

ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA  
CAMPUS DI CESENA

---

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA  
Corso di Laurea in Ingegneria e Scienze Informatiche

ANALISI DELLE PRESTAZIONI  
DEL SISTEMA GRAFICO VIDEOCORE IV  
APPLICATO AL CALCOLO GENERICO

*Elaborato in*  
HIGH PERFORMANCE COMPUTING

*Relatore*  
Prof. MORENO MARZOLLA

*Presentata da*  
SIMONE MAGNANI

Anno Accademico 2018 – 2019



# Indice

<b>Introduzione</b>	<b>v</b>
<b>1 Il calcolo Parallelo</b>	<b>1</b>
1.1 Classificazione elaboratori . . . . .	1
1.2 Valutazione di un programma parallelo . . . . .	2
1.2.1 Speedup . . . . .	2
1.2.2 Scalabilità forte . . . . .	3
1.2.3 Scalabilità debole . . . . .	3
1.3 Sviluppo di programmi paralleli . . . . .	4
1.3.1 Confronto tra CPU e GPU . . . . .	4
1.3.2 Esempi di linguaggi per lo sviluppo . . . . .	4
<b>2 Raspberry Pi</b>	<b>9</b>
2.1 Panoramica . . . . .	10
2.2 Programmazione GPU . . . . .	10
2.3 QPULib . . . . .	11
2.3.1 Confronto tra QPULib e CUDA-C . . . . .	11
2.3.2 Programmare con QPULib . . . . .	12
<b>3 Kernel computazionali</b>	<b>15</b>
3.1 Earthquake . . . . .	15
3.1.1 Modello Earthquake . . . . .	15
3.1.2 Implementazione QPULib . . . . .	16
3.2 Moltiplicazione tra matrici . . . . .	18
3.2.1 Modello Matmul . . . . .	18
3.2.2 Implementazione QPULib . . . . .	18
<b>4 Valutazioni delle prestazioni</b>	<b>21</b>
4.1 OpenMP Earthquake . . . . .	21
4.2 Matmul con Vector data type . . . . .	22
4.3 Prestazioni QPULib e confronto . . . . .	23
4.3.1 Scalabilità dato un input costante . . . . .	23

4.3.2	Scalabilità debole . . . . .	24
4.3.3	Confronto con altre implementazioni . . . . .	25
	<b>Conclusioni</b>	<b>27</b>
	<b>Ringraziamenti</b>	<b>29</b>

# Introduzione

Il calcolo parallelo rappresenta una risorsa per ridurre i tempi di esecuzione di un qualsiasi problema numerico.

Storicamente i programmi erano scritti per essere eseguiti sequenzialmente da una sola unità di calcolo, ad esempio una singola CPU. Le singole unità di calcolo però non erano sempre sufficientemente potenti per eseguire, in un tempo consono, una grande quantità di operazioni.

L'idea alla base del calcolo parallelo è di suddividere il lavoro su più unità di calcolo in modo da ridurre il tempo necessario per svolgere la computazione.

Portando un esempio più vicino alla quotidianità di tutti, è come avere un lavoro che necessita di molto tempo per essere svolto, così si decide di suddividere il lavoro in sotto-problemi e assegnare ognuno di questi a una persona.

Il calcolo parallelo mira proprio a una gestione di problemi complessi dividendoli in sotto-problemi, poi facendo in modo che le unità di calcolo designate collaborino ed eseguano la propria parte per trovare la soluzione.

Esistono due macro tipi di unità di calcolo: CPU e GPU; le prime sono più veloci ma con meno possibilità di parallelizzazione, al contrario le GPU sono generalmente più lente ma con un numero di ALU<sup>1</sup> molto maggiore.

In particolare nel capitolo 4 verrà confrontato l'uso della GPU rispetto alla CPU su un Raspberry Pi (capitolo 2): ovvero una scheda *single-board* economica, ma abbastanza potente da essere un buono strumento per un programmatore esperto, e un ottimo strumento per chi vuole avvicinarsi alla programmazione parallela.

---

<sup>1</sup>Arithmetic-Logic Unit è l'unità di calcolo vera e propria



# Capitolo 1

## Il calcolo Parallelo

La programmazione parallela è una forma ormai ben nota e diffusa di approccio ai problemi numerici, soprattutto in ambiti scientifici dove è necessario lavorare con un'importante mole di dati.

In questo capitolo verranno descritti quali sono le varie classificazioni di elaboratori, come valutare le prestazioni dei programmi paralleli e qualche possibilità per svilupparli.

### 1.1 Classificazione elaboratori

Per quanto riguarda la classificazione dei calcolatori, non esiste uno standard universalmente riconosciuto, in generale si utilizza la tassonomia di Flynn del 1972. Quest'ultima utilizza due caratteristiche principali per raggruppare i diversi tipi di macchine: il numero di istruzioni computate parallelamente e le sequenze di dati che possono essere elaborate concorrentemente.

Da questa divisione nascono 4 principali tipologie di elaboratori:

- SISD (Single Instruction-Single Data): è il caso classico dell'architettura di Von Neumann che segue il ciclo fetch-decode-execute eseguendo serialmente un'istruzione su un singolo flusso di dati.
- SIMD (Single Instruction-Multiple Data): calcolatori che possono eseguire contemporaneamente la stessa istruzione su più dati, ad esempio le GPU.
- MISD (Multiple Instruction-Single Data): macchine che supportano l'elaborazione di un singolo dato attraverso più istruzioni concorrentemente; non esistono ancora hardware di importanza rilevante appartenenti a questa categoria.

- MIMD (Multiple Instruction-Multiple Data): sistemi in grado di eseguire più istruzioni differenti su dati differenti, l'architettura sottostante può essere costituita da core autonomi che lavorano indipendentemente su diversi elementi: le CPU appartengono ormai tutte a questa categoria.

## 1.2 Valutazione di un programma parallelo

Per valutare le prestazioni di un programma parallelo è necessario definire tre concetti principali: lo Speedup, la scalabilità forte e la scalabilità debole.

### 1.2.1 Speedup

Lo Speedup si definisce come il rapporto tra il tempo impiegato dal programma seriale per risolvere il problema ( $T_{serial}$  o  $T(1)$ ) e il tempo del programma parallelo con  $p$  processi/core ( $T_{parallel}(p)$  o  $T(p)$ ).

$$S(p) = \frac{T_{serial}}{T_{parallel}(p)} \quad (1.1)$$

Nel caso ideale si ipotizza che  $T_{parallel}(p) = T_{serial}/p$  quindi avremo  $S(p) = p$  (quando questa condizione si avvera, si parla di **Speedup lineare**). È normale pensare che  $p$  sia il valore massimo dello Speedup, ma ci sono casi in cui si può ottenere  $S(p) > p$ , ovvero uno Speedup superlineare: parallelizzando il problema è possibile che ci siano meno controlli da effettuare, quindi il lavoro totale e di conseguenza il tempo di esecuzione diminuiscono.

