlo direttamente sul dispositivo. Per quanto riguarda le estensioni, è stato utilizzato NVIDIA Nsight [24] per il debug sia di CUDA che di Vulkan e linter e formatter disponibili per i linguaggi. Si è usato il Vulkan SDK di LunarG [25] per la compilazione e l'esecuzione, ma senza l'uso del *Validation Layer*, per i controlli di validazione degli shader a compile time, data la semplicità degli shader sviluppati.

Per lo sviluppo del benchmark con Vulkan, l'ecosistema Rust fornisce, out of the box, una serie di strumenti che semplificano e uniformano lo sviluppo:

- **Cargo**: package manager che permette di gestire le dipendenze e di compilare il codice
- **rustfmt**: formatter per il codice sorgente
- **clippy**: linter per il codice sorgente
- **rust-analyzer**: LSP per l'autocompletamento e la navigazione del codice
- **crates.io** e **docs.rs**: repository di librerie e documentazione

L'ecosistema Rust e i tool offerti offrono benefici in termini di esperienza di sviluppo al programmatore, permettendo di concentrarsi sullo sviluppo del codice e piuttosto che sulla configurazione dell'ambiente di sviluppo. Quindi, con la sola esclusione di NVIDIA Nsight, si è stati in grado di sviluppare facilmente applicazioni di alto livello, usando soltanto tool open-source.

Per quanto riguarda lo sviluppo con CUDA, non si può dire lo stesso. Dato che CUDA, di per sè, è un ambiente proprietario, del compilatore NVCC non si possono conoscere le ottimizzazioni che vengono applicate al codice. Inoltre, essendo praticamente un ecosistema che deve essere percepito come estensione di C/C++, la mancanza di tool standardizzati e ufficialmente mantenuti dalla *Standard C++ Foundation* ha portato a una frammentazione dell'ecosistema. Questo obbliga lo sviluppatore a scegliere a monte tra i vari sistemi di sviluppo come, ad esempio, *make*, *cmake* o *ninja*, lo stile di formattazione da usare e uno tra i linter disponibili. Inoltre, la mancanza di un package manager ufficiale obbliga il programmatore a gestire le dipendenze manualmente, con la conseguente necessità di conoscere a priori le librerie da utilizzare e come installarle, col rischio di avere un sistema di building non riproducibile e poco portabile.

A parte queste considerazioni, una volta impostati i tool e l'ambiente, lo sviluppo con CUDA è molto più facile e veloce rispetto a Vulkan. L'avere un compilatore che estende il linguaggio C++ con delle keyword che isolano il codice dei kernel da quello *host*, permette di scrivere codice molto più simile a quello che si scriverebbe in C++ puro. Quanto visto nel capitolo 4 per l'uso dei *generics* è esplicativo in tal senso. Inoltre, il fatto che CUDA sia un framework proprietario permette di avere un supporto molto più affidabile rispetto a Vulkan, che è un'API open-source.

Per quanto riguarda Vulkan, la sua natura di API a basso livello rende lo sviluppo più complesso rispetto a CUDA. La necessità di gestire manualmente la creazione e allocazione delle risorse, come ad esempio le *pipeline* o i *buffer*, rende il codice molto più verboso e poco leggibile. Sebbene parte di questa complessità sia attenuata da Rust, che si occupa del rilascio delle risorse e della gestione della memoria, in confronto allo sviluppo con CUDA, lo sviluppo con Vulkan risulta più complesso e richiede sicuramente una maggiore attenzione ai particolari. Inoltre, la necessità di dover scrivere gli shader in un linguaggio specifico, come GLSL, richiede di dover conoscere un linguaggio di programmazione in più rispetto a CUDA, che permette di scrivere i kernel praticamente in C++. Questi problemi potrebbero essere risolti scrivendo librerie che elevino il grado di astrazione per Vulkan e che permettano di scrivere codice più simile a quello che si scriverebbe in CUDA, ma questo è sicuramente uno sforzo non da poco, che non è stato affrontato ai fini del benchmark.

