Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Valutazione delle Prestazioni di Processori a Basso Consumo Energetico in Applicazioni Parallele

Relatore:
Chiar.mo Prof.
MORENO MARZOLLA

Presentata da:
FEDERICO BAROCCI

Sessione III
Anno Accademico 2014/2015

Think deeply about things.
Don't just go along because that's the way things are or that's what your friends say. Consider the effects, consider the alternatives, but most importantly, just think.
(Aaron Swartz)

Sommario

In questo lavoro di tesi si è voluto esaminare il calcolo parallelo nelle sue caratteristiche generali, quindi trovarne applicazione in contesti pratici. Nello specifico il lavoro di indagine si è concentrato nell'esaminare le potenzialità del calcolo parallelo su Rasperry Pi, un piccolo calcolatore di tipo single-board a basso consumo energetico, diffuso principalmente con lo scopo di favorire l'insegnamento dell'informatica nelle scuole e che ha ricevuto interesse da parte di molti sviluppatori ed appasionati, come si può infatti notare dalle community attive presenti in rete. Nelle varie versioni rilasciate di questo dispositivo si può notare che il processore non possiede prestazioni molto elevate e, fatta ecezione per un singolo modello fino ad ora rilasciato, il processore è di tipo single-core. Da questo aspetto non particolarmente positivo la curiosità si volge quindi alla scheda video integrata che, nonostante le ridotte dimensioni, presenta una potenza di calcolo paragonabile ad quella di una XBox di prima generazione. Tuttavia utilizzare la scheda video del Rasperry Pi al fine di eseguire calcoli paralleli, quello che viene comunemente definito GPGPU (*general-purpose computing on graphics processing units*), risulta un compito particolarmente difficile. Gli strumenti comunemente impiegati per l'implementazione degli algoritmi paralleli su GPU non risultano disponibili, come CUDA, o comunque non sono in grado di accedere alle potenzialità della scheda video, come OpenCL. Nonostante le limitazioni esistenti si è cercato di sperimentare metodi alternativi per l'implementazione degli algoritmi paralleli su GPU facendo uso di OpenGL ES 2, la libreria di *computer graphics* disponibile per i sistemi embedded e supportata dal

Raspberry Pi. Da questa indiagine è stato possibile individuare una tecnica capace di utilizzare le API grafiche per eseguire calcoli di varia tipologia ed implementare degli algoritmi paralleli, analizzando quali siano le limitazioni e quali invece le potenzialità di impiego.

Questo documento è così suddiviso: nel primo capitolo verranno presentati alcuni dei concetti principali relativi al calcolo parallelo, la tassonomia di Flynn, le metriche di valutazione, si parlerà di GPGPU e di alcuni degli strumenti esistenti per implementare algoritmi paralleli; nel secondo capitolo si descriverà il Rasperry Pi, verranno presentati alcuni dei casi di utilizzo e si presenterà il problema di implementare GPGPU su Raspberry Pi; il terzo capitolo presenterà la tecnica per implementare algoritmi paralleli utilizzando le librerie OpenGL, descrivendo la modalità di impiego, quali sono le potenzialità e quali sono le limitazioni; il quarto capitolo descriverà l'algoritmo di Seam Carving per il restringimento di immagini nelle sue implementazioni sequenziali e parallele, si valuteranno le difficoltà implementative e le soluzioni adottate; il quinto capitolo esaminerà le prestazioni ottenute con l'impiego del Seam Carving su Rasperry Pi nelle implementazioni sequenziale e parallela, traendo infine alcune conclusioni su vantaggi e potenzialità date dall'impiego del calcolo parallelo sulla GPU del Raspberry Pi.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Concetti di Calcolo Parallelo

In questo capitolo affrontiamo il tema del calcolo parallelo, descrivendo le caratteristiche, i vantaggi e le eventuali difficoltà che si riscontrano nella loro implentazione. Le differenti tipologie di calcolo parallelo verrano presentate sia in base al tipo di approccio risolutivo, con riferimento alla tassonomia di Flynn, ed in base alle risorse fisiche impiegate per l'effettiva esecuzione. Vengono inoltre presentate alcune delle metriche comunemente impiegate per valutare l'efficienza e le prestazioni degli algoritmi paralleli.