Normalmente esistono parti del programma che non possono essere parallelizzate, più in particolare definendo  $\alpha$  come la frazione di tempo del programma non parallelizzabile avremo che:

$$T_{parallel}(p) = \alpha T_{serial} + \frac{(1 - \alpha)T_{serial}}{p} \quad (1.2)$$

dato  $0 \leq \alpha \leq 1$  si evince

$$T_{parallel}(p) \geq T_{serial}/p \quad (1.3)$$

sostituendo l'ultima formula (eq. 1.3) alla definizione di Speedup (eq. 1.1) e raccogliendo  $T_{serial}$  si ottiene la legge di Amdahl (eq. 1.4), che definisce il massimo Speedup come

$$S(p) = \frac{1}{\alpha + \frac{(1 - \alpha)}{p}} \quad (1.4)$$



da cui si capisce che  $\frac{1}{\alpha}$  è un limite massimo per lo Speedup di programmi non completamente parallelizzabili, infatti considerando  $p \rightarrow \infty$

$$S(p) = \lim_{p \rightarrow \infty} \frac{1}{\alpha + \frac{(1-\alpha)}{p}} = \frac{1}{\alpha} \quad (1.5)$$

### 1.2.2 Scalabilità forte

Per studiare la scalabilità forte si confrontano i tempi di esecuzione mantenendo costante il carico di lavoro totale e modificando solo il numero di processi  $p$ ; la scalabilità forte dipende direttamente dallo Speedup:

$$E(p) = \frac{S(p)}{p} = \frac{T(1)}{pT(p)} \quad (1.6)$$

In questo caso vengono confrontati il tempo impiegato da  $p$  processori per eseguire l'algoritmo e il tempo di un singolo processore. Se all'aumentare di  $p$  il valore di  $E$  risulta costante allora si dice che l'algoritmo è scalabile.

Il valore ideale della scalabilità è 1, infatti a questo valore corrisponde il pieno utilizzo delle risorse. Generalmente al crescere di  $p$  si ha un incremento dell'overhead dovuto alla sincronizzazione tra i processi, quindi una diminuzione di  $E(p)$ . Considerando la formula 1.6 e sapendo che esiste il caso di Speedup superlineare, è ovvia la possibilità che  $E(p) \geq 1$

### 1.2.3 Scalabilità debole

La scalabilità debole considera i diversi tempi di esecuzione dei programmi in cui viene mantenuto fisso il carico di lavoro per processo  $p$  cambiando sia  $p$  che la dimensione del problema.

$$W(p) = \frac{T_1}{T_p} \quad (1.7)$$

Dove  $T_1$  è il tempo di esecuzione di una unità di lavoro svolto da un processore, invece  $T_p$  è il tempo di  $p$  unità di lavoro svolto da  $p$  processori.

Anche in questo caso il valore ideale della scalabilità è 1, nonostante esista la possibilità che questo venga superato.

## 1.3 Sviluppo di programmi paralleli

Prima di iniziare lo sviluppo di codice parallelo è necessario scegliere le tecnologie hardware e software attentamente. L'esecuzione dei programmi avrà rendimenti diversi in base alle tecnologie su cui si basa: in particolare per alcune computazioni l'uso della CPU o della GPU può portare a importanti differenze nei tempi di calcolo.

### 1.3.1 Confronto tra CPU e GPU

Parlando di programmi paralleli ad alta velocità possiamo parlare in generale di due unità di calcolo hardware: CPU e GPU.

La CPU è composta oggi da massimo una decina di core, questi lavorano ad alta frequenza (nell'ordine dei 2.5-4 GHz) per effettuare calcoli e coordinare il resto delle periferiche della macchina. Storicamente tutti i programmi erano mandati in esecuzione sulla CPU, questo è il motivo per cui ancora oggi la maggior parte degli eseguibili e dei linguaggi di programmazione sono specifici per CPU.

La GPU presenta un numero di core molto maggiore rispetto alla CPU (da centinaia fino a migliaia) che lavorano a frequenze inferiori (circa 1-2 GHz) e sono specializzati nella grafica digitale, operando SIMD su una grande mole di numeri in virgola mobile. Per questa caratteristica è nata l'idea di usare queste unità di calcolo anche per scopi non strettamente legati alla grafica ed è stato coniato il nome di **GPGPU** (*General-Purpose GPU*).

L'uso della programmazione per CPU piuttosto che per GPU dipende solitamente dal progetto che si deve svolgere: la CPU è più veloce nella gestione di operazioni complesse su pochi dati, al contrario la GPU può gestire una quantità di dati molto maggiore a discapito della complessità del programma e della precisione (alcune GPU consentono di eseguire calcoli solamente con i float).

### 1.3.2 Esempi di linguaggi per lo sviluppo

Ci sono vari linguaggi per lo sviluppo di programmi paralleli, in particolare per la programmazione CPU vengono descritti `OpenMP` e i `Vector data type` che verranno riutilizzati anche nel capitolo 4 per il confronto con altre versioni.

## OpenMP

OpenMP [1] è un modello per la programmazione parallela in architetture a memoria condivisa; il modello si basa su direttive del compilatore per permettere al programmatore di dividere le parti seriali da quelle parallele e per gestire le sincronizzazioni dei thread.

Per usare OpenMP non è necessario essere esperti della programmazione parallela: modificando il codice seriale tramite le direttive adeguate si ottiene subito un livello base di parallelizzazione. OpenMp definisce la direttiva `#pragma omp parallel` per delimitare le sezioni di codice parallele: in particolare è possibile parallelizzare dei cicli `for` che rispettano alcune condizioni tramite la `#pragma omp parallel for`.

Ogni direttiva può essere integrata con delle clausole che possono limitare o meno la visibilità delle variabili (`shared`, `private`, `default`), specificano come gestire il carico di lavoro (`schedule`, `collapse`) o effettuano operazioni sui dati (`reduction`).

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[] )
{
    int n = 1024, sum = 0, *v1 = (int*)malloc(n * sizeof(v1[0]));
    //fill the array
    for (int i=0; i<n; i++) {
        v1[i] = i%4;
    }

    #pragma omp parallel for reduction (+:sum)
    for (int i=0; i<n; i++) {
        sum += v1[i];
    }

    printf("sum = %d",sum);
    free(v1);
    return EXIT_SUCCESS;
}
```

Listato 1.1: Riduzione di un vettore OpenMP

Come possiamo vedere dal codice (Listato 1.1) il ciclo viene eseguito in modo parallelo, e ogni thread calcola la sua somma provvisoria `sum` che viene

a sua volta usata per calcolare la somma finale dalla `reduction`; infine viene stampato una sola volta il risultato finale.

## Programmazione SIMD e GCC Vector data type

La programmazione SIMD sfrutta appositi registri e unita di calcolo per effettuare una singola operazione su più elementi durante lo stesso ciclo di clock. Ogni architettura presenta un suo linguaggio specifico per la programmazione SIMD, questo induce il codice a non essere portabile; per evitare questo si possono usare estensioni basate sul compilatore: nel caso di GCC, i  `GCC Vector data type`  [6].

Come suggerito dal nome i `Vector data type` sono vettori di elementi sui quali si possono applicare le più comuni operazioni aritmetiche (somma, prodotto..).

Il compilatore, quando viene richiamato, emette le istruzioni SIMD compatibili per l'architettura sottostante, oppure il codice viene compilato per l'esecuzione seriale nel caso l'hardware non supporti la programmazione SIMD.

I `Vector data type` possono essere usati come i classici array in C; per usarli però vanno prima definiti attraverso la `typedef`. Esiste una convenzione per i nomi dei vettori: è buona norma che il nome inizi sempre con la lettera `v`, dopodiché vengono inseriti il numero di elementi del vettore e una lettera che rappresenta il tipo degli elementi; ad esempio i `v4i` sono vettori di 4 `int`.