In conclusione, sia CUDA che Vulkan hanno punti di forza e debolezze per quanto riguarda la GPGPU: quelle di CUDA derivano in larga parte dall'integrazione con C++ e il suo ecosistema, mentre, quelli di Vulkan dalla sua natura di API a basso livello, open-source, più improntata alla grafica che al computing. Per quanto riguarda lo sviluppo di applicazioni multi-piattaforma, comunque, Vulkan è, tra i due, la scelta obbligata e i risultati ottenuti dimostrano che è in grado di competere con CUDA in termini di prestazioni. Sebbene siano in sviluppo librerie per compilare direttamente codice Rust in SPIR-V, come *rspirv* [26], che potrebbe

rendere lo sviluppo di applicazioni Vulkan più semplice, al momento non è possibile fare un confronto diretto tra le due API in termini di sviluppo. Inoltre l'attenzione che sta mettendo NVIDIA su CUDA e il suo ecosistema, come dimostrato dal recente rilascio di *NVIDIA HPC SDK* [27], fa intendere che CUDA può essere ancora considerato la scelta principale per lo sviluppo di applicazioni di GPGPU.

Capitolo 6

Progetto

Alla luce delle considerazioni esposte nel capitolo precedente, tenendo conto che riscrivere l'intero microservizio in Rust con Vulkan non sia un lavoro banale (e comunque non ci sarebbero miglioramenti tangibili in termini di prestazioni), si è scelto di adottare un approccio ibrido: mantenere i kernel in CUDA e usare Rust per la parte web del microservizio. Questo permette di mantenere le prestazioni e il *know-how* di CUDA e di poter trarre benefici l'ecosistema e la *development experience* di Rust. Inoltre, l'uso di Rust per la parte web permette di avere un sistema più sicuro rispetto al C++, e, soprattutto, più portabile e facilmente mantenibile, grazie al suo sistema di gestione delle dipendenze e alla sua capacità di compilare codice per diverse architetture.

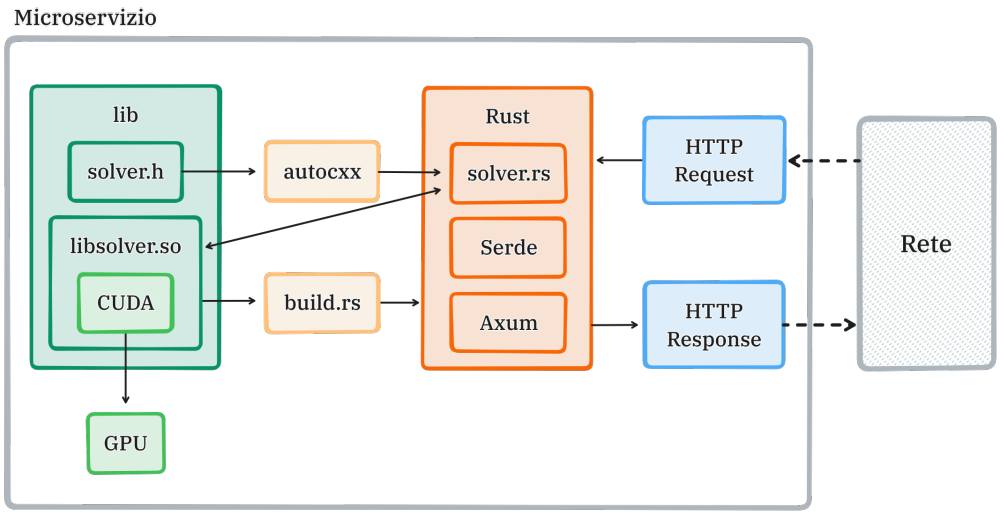

Figura 6.1: Architettura del microservizio

6.1 Struttura del microservizio

Mentre in fig. 6.1 è mostrato lo schema logico del microservizio, che illustra tutti i componenti essenziali, in 6.1 è presente la struttura delle cartelle del progetto. Il progetto è strutturato in modo da separare il codice Rust dalla libreria dinamica in CUDA, in modo da poter gestire facilmente le dipendenze e le configurazioni di compilazione.