1.1 Motivazioni

Il calcolo parallelo è una disciplina che ha trovato applicazione in molti ambiti scientifici come fisica, biologia, matematica, meteorologia, analisi delle immagini e in tutte quelle situazioni in cui grandi quantità di dati devono essere elaborati per effettuare delle simulazioni o per estrarre informazioni il più rapidamente possibile, oppure la cui elaborazione sequenziale richiederebbe troppo tempo. Da un punto di vista storico, tradizionalmente i programmi vengono scritti per essere eseguiti da una unità di calcolo sequenziale, ad esempio un computer con una singola CPU. Tuttavia una singola CPU potrebbe non fornire sufficienti prestazioni per eseguire velocemente grandi quantità di calcoli, l'idea di impiegare più unità di calcolo risulta quindi

abbastanza chiara, ovvero si cerca di ridurre il tempo di elaborazione suddividendo il lavoro tra più unità di calcolo. Se prendiamo come esempio situzioni della vita quotidiana sappiamo che generalmente si può trarre vantaggio dal parallelismo, nella vita domestica come in quella lavorativa. Se un solo individuo deve svolgere un incarico particolarmente complesso, questo potrebbe richiedere molto tempo per completarlo. Se invece più individui hanno un incarico comune, una volta individuato come suddividere i compiti ed aver concordato un opportuno assegnamento del lavoro, generalmente completare il compito dovrebbe richiedere complessivamente meno tempo di quello che otterrebbe un singolo individuo. Questa analogia rappresenta quello che essenzialmente riguarda il calcolo parallelo, ovvero si pone come obiettivo di suddividere un problema in più sottoproblemi e far si che più unità di calcolo collaborino simultaneamente per la sua risoluzione.

Nella progettazione degli algoritmi uno tra i parametri che viene più frequentemente considerato per valutarne la qualità è l'efficienza, ovvero la capacità di sfruttare meno risorse possibili sia in termini di tempo necessario al calcolo, sia in termini di spazio necessario in memoria. Sebbene l'evoluzione della tecnologia abbia consentito di avere a disposizione hardware capace di effettuare calcoli sempre più velocemente e memoria a disposizione in quantità sempre maggiore [19], il parametro dell'efficienza resta un indice di qualità importante. Solitamente ci si riferisce all'efficienza di un algoritmo con termini quali *costo asintotico* o *complessità computazionale* in modo da indicare le risorse di calcolo necessarie in modo astratto rispetto le reali caratteristiche fisiche dell'hardware impiegato.

Tradizionalmente gli algoritmi vengono pensati in maniera astratta rispetto le caratteristiche fisiche del calcolatore che li eseguirà, tuttavia il tempo effettivamente necessario alla risoluzione di un dato problema non può essere trascurato totalmente. Pensiamo come esempio alle previsioni climatiche, se il calcolo richiedesse troppo tempo di elaborazione si potrebbe rischiare di ottenere la previsione meteorologica di un momento passato, un risultato poco utile nella pratica. Questa ipotetica situazione spinge a considerare i

vincoli temporali oltre la complessità computazionale, ovvero considerare i casi in cui la sola efficienza può non essere sufficiente.

In queste situazioni in cui il tempo di calcolo risulta essere cruciale, una delle possibili soluzioni è quella di utilizzare macchine più veloci. Questa soluzione è semplice per quanto riguarda il software, non occorre implementare algoritmi più efficienti, sarà sufficiente eseguire il programma su una macchina nuova e più potente. Questo approccio è stato possibile per molti anni grazie al progresso tecnologico che ha consentito di sviluppare processori sempre più piccoli e veloci, un fatto evidenziato anche dalla **legge di Moore** [30] la quale afferma per via empirica che il numero dei transistor all'interno di una CPU raddoppia ogni 18 mesi. Vi sono tuttavia delle perplessità oggettive riguardo il numero di componenti presenti continui ad aumentare all'infinito, motivate da osservazioni come, la più evidente, la CPU possiede una superficie finita che è un limite fisico per il numero di transistor, per poi considerare problemi come la dissipazione di calore ed altri ancora. Tuttavia è stato possibile osservare che le prestazioni dei microprocessori sono raddoppiate a distanza di breve tempo, circa ogni 18 mesi, evidenziando quindi una crescita esponenziale nella potenza di calcolo fornita dall'hardware.

Date le difficoltà legate ai limiti fisici per la costruzione di CPU più veloci, la soluzione adottata per aumentarne le prestazioni complessive fu di aumentare il numero di core (unità di calcolo) all'interno di uno stesso processore, introducendo le architetture parallele. Se infatti osserviamo un sistema operativo in un momento successivo al suo avvio possiamo notare che nello stesso momento sono in esecuzione molti programmi e servizi di sistema, gestiti in una coda di processi e ciascuno dei quali utilizza per brevi istanti di tempo la CPU. Molti processi anche se richiedono minime risorse di calcolo, tendenzialmente sono necessari per il corretto funzionamento del sistema. Grazie ai processori multicore più processi possono essere eseguiti contemporaneamente su più unità di calcolo, indipendentemente gli uni dagli altri, aumentando le prestazioni complessive del sistema.