```
/* The following #define is required by posix_memalign() */
#define _XOPEN_SOURCE 600

#include "hpc.h"
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef int v4i __attribute__((vector_size(16)));
#define VLEN (sizeof(v4i)/sizeof(int))

int main(int argc, char* argv[])
{
    int n = 1024, sum = 0, *v;
    v4i vsum = {0,0,0,0}, *pv;

    int ret = posix_memalign((void**)&v, __BIGGEST_ALIGNMENT__, n *
        sizeof(*v));
    assert( 0 == ret );
    //fill the array
```

```
for(int i=0; i<n; i++){
    v[i]=i%4;
}
//calculating partial sum
for(int i=0; i<n-VLEN+1; i+=VLEN) {
    pv = (v4i*)(v+i);
    vsum += *pv;
}
//calculating sum
for(int i=0; i<VLEN; i++){
    sum += vsum[i];
}

printf("sum = %d\n", sum);
free(v);
return EXIT_SUCCESS;
}
```

Listato 1.2: Codice SIMD

Il codice (Listato 1.2) effettua le stesse operazioni di quello precedente implementato in OpenMP (Listato 1.1): ogni elemento del v4i `vsum` calcola la propria somma parziale tramite le operazioni SIMD, dopodiché viene calcolata serialmente la somma finale.



## Capitolo 2

# Raspberry Pi

*La nostra missione è mettere il potere dell'informatica e del calcolo digitale nelle mani delle persone di tutto il mondo [5].*

Questo è lo slogan di Raspberry, per presentare *Il computer delle dimensioni di una carta di credito che costa 25 sterline [5]: Raspberry Pi.*



Figura 2.1: Confronto tra le dimensioni di Raspberry Pi e una carta di credito. (Fonte: [3])

## 2.1 Panoramica

Raspberry Pi è una classe di calcolatori *Single-Board* che presenta alcune caratteristiche comuni: una CPU ARM, una GPU con innestato il chip VideoCore IV, un supporto per la scheda MicroSD, ingressi USB 2.0 o 3.x e ingressi Ethernet che vanno dai 10/100 Mbit/s fino a 1Gbit/s.

Le CPU ARM installate sono in grado di gestire sistemi operativi tra cui Raspbian e Ubuntu (considerati i più rilevanti). Nonostante nelle prime versioni le CPU non fossero molto performanti (700MHz single-core fino al Raspberry 2), Raspberry Pi presentava GPU con capacità di calcolo molto promettenti, considerando anche il ridotto consumo di energia (massimo  $\sim 5W$ ).

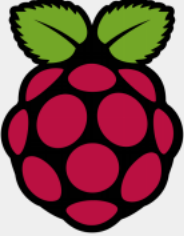
	Raspberry Pi 3 Model B	Raspberry Pi Zero	Raspberry Pi 2 Model B	Raspberry Pi Model B+
Introduction Date	2/29/2016	11/25/2015	2/2/2015	7/14/2014
SoC	BCM2837	BCM2835	BCM2836	BCM2835
CPU	Quad Cortex A53 @ 1.2GHz	ARM11 @ 1GHz	Quad Cortex A7 @ 900MHz	ARM11 @ 700MHz
Instruction set	ARMv8-A	ARMv6	ARMv7-A	ARMv6
GPU	400MHz VideoCore IV	250MHz VideoCore IV	250MHz VideoCore IV	250MHz VideoCore IV
RAM	1GB SDRAM	512 MB SDRAM	1GB SDRAM	512MB SDRAM
Storage	micro-SD	micro-SD	micro-SD	micro-SD
Ethernet	10/100	none	10/100	10/100

Figura 2.2: Confronto delle specifiche di alcuni modelli Raspberry Pi (Fonte: [2])

## 2.2 Programmazione GPU

All'interno della GPU di Raspberry, sono presenti 12 QPU (Quad Processing Units), ossia 12 unità costruite per processare vettori da 16 elementi in 4 cicli di clock, che rendono la GPU potenzialmente in grado di elaborare fino a 24 GFLOPS<sup>1</sup> fin dai primi modelli di Raspberry Pi.

La comunità scientifica si è mostrata interessata a questo vantaggioso rapporto *capacità di calcolo/consumo di energia* da quando Raspberry Pi è stato

<sup>1</sup>Un GFLOPS equivale a  $10^9$  operazioni in virgola mobile al secondo



presentato al pubblico (2012). Per qualche anno non è stato possibile programmare la GPU per scopi diversi dalla sola grafica: i classici metodi di programmazione infatti, non erano disponibili. Nel 28 febbraio 2014 Raspberry e l'azienda costruttrice della scheda video pubblicarono la documentazione completa della VideoCore IV [12]; da quella data sono stati sviluppati diversi esempi di programmazione GPU Raspberry Pi per vari livelli di linguaggio:

- GPU\_FFT [7]: un programma scritto direttamente usando il codice assembly della GPU Raspberry, è la prima diretta conseguenza delle pubblicazioni sopra descritte.
- PyVideoCore [11]: una libreria Python che permette una migliore gestione dei dati, ma comunque limita l'uso della GPU alla programmazione attraverso il suo assembly.
- QPULib [10]: implementa tramite una libreria C++ la possibilità di compilare funzioni per la GPU, in modo da poterle invocare per delegare una parte del lavoro.

## 2.3 QPULib

Come già accennato, QPULib è un linguaggio di programmazione per le QPUs di Raspberry Pi ed è implementato come una libreria in C++ che viene eseguita dalla CPU ARM di Raspberry Pi. Per valutare QPULib è utile compararlo con un linguaggio simile: CUDA-C.

### 2.3.1 Confronto tra QPULib e CUDA-C

CUDA-C [4] è un linguaggio proprietario NVIDIA nato nel 2006 per rendere *general-purpose* le GPU compatibili; come QPULib genera programmi in esecuzione sulla CPU i quali demandano le computazioni più onerose alla GPU. Sotto questo punto di vista QPULib e CUDA-C sono simili, infatti una volta inizializzati i dati nella memoria della GPU, è permesso compilare e invocare le funzioni scritte per quest'ultima.

CUDA-C può contare sulla storia di NVIDIA e su un maggior numero di anni di evoluzione: questo rende l'ambiente di sviluppo e il linguaggio meno inclini alla presenza di errori non documentati; al contrario di QPULib che non può dare alcuna garanzia vista anche la versione sicuramente meno sviluppata 0.1.0.

Entrambi i linguaggi vengono eseguiti su hardware specifici, però a vantaggio di QPULib, Raspberry è più economico di una qualsiasi GPU NVIDIA compatibile.

Non da tutti classificato come vantaggio o svantaggio, bisogna considerare che QPULib è open-source, e in quanto tale può essere ritenuto più sicuro oltre che aggiornabile e personalizzabile da chiunque; al contrario CUDA-C essendo scritto specificatamente per hardware costruito da NVIDIA potrebbe essere più veloce. Purtroppo non esistono GPU compatibili a entrambi i linguaggi, quindi non è possibile confrontarli direttamente.

### 2.3.2 Programmare con QPULib

Per programmare usando QPULib bisogna innanzitutto prendere confidenza con i tipi di dato, come gestirli e con le funzioni principali che possono essere utilizzate all'interno del codice che verrà compilato per la GPU.

Le QPUs possono lavorare con solo tre tipi di dato `Int`, `Float` e `Ptr`, i primi due non sono altro che vettori da 16 elementi di 32-bit interpretati come interi o come numeri in virgola mobile, mentre l'ultimo è un puntatore ad un indirizzo di memoria. Parlando di indirizzi di memoria e sapendo che non esiste una memoria condivisa per CPU e GPU, è necessario introdurre la classe degli `SharedArray`: una classe utilizzabile solo dal codice in esecuzione sulla CPU, ma che alloca un vettore in memoria in modo che questa sia accessibile da entrambe le unità di calcolo.

È importante precisare che le QPUs non possono lavorare che con questi tre tipi di dato (quindi non sarà possibile salvare un solo `int`, ma sarà necessario salvare un `Int`) ed è necessario calcolarlo quando si alloca lo spazio per le strutture dati che conterranno questi dati.