Listing 6.1: Directory del progetto

```
$ tree .
.
|-- build.rs
|-- Cargo.toml
|-- lib
|   |-- libsolver.so
|   \-- solver.h
\-- src
    |-- main.rs
    |-- solver.rs
    \-- models.rs
```

Il file *Cargo.toml* contiene le dipendenze del progetto, mentre nel file *build.rs* sono presenti le istruzioni necessarie per il linking della libreria dinamica `libsolver.so` di CUDA, specificando l'header da cui recuperare la *signature* delle funzioni con *autocxx*, come mostrato in 6.2. La cartella *src* contiene il codice Rust del microservizio, in cui: nel file *main.rs* è presente il codice principale con le REST API, *solver.rs* contiene le funzioni Rust per interfacciarsi con la libreria dinamica tramite FFI, e *models.rs* le strutture dati necessarie per la computazione e il parsing dei dati JSON inviati dal client.

Listing 6.2: build.rs

```rust
use std::env;
use std::path::PathBuf;

fn main() {
    let manifest = env::var("CARGO_MANIFEST_DIR").unwrap()
    let manifest_dir = PathBuf::from(manifest);
    println!("cargo:rerun-if-changed=src/solver.rs");

    // Define path to resolve #include relative position
    let include_path = manifest_dir.join("lib");
    let cuda_path = manifest_dir.join("/usr/local/cuda/include");

    let mut b = autocxx_build::Builder
        ::new("src/solver.rs", [&include_path, &cuda_path])
        .auto_allowlist(true)
        .build()
        .unwrap();
    b.flag_if_supported("-std=c++17").compile("ms-solver");

    println!("cargo:rustc-link-search=lib");
    println!("cargo:rustc-link-lib=solver");
    println!("cargo:rustc-env=LD_LIBRARY_PATH={}",
        manifest_dir.join("lib").to_str().unwrap());
}
```

Per integrare Rust con la libreria in CUDA si è scelto di utilizzare la libreria *autocxx* che, partendo dai file di header CUDA, genera tramite macro il modulo `ffi` Rust, necessario per richiamare le funzioni della libreria. Questo permette di avere codice Rust aggiornato, in grado di chiamare le funzioni della libreria CUDA e di gestire i puntatori e le strutture dati necessarie per la computazione. Inoltre, *autocxx* permette di gestire automaticamente la conversione dei tipi di dato tra Rust e C++, permettendo di scrivere codice Rust più pulito e mantenibile.

Listing 6.3: Macro autocxx e uso della FFI

```rust
// src/solver.rs

use std::ffi::{CStr, CString};
use crate::models::{Matrix, Model, QuboMatrix, Solutions};

autocxx::include_cpp! {
    #include "solver.h"

    safety!(unsafe_ffi)
    generate!("solve_matrix")
    generate!("health_check")
    generate!("get_device_count")
}

pub fn get_device_count() -> i32 {
    ffi::get_device_count().into()
}

pub fn health_check() -> String {
    unsafe {
        let ptr = ffi::health_check();
        let c_str = CStr::from_ptr(ptr);
        let c_string = CString::from(c_str);

        c_string.into_string().unwrap_or_default()
    }
}

pub fn solve_matrix(params: Matrix<f64>) -> Solutions<f64> {
    ...
    unsafe { let _ = ffi::solve_matrix(params.qubo_matrix, ...); }
    ...
}

// lib/solver.h

char const *health_check() { return "Hello from CUDA!"; }
int get_device_count();
int solve_matrix(...);
```

Per quanto riguarda la parte web, si è scelto di utilizzare il framework *Axum* per la gestione delle REST API, in quanto è un framework molto leggero e performante, che sfrutta la feature *async/await* di Rust. Inoltre, *Axum* permette di gestire facilmente le richieste HTTP e di serializzare e deserializzare i dati in formato JSON tramite l'integrazione automatica con la libreria *Serde*. Gestire le richieste HTTP in modo asincrono e parallelo, permette di avere un numero maggiore di

richieste al server senza che vi siano rallentamenti nell'esecuzione. In caso di configurazioni con più GPU questo può facilitare la gestione delle risorse in modo parallelo, permettendo di sfruttare al meglio le risorse disponibili e diminuire i tempi di risposta per il client. Questo approccio è simile a quanto avverrebbe usando thread e processi, ma è trasparente allo sviluppatore, che può scrivere codice parallelo come se fosse sequenziale, delegando al runtime la gestione dei task in modo efficiente.