La CPU ARM di Raspberry Pi implementa lo standard IEEE 754 [8] che impone, tra le altre specifiche, di approssimare il numero per eccesso o per difetto in base alle cifre meno significative; QPULib al contrario, approssima sempre per difetto. Nonostante questo possa sembrare irrilevante in molti problemi, è importante saperlo e considerarlo nel caso le operazioni da svolgere rielaborino iterativamente i propri output. In particolare, nel nostro caso renderà impossibile il confronto diretto dell'output dei programmi (capitolo 4).

#### Funzioni principali

Per quanto riguarda i costrutti principali della programmazione C, QPULib implementa delle funzioni nascoste dietro a delle macro, per agevolare la programmazione della GPU:

- **For**: simile al costrutto per il ciclo `for`, deve essere richiamato usando le virgole al posto dei punti e virgola, inoltre si può inizializzare la variabile su cui iterare ricordandosi che deve essere `Int` o `Float`.

- **Print**: serve principalmente in fase di sviluppo per scrivere sullo stdout delle stringhe, degli `Int` o, modificando a dovere la libreria, dei `Float`.
- **If**: questa funzione implementa l'`if` del C, assegnando all'espressione che viene passata al suo interno un solo valore (vero o falso), in modo da decidere il flusso di istruzioni da seguire.
- **Where**: a differenza dell'`If` questa funzione assegna ad ognuno dei 16 elementi del vettore risultato un valore (vero o falso), quindi solo gli elementi che soddisfano l'espressione accedono all'area di codice all'interno del `Where`.

### Gestione dei dati

Il caricamento e il salvataggio dei dati in un vettore (o in una matrice) può essere effettuato usando la sintassi d'accesso del C++ con operazioni bloccanti; per rendere gli accessi alla memoria più veloci QPULib implementa funzioni non bloccanti:

- la `gather` serve per iniziare il caricamento del dato, questa infatti, inserisce in modo asincrono in una coda FIFO (First In First Out) i primi 16 elementi (un `Int` o un `Float`) a partire dall'indirizzo di memoria specificato come argomento della funzione.
- La `receive` recupera il primo dato (`Int` o `Float`) della coda FIFO dedicata e lo inserisce nella variabile passata come parametro alla funzione; questa viene considerata l'operazione complementare della `gather`.
- La `store` prende in input un `Int` o un `Float` e un indirizzo di memoria, dopodiché salva il primo input all'interno del secondo. Attualmente può essere presente solo una `store` in esecuzione, per questo la funzione è considerata ancora in fase di sviluppo.

Se ne vengono richiamate due di fila, la seconda risulta bloccante fino al completamento della prima.



# Capitolo 3

## Kernel computazionali

L'obiettivo di questa tesi è la valutazione delle implementazioni tramite QPULib di due kernel computazionali<sup>1</sup>: Earthquake e Matmul.

### 3.1 Earthquake

Lo scopo del progetto Earthquake è l'implementazione di un semplice modello matematico di propagazione dei terremoti. Il modello che consideriamo è una estensione in due dimensioni dell'automa cellulare Burridge-Knopoff (BK) [9]. È importante studiare e valutare questo kernel come esempio per un qualsiasi pattern stencil<sup>2</sup>.

#### 3.1.1 Modello Earthquake

Simuleremo una porzione di crosta terrestre di dimensioni  $n \times n$  celle; ogni cella contiene il valore dell'energia potenziale relativamente alla propria posizione.

In accordo con il modello delle placche [13], l'energia aumenta ad ogni iterazione a causa del movimento continuo delle zolle; poi quando questa supera una soglia data, si scarica nelle celle adiacenti emulando un terremoto.

Più nello specifico inizializzeremo una matrice quadrata di dimensione  $n \times n$  di valori reali casuali (float) per emulare la crosta terrestre. Ogni cella rappresenta una porzione di crosta terrestre e il valore ad essa associato è l'energia potenziale; il valore viene aggiornato ad istanti discreti di tempo  $t = 0, 1, 2, \dots$  seguendo due regole: incremento e propagazione.

---

<sup>1</sup>viene chiamato kernel computazionale una piccola funzione ricorrente nell'esecuzione del programma

<sup>2</sup>La computazione di uno stencil riguarda solitamente una matrice le cui celle sono aggiornate in base a un calcolo fisso che coinvolge anche le celle vicine.

La fase 1 o incremento, aggiorna tutti i valori delle celle aumentandoli di una costante EDELTA.

La propagazione o fase 2 controlla che l'energia  $E$  di ogni cella non superi il valore critico EMAX, altrimenti ad  $E$  viene sottratta EMAX e ad ogni cella vicina (distanza 1 in metrica City-Block<sup>3</sup>) si somma EMAX/4.

Per avere dei risultati interessanti riguardo all'energia media, è utile che il dominio non sia ciclico, in modo che le celle di bordo abbiano meno di 4 vicini. Infatti, assumendo un dominio non ciclico durante ogni terremoto delle celle di margine, e sapendo che l'energia trasferita alle celle adiacenti (massimo 3) è comunque EMAX/4, l'energia totale diminuisce.

Il modello BK è un automa cellulare sincrono, ovvero un automa in cui tutte le celle vengono aggiornate contemporaneamente. Questa è la motivazione per cui esisteranno due matrici, `cur` e `next`, contenenti rispettivamente i valori delle energie al passo corrente e al passo successivo.

### 3.1.2 Implementazione QPULib

Prima di iniziare a programmare è stato inevitabile decidere come dividere la matrice in modo che ogni QPU abbia un carico di lavoro bilanciato, cosicché ho deciso di assegnare ad ogni QPU un numero di righe uguali ( $\pm 1$ ); inoltre ho reputato vantaggioso l'uso di ghost cell intorno alle matrici in modo da non dover controllare ogni volta gli accessi alle celle vicine.

#### Inizializzazione del programma

Ad ogni QPU è stato assegnato un numero di righe pari a  $n/QPUs$  con un +1 alle prime QPUs se è necessario per computare tutta la matrice. Ho inserito all'interno di uno `SharedArray` l'indice da cui ogni QPU deve iniziare a computare (non considerando le ghost cell), più precisamente la QPU di indice  $x$  inizia dall'indice `init_i[(x)*16]+1(GC)`. Dopodiché ho creato le due matrici principali `cur` e `next` e due `SharedArray`. Le matrici sono state inizializzate considerando come dimensione  $n+ = 2(GC)$  in modo che esista attorno alle matrici un margine di ghost cell. Gli `SharedArray` si dividono due compiti: uno serve per memorizzare quante celle hanno generato la scossa durante l'ultima iterazione (`count`) e l'altro per la somma totale delle energie (`sum`).

Per una migliore gestione delle matrici ho anche usato 3 puntatori a `SharedArray` (`pcur`, `pnext`, `tmp`), in modo da poter scambiare i riferimenti a `cur` e `next`.

---

<sup>3</sup>la metrica City-Block considera come vicini le celle a Nord, Est, Sud e Ovest

### parte centrale del programma

Una volta entrati nel ciclo principale, è opportuno implementare la funzione di aggiornamento, affinché sia in esecuzione sulle QPUs per ottimizzare il cuore del programma, ovvero la parte in cui viene speso la maggior parte del tempo di esecuzione.