Tramite l'uso di macro Rust e le funzioni di *Axum* si può scrivere un microservizio web con pochissimo codice, come mostrato in 6.4. Le strutture dati definite in *models.rs* vengono usate per serializzare e deserializzare i dati in formato JSON in modo automatico, permettendo di gestire facilmente i dati in input e in output delle REST API. È importante notare che, data la mole di dati e parametri che vengono solitamente usati per la computazione di matrici QUBO (con payload nell'ordine della centinaia di MB), è fondamentale gestire il parsing in modo efficiente e altamente prestazionale. Inoltre, *Axum* permette di gestire facilmente i middleware, permettendo di aggiungere funzionalità come il *logging*, la gestione degli errori e la gestione delle autorizzazioni in modo semplice e modulare.

Listing 6.4: Inizializzazione Axum

```rust
#[tokio::main]
async fn main() -> Result<(), anyhow::Error> {
    let address = "127.0.0.1:3000".parse().unwrap();
    println!("=> Running on http://{address}");

    axum::Server::bind(&address)
        .serve(app().into_make_service())
        .await
        .unwrap();

    Ok(())
}

fn app() -> Router {
    Router::new()
        .route("/health-check", get(health_check))
        .route("/device-count", get(get_device_count))
        .route("/solve-matrix", post(solve_matrix))
        .layer(axum::middleware::from_fn(logging))
        .layer(axum::middleware::from_fn(authentication))

}
```

Le REST API sono definite come funzioni *async*, macchine a stati, detti *task*, in cui l'esecuzione è delegata al runtime asincrono *Tokio*. I task vengono ripartiti in più thread usando il modello del *cooperative scheduling* [28], il risultato è che i task collaborano in modo da gestire le richieste e i dati in modo efficiente.

Listing 6.5: Web API

```
async fn health_check()
        -> Result<Json<serde_json::Value>, AppError> {
    let msg = solver::health_check();
    let response = serde_json::json!(msg);

    Ok(Json(response))
}

async fn get_device_count()
        -> Result<Json<serde_json::Value>, AppError> {
    let count = solver::get_device_count();
    let response = serde_json::json!({ "devices": count });

    Ok(Json(response))
}

async fn solve_matrix(Json(payload): Json<Matrix<f64>>)
        -> Result<Json<Solutions<f64>>, AppError> {
    let solutions = solver::solve_matrix(payload);

    Ok(Json(solutions))
}
```

Un esempio di richiesta API è mostrato in 6.6, in cui si invia una matrice QUBO in formato JSON al microservizio, che la computa e restituisce le soluzioni in formato JSON. Questo permette di avere un'interfaccia semplice e standardizzata per l'interazione con il microservizio, permettendo di integrarlo facilmente con altri servizi e applicazioni.

Listing 6.6: Esempio richiesta API

```
$ curl -X POST http://localhost:3000/solve-matrix \
    -H "Content-Type: application/json" \
    -d '{"qubo_matrix": [1, 0, 0, 0, 1, 0, 0, 0, 1], "N": 3,
         "solutionsNumber": 3 }'

{
    "quboSolutions": [
        {
            "quboEnergy": 0.0,
            "quboSolution": [1, 0, 1],,
            "solutionTime": 0.0
        },
        ...
    ]
}
```

Per la verifica della correttezza delle API e dell'integrazione con CUDA sono stati scritti degli *unit test* usando *axum-test-helper* e *cargo test* per l'esecuzione. Un esempio di test è mostrato in 6.7, in cui viene testata la funzione *health_check* per verificare che restituisca il messaggio corretto e che quindi l'integrazione con CUDA sia eseguita correttamente.