Dopo una prima elaborazione dei dati richiesti in input per adattarli all'ambiente QPU, è stato utile costruire una maschera (`imask`) per estendere il dominio di `n` in input, dai soli multipli di 16 a qualsiasi valore maggiore di 15. Inserendo la maschera come condizione di un `Where` (controllando se è  $=0$  o  $> 0$ ), si possono manipolare solo alcuni elementi invece che tutti e 16 insieme. È comunque necessario tenere presente che durante la `store` verrà salvato un `Float` (Sezione 2.3.2) all'interno della matrice, ma sarà poi, in parte, sovrascritto.

Dopo aver considerato la parte della riga non multipla di 16, ogni QPU potrà continuare a calcolare `Earthquake` senza la necessità di fare controlli sullo stato della cella (`ghost cell` o no).

Ogni elemento quando deve essere elaborato, come prima cosa viene aumentato, dopodiché si controlla se è maggiore di `EMAX` per aggiornare `count`, viene propagata l'energia dei vicini che in quell'iterazione superano `EMAX`, infine viene tolta l'energia dovuta al sisma e aggiornato `sum`. Tutta la gestione dei dati, dove possibile è stata effettuata in modo non bloccante, in modo da rendere il programma più fluido, a discapito della leggibilità.

Quando tutti gli elementi sono stati considerati, ogni QPU deve calcolare il proprio `count` e `sum` e salvarlo nello `SharedArray` appositamente creato; per fare questo è stata usata la funzione `rotate`, grazie alla quale si ottimizza il numero di somme da effettuare.

È inevitabile una parte seriale per sommare i `count` e `sum` rilevati da ogni QPU, per scrivere e per invertire i puntatori a `cur` e `next`.

## 3.2 Moltiplicazione tra matrici

La moltiplicazione tra matrici, anche detta prodotto tra matrici o prodotto riga per colonna viene definita così:

Data una matrice reale  $A_{m \times n}$  e una matrice reale  $B_{n \times p}$ , il prodotto matriciale è una matrice reale  $C_{m \times p}$ , i cui elementi  $c_{ik}$  sono la somma dei prodotti degli elementi corrispondenti della riga  $i$  di  $A$  e della colonna  $k$  di  $B$ .

È utile studiare le prestazioni di questo kernel in quanto la moltiplicazione tra matrici è un'operazione molto comune all'interno di svariati problemi (la costruzione di oggetti 3D, il render e la modifica di immagini, ..), quindi è opportuno che sia il più veloce possibile.

### 3.2.1 Modello Matmul

Matmul viene eseguito con l'impiego di 4 matrici di dimensioni  $n \times n$ ; ogni cella delle prime due matrici  $C_{i,j}$ , dove  $i$  e  $j$  sono rispettivamente l'indice della riga e l'indice della colonna, sarà inizializzata tramite la formula  $C_{i,j} = (i \% 10 + j) / 10$ .

La moltiplicazione tra matrici non è un'operazione **cache efficient**, infatti l'accesso a una matrice per colonna (come è necessario fare per la seconda) è sconsigliato; per questo motivo, in situazioni del genere è utile trasporre<sup>4</sup> la seconda matrice.

### 3.2.2 Implementazione QPULib

Le QPUs nel recuperare i valori in memoria, non hanno la possibilità di accedere ai dati in modo sparso, in particolare, non potrebbero accedere ai valori della seconda matrice per colonna, quindi la trasposizione è d'obbligo.

#### Inizializzazione del problema

Anche in questo caso è stata eseguita una divisione del dominio per righe tramite lo `SharedArray init_i`. A differenza di prima, data la mancanza delle ghost cell in questo problema l'indice di partenza è `init_i[(x)*16]`. L'inizializzazione delle matrici risulta più intuitiva per questo esercizio, non essendoci le ghost cell.

---

<sup>4</sup>dati  $a_{ij}$  e  $a'_{ij}$  generici elementi delle matrici  $A$  e  $A^T$  quadrate di dimensione  $n$ ;  
 $A^T$  è la trasposta di  $A$  se  $a'_{ij} = a_{ij} \forall i, j < n$



## Problema trasposizione

La trasposizione in QPULib non è un problema facile da affrontare, come ho già detto (Sezione 3.2.2) lavorare con dati sparsi nella memoria usando QPULib è dispendioso, in termini di tempo di esecuzione; per completezza ho provato tre diversi approcci per risolvere questo problema:

- il primo tentativo è stato quello di un approccio *totalmente QPULib*, per meglio dire, tutta la trasposizione della matrice è effettuata tramite le QPUs. Questo portava a una 'matrice' trasposta in cui alcune righe (l'ultima riga computata da ogni QPU) erano più lunghe delle altre, questo come conseguenza diretta dell'uso esclusivo dei `Float`. Così per rendere l'output della funzione una vera matrice, è obbligatoria una parte seriale che riorganizza le celle extra.
- Ho provato anche con una soluzione ibrida: ovvero l'utilizzo delle QPUs escludendo l'ultima riga dal calcolo, la quale viene successivamente computata in modo seriale. Logicamente sembra il metodo più veloce, in quanto parallelizza il più possibile senza 'sprecare' calcoli; in pratica però non lo è.
- Per ultimo ho provato, quasi per curiosità ad effettuare la trasposizione in modo seriale e ho notato che è l'opzione più veloce per ogni input gestibile da Raspberry Pi 2.

## parte centrale del programma

Al contrario della soluzione di Earthquake (sezione 3.1.2), si è deciso di limitare l'input `n` a un valore maggiore di 15 e multiplo di 16; così facendo sono necessari meno controlli e il programma risulta più veloce.

Sviluppato già il problema della trasposizione, la funzione che effettua la moltiplicazione prende in ingresso la seconda matrice già trasposta. L'implementazione QPULib di `Matmul` per la sua risoluzione invoca 3 cicli `For` innestati; i primi due agiscono sull'indice riga `i` e sull'indice `j` colonna della matrice risultato. Il ciclo più interno modifica l'indice `k` per spostarsi lungo la riga di entrambe le matrici (una volta trasposta la seconda matrice, la moltiplicazione avviene riga per riga). Considerando che il dominio delle matrici è obbligatoriamente multiplo di 16, non è utile inserire `Where` in quanto rallenterebbero solo il flusso del programma.

Completato il ciclo più interno si ha come risultato un `Float` i cui elementi vanno sommati per trovare il valore risultato della cella `i,j`; anche in questo caso si utilizza la `rotate` per risparmiare qualche ciclo di clock.

Ricordando che QPULib non può salvare in memoria meno dati di un `Float`; alla fine del calcolo avremo (come nel caso della trasposizione totalmente QPULib) una 'matrice' che presenta nell'ultima riga calcolata da ogni QPU 15 elementi extra. Anche in questo caso è necessaria una parte seriale che ricontrolli l'output e lo renda una matrice.

# Capitolo 4

## Valutazioni delle prestazioni

In questo capitolo valuteremo le prestazioni delle implementazioni QPU-Lib appena descritte. Per farlo ogni programma verrà confrontato anche con una versione parallela su CPU: per farlo sono state implementate una versione OpenMP di Earthquake, e una SIMD (basata sui `Vector data type`) di Matmul.

### 4.1 OpenMP Earthquake

Per l'implementazione di Earthquake è stato utile condensare le quattro funzioni del codice seriale in una sola, in modo che la sezione da parallelizzare sia unica e ben definita, come si può vedere dal listato 4.1.

```
for (int i = 1; i<n-1; i++) {
    for (int j = 1; j<n-1; j++) {
        float F = *IDX(grid, i, j, n);
        /* update cells */
        if (F > EMAX) { count++; }
        *IDX(next, i, j, n) = F;
        sum += F;
    }
}
```

Listato 4.1: Ciclo di aggiornamento da parallelizzare

Modificata a dovere la versione seriale del programma, una prima versione parallela OpenMP è stata pressoché immediata: infatti è bastato aggiungere una direttiva `#pragma omp parallel for` prima del ciclo più esterno.

La prima versione parallela è stata poi studiata e sottoposta a diverse considerazioni: in particolare sono sorte delle domande riguardo la clausola `reduction`<sup>1</sup> e sulla clausola `collapse`<sup>2</sup>.

La clausola `reduction` può essere utilizzata per effettuare le due somme che si vedono nel ciclo: quella di `count` e di `sum`. In particolare, è sembrata utile in quanto è risultata più efficiente di altre due implementazioni: la prima era realizzata tramite un array nel quale ogni thread OpenMP inseriva la sua somma parziale e alla fine del ciclo tutti gli elementi venivano sommati in modo seriale (dal `master`); un'altra soluzione era proteggere le variabili risultato con una direttiva `atomic`, in modo che gli accessi in memoria non si potessero sovrapporre. Nonostante la clausola `reduction` aggiunga un po' di overhead è comunque la soluzione migliore per il calcolo di `count` e di `sum`.

La clausola `collapse` può aumentare le prestazioni in quanto rende unico lo spazio di iterazione, rendendo più bilanciato il carico di lavoro per thread, però considerando il carico di ogni iterazione circa costante notiamo che: dato un numero di iterazioni sufficientemente alto rispetto al numero dei thread OpenMP, lo sbilanciamento di carico è minimo, quindi l'overhead generato dalla clausola `collapse` è maggiore del guadagno che porterebbe usarla.

## 4.2 Matmul con Vector data type

Programmando con i `Vector data type` è fondamentale definire il tipo di vettore con cui si andrà a lavorare; in questo caso è stato definito il tipo `v4f` (vettore di 4 `float`) per effettuare l'accesso per riga alle matrici. È necessario usare un tipo `v4f` anche per sommare i valore dei vari prodotti e per poter calcolare il valore finale da assegnare a ogni cella risultato.

Considerando che l'hardware potrebbe non supportare le istruzioni SIMD (come detto nella sezione 1.3.2 il codice compilato potrebbe essere seriale) è utile mantenere una variabile o una macro per sapere di quanti elementi è composto il vettore: in questo caso ho definito la macro `VLEN` usata come addendo nel ciclo più interno.

---

<sup>1</sup>la clausola `reduction` prende in input un'operazione e una lista di variabili; dopodiché esegue l'operazione su ogni elemento della lista

<sup>2</sup>la clausola `collapse` specifica quanti cicli, in una serie di cicli innestati, devono essere collassati in un unico spazio di iterazione

## 4.3 Prestazioni QPULib e confronto

In questa sezione valuteremo le prestazioni delle implementazioni QPULib usando i criteri di scalabilità introdotti nella sezione 1.2, infine confronteremo i tempi d'esecuzione con le versioni operanti su CPU appena descritte.

Per tutti i calcoli delle prestazioni è stato usato un Raspberry Pi 2. Sono state effettuate 11 rilevazioni dei tempi dalle quali è stato tolto il valore che più si discostava dal valor medio, dopodiché è stata ricalcolata la media e considerata come valore effettivo.

Calcolare i tempi di esecuzione dato un carico di lavoro totale costante non ha portato alcun problema: una volta deciso l'input iniziale è bastato modificare il numero di QPUs attive.