Listing 6.7: Health check test

```
#[cfg(test)]
mod tests {
    #[tokio::test]
    async fn test_api_health_check() {
        let client = TestClient::new(app());
        let res = client.get("/health-check").send().await;

        assert_eq!(res.status(), StatusCode::OK);
        assert_eq!(
            res.json::<String>().await,
            serde_json::json!("Hello from CUDA!")
        );
    }
    ...
}
```

La compilazione e il rilascio sono gestiti tramite *cargo* e *docker*, permettendo di avere un sistema di *build* e *deploy* automatizzato e riproducibile. L'uso della libreria dinamica, in questo caso, può essere utile, per aggiornare le funzioni CUDA senza dover ricompilare l'intero microservizio, così da aumentare il riutilizzo del

codice. È comunque possibile anche usare una libreria statica, per avere un sistema più portabile e indipendente dalle librerie esterne. Integrando la compilazione tra *cargo* e NVCC, e impostando conseguentemente il linking dell'eseguibile finale in *build.rs* si può generare un singolo binario che contenga sia il codice Rust che CUDA. Mi sento di consigliare questo approccio per avere un sistema più robusto e mantenibile, in quanto permette di avere un controllo maggiore sulle dipendenze e di evitare problemi di compatibilità e versioning delle librerie.

6.2 Sviluppi futuri

Possibili migliorie per rendere il microservizio ancora più efficiente potrebbe essere l'uso di *gRPC* al posto delle REST API, in quanto permette di avere una comunicazione più efficiente e performante, grazie al protocollo binario e alla gestione automatica della serializzazione e deserializzazione dei dati. Inoltre, *gRPC* permette di definire i servizi e i messaggi tramite *Protocol Buffers*, che permette di avere una definizione standardizzata e facilmente mantenibile dei dati e delle API. Questo permetterebbe di avere un sistema più scalabile e performante, in grado di gestire un numero maggiore di richieste e in modo più efficiente. Questo approccio permette di richiamare le API da altre macchine in modo più facile, integrando il modo più solido il microservizio nell'architettura software aziendale.

Data l'architettura modulare del microservizio, è possibile estenderlo facilmente aggiungendo nuove API e nuove funzionalità, come la gestione di più GPU, la gestione di più matrici QUBO in parallelo, o la possibilità di integrare le API Vulkan per sostituire alcune funzioni CUDA, oppure combinare i due framework e usarli in modo interscambiabile in base alle esigenze. Sarebbe, teoricamente, anche possibile avere un sistema multi-GPU con hardware di vendor differenti ed usare Vulkan per GPU AMD e CUDA per GPU NVIDIA, parallelamente.

Capitolo 7

Conclusioni

L'obiettivo di questo lavoro di tesi era la comparazione delle performance di due sistemi per il GPGPU computing, CUDA e Vulkan. Come dimostrato dai dati e dall'esperienza di sviluppo, per quanto CUDA è ancora la prima scelta per sistemi di computazione eterogenea per GPGPU, Vulkan è un'alternativa valida, in molti casi preferibile, e, per sistemi che non supportano CUDA, l'unica scelta valida che abbia prestazioni paragonabili. Dato che le performance di Vulkan sono molto vicine a quelle di CUDA, la scelta tra i due dipende da altri fattori: la complessità del codice, la disponibilità di librerie e la facilità di sviluppo. Inoltre, Vulkan è una tecnologia più recente, rispetto a CUDA, che continua ad evolversi e migliorarsi, quindi è possibile che in futuro diventi la tecnologia preferita per il GPGPU computing.

La scelta di usare Rust in combinazione con Vulkan si è rivelata corretta. Rust è un linguaggio di programmazione moderno e sicuro, che permette di scrivere codice efficiente e mantenibile, senza rinunciare alle performance. Il vasto ecosistema di librerie e documentazione che si è sviluppato attorno a Rust è un ulteriore vantaggio per gli sviluppatori, soprattutto per quanto riguarda lo sviluppo web. Grazie alla FFI di Rust anche l'integrazione con CUDA è stata agevole e, al netto della preliminare fase di configurazione di building e linking delle librerie CUDA, non ci sono stati impedimenti di sorta. Grazie alla politica *zero cost abstraction* di Rust, l'integrazione con CUDA è stata molto efficiente senza la presenza di alcun overhead nell'applicativo.

In futuro, sarebbe interessante esplorare le performance di entrambi i framework su sistemi multi-GPU, per vedere se Vulkan è in grado di sfruttarne in modo migliore l'hardware rispetto a CUDA. L'idea di poter anche usare Vulkan in combinazione con CUDA per sfruttare le potenzialità di entrambe le tecnologie potrebbe portare a risultati molto interessanti. L'ostacolo principale che, ad oggi, impedisce il diffondersi di Vulkan per la GPGPU è verbosità nello scrivere codice: lnizializzare delle risorse Vulkan, impostare il trasferimento di memoria con i

buffer e sincronizzare i comandi, di certo inficia la scrittura di codice facilmente comprensibile e mantenibile. Sarebbe interessante sviluppare librerie e framework per semplificare lo sviluppo di applicazioni GPGPU con Vulkan, in modo da rendere questa tecnologia più accessibile agli sviluppatori. L'idea di poter compilare Rust direttamente in SPIR-V, senza passare per il GLSL, ed eseguire il codice direttamente su Vulkan, potrebbe essere la killer feature che renderebbe Vulkan una valida alternativa a CUDA per il GPGPU computing.

Glossario

FFI

Foreign Function Interface

API

Application Program Interface

CAD

Computer Aided Design

CPU

Central Processing Unit

CUDA

Compute Unified Device Architecture

GPGPU

General-purpose computing on graphics processing units

GPU

Graphic Processing Unit

HPC

High Performance Computing

QUBO

Quadratic Unconstrained Binary Optimization

MIMD

Multiple Instruction Multiple Data

SIMD

Single Instruction Multiple Data

SIMT

Single Instruction Multiple Thread

GDDR

Graphics Double Data Rate

SDRAM

Synchronous DRAM

NVCC

NVIDIA C Compiler

SPMD

Single Program Multiple Data

RAII

Resource Acquisition Is Initialization

GLSL

OpenGL Shading Language

AOT

Ahead-of-time

FMA

Fused Multiply-Add

LSP

Language Server Protocol

SPIR-V

Standard Portable Intermediate Representation

PCIe

Peripheral Component Interconnect Express

JSON

Javascript Object Notation

CSV

Comma Separated Values

HTTP

HyperText Transfer Protocol

Bibliografia

[1] *Legge di Moore*. URL: https://www.treccani.it/enciclopedia/legge-di-moore_(Enciclopedia-della-Scienza-e-della-Tecnica)/ (cit. a p. 1).

[2] *Top500 Tianhe-1A*. 2011. URL: https://www.top500.org/system/176929/ (cit. a p. 2).

[3] *NVIDIA Powers Titan, World's Fastest Supercomputer For Open Scientific Research*. 2012. URL: https://nvidianews.nvidia.com/news/nvidia-powers-titan-world-s-fastest-supercomputer-for-open-scientific-research-6622738 (cit. a p. 2).

[4] *Powered by AMD, Frontier Expected to be World's Fastest Supercomputer*. 2022. URL: https://www.amd.com/en/products/frontier (cit. a p. 3).

[5] IEEE Spectrum. *Move Over, Moore's Law: Make Way for Huang's Law*. 2018. URL: https://spectrum.ieee.org/move-over-moores-law-make-way-for-huangs-law (cit. a p. 3).

[6] *NVIDIA. Compute unified device architecture (CUDA) programming guide*. Ver. 12.2. 2023. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ (cit. a p. 4).

[7] *Open Standard for Parallel Programming of Heterogeneous Systems*. URL: https://www.khronos.org/opencl/ (cit. a p. 4).

[8] *Vulkan Guide*. URL: https://docs.vulkan.org/guide/latest/index.html (cit. a p. 4).

[9] Khronos Group. *The Industry Open Standard Intermediate Language for Parallel Compute and Graphics*. URL: https://www.khronos.org/spir/ (cit. a p. 4).

[10] *Safe Rust wrapper around the Vulkan API*. URL: https://vulkano.rs/ (cit. a p. 5).

[11] *NVIDIA CUDA Compiler Driver NVCC*. Ver. 12.2. 2023. URL: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html (cit. a p. 5).