Riferendosi invece al carico di lavoro per thread costante, essendo una matrice quadrata l'oggetto di entrambi i problemi, il carico di lavoro scala con la radice quadrata del lato. In quanto ovviamente non si possono avere numeri di celle non interi, per calcolare l'input del lato della matrice è stato necessario moltiplicare l'input di partenza per la radice quadrata del fattore di parallelizzazione e arrotondare all'intero più vicino. Questo porta una leggera imprecisione nel calcolo di alcune scalabilità deboli (in particolare nel caso di Earthquake QPULib in cui l'input deve essere anche multiplo di 16).

### 4.3.1 Scalabilità dato un input costante

Per valutare le prestazioni dei due kernel bisogna prima di tutto decidere su quale input si effettueranno le rilevazioni: per Earthquake sono sembrate opportune 30.000 iterazioni su una matrice  $150 \times 150$ ; invece per Matmul le matrici sono di dimensione  $n = 512$ .

Come si può vedere nel grafico 4.1, dato un input costante Earthquake non scala in modo adeguato dopo le 3 QPUs attive; possiamo individuare il problema controllando il grafico 4.2 dei tempi relativo alle stesse prove: la durata dell'esecuzione tende ad un valore asintotico di circa 33 secondi. Probabilmente quel tempo è dato dal calcolo seriale di `count` e `sum` che limita lo Speedup massimo e di conseguenza il tempo d'esecuzione per la legge di Amdahl (formula 1.4). Al contrario Matmul, nonostante contenga una parte seriale, mantiene la scalabilità forte oltre il 50%.

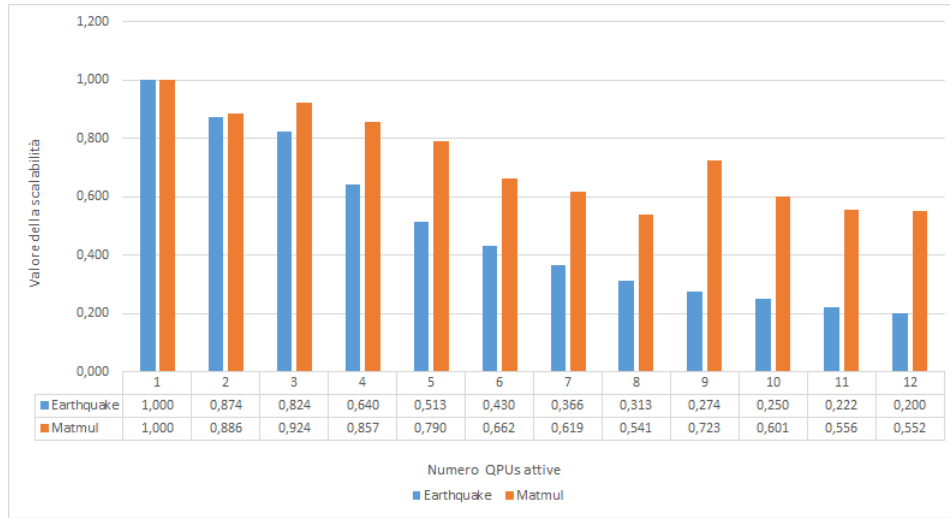


Figura 4.1: Grafico della scalabilità forte delle implementazioni QPULib

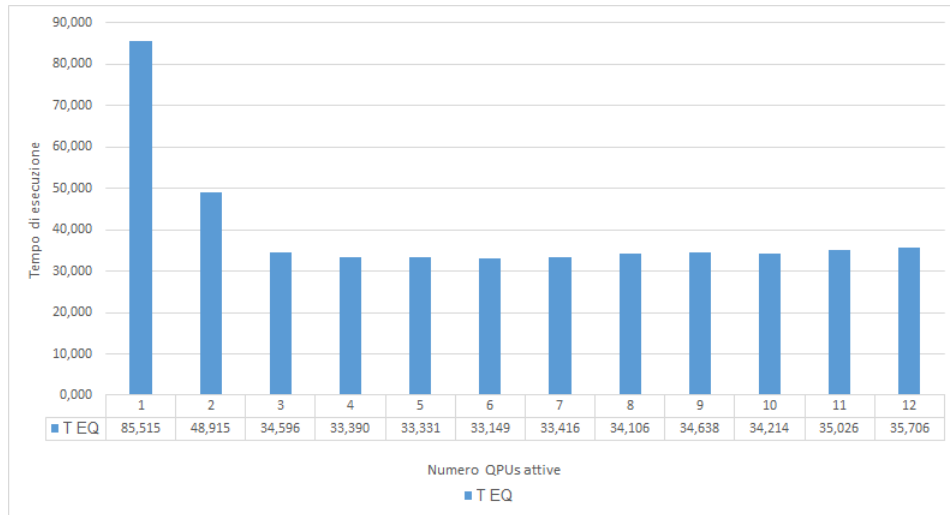


Figura 4.2: Grafico dei tempi dato un input costante di QPULib Earthquake

### 4.3.2 Scalabilità debole

In entrambi i kernel è stato necessario diminuire l'input iniziale su cui si sono basati i precedenti grafici. Per Matmul è stata ridotta la dimensione iniziale della matrice a  $n_1 = 256$  in quanto altrimenti al crescere del numero di QPUs la memoria di Raspberry Pi non avrebbe contenuto tutte le matrici. L'esecuzione di Earthquake è stata effettuata con 20.000 iterazioni fisse su una matrice a dimensione variabile (in base alle QPUs), a partire da  $n = 128$ ;

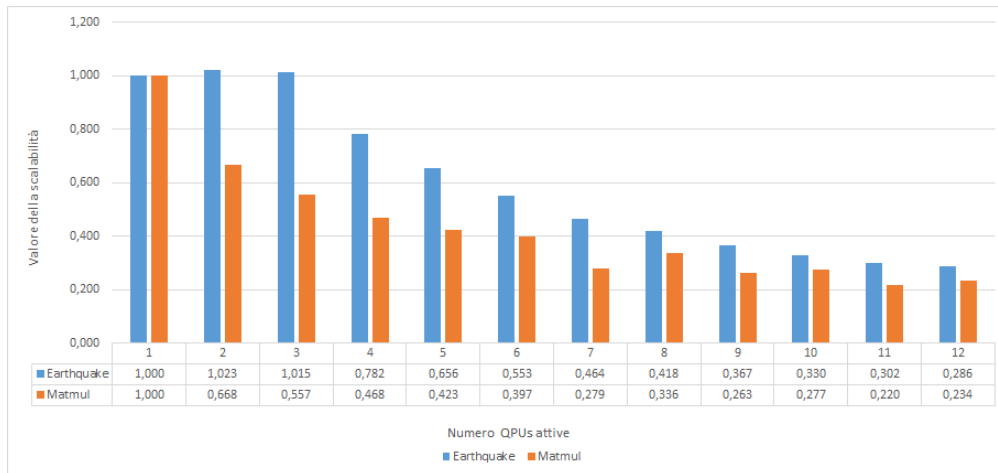


Figura 4.3: Grafico della scalabilità debole delle implementazioni QPULib

Come si può vedere dal grafico 4.3 in entrambe le implementazioni abbiamo un notevole abbassamento della scalabilità debole. Questo può essere dovuto dalle caratteristiche generali di Raspberry Pi nella gestione dei dati condivisi, oppure alla velocità del bus nella gestione delle QPUs.

### 4.3.3 Confronto con altre implementazioni

Come si vede dal grafico 4.4 per valutare più implementazioni di un programma è necessario anche tenere conto anche dei tempi di esecuzione seriali: considerando  $T_{serial} = T(1)$  tutti i concetti descritti nella sezione 1.2 riguardano l'esecuzione di una stessa implementazione.

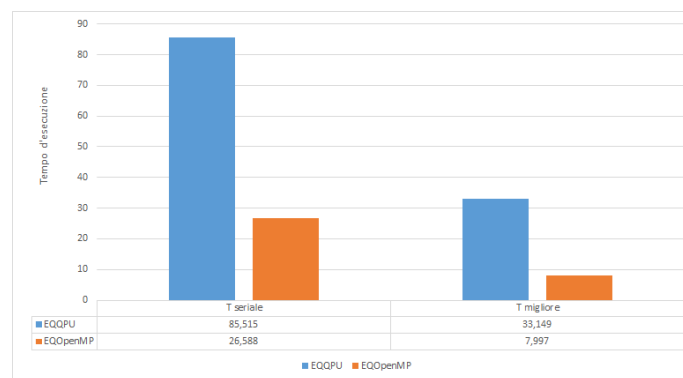


Figura 4.4: tempi seriali e migliori per le implementazioni di Earthquake

La prima colonna del grafico 4.4 sottolinea che l'implementazione QPULib necessita di molti più calcoli (o in generale di più gestione dei dati) della

versione seriale e nonostante la parallelizzazione questa differenza non riesce ad essere colmata.

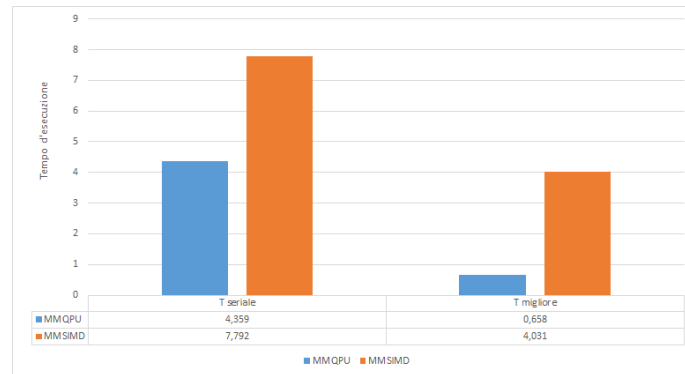


Figura 4.5: tempi seriali e migliori per le implementazioni di Matmul

Al contrario l'implementazione QPULib di Matmul (grafico 4.5) risulta essere più efficiente già con una QPU attiva, a maggior ragione utilizzandone di più dato che scala meglio della versione SIMD.