[12] Khronos Group. *The OpenGL® Shading Language, Version 4.60.8*. URL: https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.60.pdf (cit. a p. 5).

[13] Michael J. Flynn. «Some Computer Organizations and Their Effectiveness». In: 21 (set. 1972), pp. 948–960 (cit. a p. 8).

[14] David B. Kirk e Wen-mei Hwu. *Programming Massively Parallel Processors*. Elsevier Inc, 2013. Cap. 1, 5, 6 (cit. alle pp. 10, 43).

[15] cppreference.com. *RAII pattern in C++*. 2023. URL: https://en.cppreference.com/w/cpp/language/raii (cit. a p. 18).

[16] Steve Klabnik, Carol Nichols e contributions from the Rust Community. *References and Borrowing*. 2023. URL: https://doc.rust-lang.org/1.74.0/book/ch04-02-references-and-borrowing.html (cit. alle pp. 18, 26).

[17] G. Sellers e J. Kessenich. *Vulkan Proramming Guide*. Person Education, 2016 (cit. a p. 21).

[18] Khronos Group. *Vulkan® 1.3.278 - A Specification (with all registered extensions)*. URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html#VkMemoryPropertyFlagBits (cit. a p. 21).

[19] Gavin Thomas e MSRC. *A proactive approach to more secure code*. 2019. URL: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/ (cit. a p. 26).

[20] Benchmarks Game. *Rust versus C gcc fastest performance*. 2023. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust-gcc.html (cit. a p. 27).

[21] Sara Verdi. *Why Rust is the most admired language among developers*. 2023. URL: https://github.blog/2023-08-30-why-rust-is-the-most-admired-language-among-developers/ (cit. a p. 28).

[22] The White House. *Back to the Building Blocks: A Path Toward Secure and Measurable Software*. 2024. URL: https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf (cit. a p. 28).

[23] Alex Fit-Florea Nathan Whitehead. *Precision and Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs*. NVIDIA. 2022. URL: https://developer.nvidia.com/sites/default/files/akamai/cuda/files/NVIDIA-CUDA-Floating-Point.pdf (cit. a p. 39).

[24] *NVIDIA Nsight Visual Studio Code Edition*. Ver. 2023.2. 2023. URL: https://developer.nvidia.com/nsight-visual-studio-code-edition (cit. a p. 50).

[25] LunarG. *LunarG: Creator and Curator of the Vulkan SDK*. URL: https://www.lunarg.com/vulkan-sdk/ (cit. a p. 50).

[26] gfx-rs. *rspirv module*. URL: https://github.com/gfx-rs/rspirv (cit. a p. 51).

[27] *A Comprehensive Suite of Compilers, Libraries and Tools for HPC*. 2024. URL: https://developer.nvidia.com/hpc-sdk (cit. a p. 52).

[28] *Reducing tail latencies with automatic cooperative task yielding*. URL: https://tokio.rs/blog/2020-04-preemption (cit. a p. 58).

[29] Nicola Capodieci e Roberto Cavicchioli. «vkpolybench: A crossplatform Vulkan Compute port of the PolyBench/GPU benchmark suite». In: (ago. 2021).

[30] Trinayan Baruah. «Improving the Virtual Memory Efficiency of GPUs». Tesi di dott. Northeastern University, 2021.

[31] Chiara Varini. «Accelerating large data modeling for quantum computation with GPUs». Tesi di laurea mag. Università di Bologna, 2020.

[32] Andrea Marchese. «Sviluppo e implementazione di algoritmi paralleli in ambiente CUDA per la ricostruzione tridimensionale densa dell'ambiente». Tesi di laurea mag. Università di Padova, 2011.

[33] Juuso Haavisto. «Leveraging APL and SPIR-V languages to write network functions to be deployed on Vulkan compatible GPUs». Tesi di laurea mag. University of Lorraine, 2021.

[34] BodurriK Lajdi. «Comparative performance analysis of Vulkan and CUDA programming model implementations for GPUs». Tesi di laurea mag. University of Thessaly, 2019.

[35] Fanfu Meng. «Analyzing General-Purpose Computing Performance on GPU». Tesi di laurea mag. California Polytechnic State University, 2015.