# Conclusioni

In questo lavoro abbiamo mostrato come è possibile programmare le GPU di Raspberry Pi per scopi diversi dal solo calcolo grafico. Per ogni chiarimento sul codice, questo è reperibile nel repository pubblico all'indirizzo <https://bitbucket.org/SimoneMagnani/videocore-iv-e-il-calcolo-generico/>.

Nonostante QPULib sia un linguaggio ancora in fase di sviluppo, in alcune implementazioni il miglioramento delle prestazioni è evidente. Come si può vedere dai tempi di esecuzioni, soprattutto nella gestione di grandi quantità di dati, è importante saper programmare la GPU per scopi non strettamente legati alla grafica.

È anche importante ricordare che, nonostante in QPULib attraverso gli `SharedArray` la gestione della memoria condivisa sia del tutto trasparente, questo porta un overhead non trascurabile soprattutto finché la `store` non possiede una coda di esecuzione (sezione 2.3.2). Per questo motivo è corretto analizzare il problema e la quantità di computazioni da svolgere prima di decidere qual è l'unità di calcolo più adatta: per ottenere un effettivo miglioramento delle prestazioni sfruttando il calcolo parallelo sulla GPU del Raspberry Pi, occorre che la quantità di dati da elaborare sia significativa e che le operazioni di calcolo siano principalmente SIMD.

Al fine di ottenere un programma ad alte prestazioni la scelta migliore risulta essere un compromesso tra le tipologie di calcolo, mantenendo su GPU le operazioni SIMD per oltre una soglia minima di valori e su CPU i calcoli sequenziali o su piccole quantità di dati.



# Ringraziamenti

Ringrazio prima di tutto il prof. Moreno Marzolla per la prontezza e la disponibilità che ha avuto nel seguirmi.

Ringrazio inoltre lo Spaceteam, l' #Einstein, gli amici e tutti i compagni di corso con i quali ho condiviso molte ore di studio e di svago.

Ringrazio in particolare Mattia Magnani e Jessica Biondi per avermi supportato e sopportato durante la stesura della tesi ed infine, senza il cui supporto non sarei riuscito a raggiungere questo risultato, ringrazio la mia famiglia per tutto ciò che mi hanno permesso di fare.



# Bibliografia

- [1] Blaise Barney. Lawrence livermore national laboratory. <https://computing.llnl.gov/tutorials/openMP/>.
- [2] Brian Benchoff. Introducing the raspberry pi 3. <https://hackaday.com/2016/02/28/introducing-the-raspberry-pi-3/>.
- [3] Ritesh Bendre. Raspberry pi 3: Everything you wanted to know about the credit card sized computer. <https://www.bgr.in/news/raspberry-pi-3-everything-you-wanted-to-know-about-the-credit-card-sized-computer/>.
- [4] NVIDIA Corporation. Cuda zone, 2006. <https://developer.nvidia.com/cuda-zone>.
- [5] Raspberry Pi Foundation. Raspberry pi, 2015. <https://www.raspberrypi.org>.
- [6] GCC Official Guide. Using vector instructions through built-in functions. <https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html>.
- [7] Andrew Holme. Fast fourier transform library for the raspberry pi, 2014. [http://www.aholme.co.uk/GPU\\_FFT/Main.htm](http://www.aholme.co.uk/GPU_FFT/Main.htm).
- [8] IEEE. Ieee 754-2019 - ieee standard for floating-point arithmetic. <https://standards.ieee.org/content/ieee-standards/en/standard/754-2019.html>.
- [9] R. Burridge; L. Knopoff. Model and theoretical seismicity. *Seismological Society of America*, 1967. <https://pubs.geoscienceworld.org/ssa/bssa/article-abstract/57/3/341/116471/model-and-theoretical-seismicity>.
- [10] mn416. A language and compiler for the raspberry pi gpu, 2016. <https://github.com/mn416/QPULib>.
- [11] nineties (Koichi NAKAMURA). Python library for gpgpu on raspberry pi, 2015. <https://github.com/nineties/py-videocore>.
- [12] Eben Upton. A birthday present from broadcom. <https://www.raspberrypi.org/blog/a-birthday-present-from-broadcom> (Official Blog Raspberry Pi).
- [13] Philip Kearey; Frederick J. Vine. *Tettonica globale*. Zanichelli, 1994.