Con il parallelismo implementato a livello architetturale, i programmi e

gli algoritmi sequenziali continuano tuttavia ad essere tali come per i processori single core. Una architettura parallela non produce "automaticamente" software che esegue calcoli in parallelo, viceversa gli algoritmi paralleli necessitano ovviamente di una architettura parallela per essere eseguiti. La possibilità di avere a disposizione più unità di calcolo porta quindi alla formulazione di alcune domande che spingono ad esaminare con attenzione le potenzialità derivanti:

- Come suddividere il lavoro necessario ad un singolo algoritmo tra più unità di calcolo per ridurre il tempo di elaborazione?
- Come misurare quanto e se aumentano le prestazioni?
- Come valutare la qualità di un algoritmo parallelo?
- Come implementare concretamente algoritmi paralleli?

1.2 Modelli di calcolo

La presenza di più unità di calcolo all'interno di un unico elaboratore non è l'unica motivazione che spinge allo studio e allo sviluppo di algoritmi non seriali. Questi studi sono stati infatti approfonditi per decenni anche prima della disponibilità di processori multi-core, valutando l'applicabilità e le caratteristiche per l'impiego degli algoritmi in vari modelli di calcolo. Occorre distinguere le caratteristiche dei principali modelli utilizzati [18]:

Calcolo concorrente contraddistinto da più processi presenti sulla stessa macchina che condividono un insieme di risorse, ognuno dei quali svolge un compito specifico. Non è vincolante che questi collaborino per la risoluzione di uno stesso problema, tuttavia più processi potrebbero dedicarsi ad un compito distinto che contribuisce alla risuluzione di un problema comune, ognuno procedendo in maniera indipendente dagli altri. Ad esempio, nel *file sharing* un processo potrebbe dedicarsi all'upload di dati mentre un altro processo si occupa del download.

Calcolo distribuito composto da più unità di calcolo distinte e connesse in rete, contribuiscono alla risoluzione di un problema mediante lo scambio di messaggi. Questo modello punta ad unire più risorse di calcolo per risolvere un problema comune quando l'elaborazione potrebbe richiedere troppo tempo su di una singola macchina, oppure quando si vuole garantire che un sistema sia sempre disponibile e deleghi i calcoli da svolgere in altre macchine, oppure quando i dati da elaborare sono collocati su più nodi di una rete.

Calcolo parallelo caratterizzato da più unità di calcolo presenti in un'unica macchina che collaborano alla risoluzione di un unico problema comune operando contemporaneamente su un insieme di dati. Le modalità in cui i dati vengono elaborati e le istruzioni eseguite si distinguono in base ad alcune caratteristiche, come evidenziato nella **tassonomia di Flynn** [31]:

- **Single Instruction Single Data** (**SISD**), corrisponde al modello classico di elaborazione seriale dei dati, viene eseguita una sola istruzione alla volta per un singolo dato;
- **Single Instruction Multiple Data** (**SIMD**), viene eseguita una sola istruzione contemporaneamente su più dati, si tratta del modello di calcolo tipico delle GPU e dei processori vettoriali;
- **Multiple Instruction Single Data** (**MISD**), più istruzioni differenti vengono eseguite su un solo dato alla volta, un modello di calcolo raramente impiegato in teoria e non sono note realizzazioni hardware, viene solitamente sostituito dal modello SIMD;
- **Multiple Instruction Multiple Data** (**MIMD**), più istruzioni differenti vengono eseguite su insiemi di dati differenti, consiste tipicamente da più unità di calcolo indipendenti che eseguono operazioni in maniera asincrona.

Un altro modo per classificare i modelli di calcolo si basa sull'accesso alla memoria, che può essere **condivisa** quando le unità di elaborazione operano

sulla stessa area di memoria o **distribuita** quando le unità di calcolo possiedono ciascuna una memoria personale e lo scambio di informazioni avviene mediante comunicazioni in rete. Nelle architetture a memoria condivisa si possono effettuare ulteriori distinzioni in **UMA** (*Uniform Memory Access*) quando l'acesso alla memoria avviene direttamente e tutte le unità di elaborazione hanno quindi gli stessi tempi di accesso, oppure **NUMA** (*Non-Uniform Memory Access*) quando ogni unità di calcolo possiede una propria memoria locale e accede alla memoria condivisa mediante una rete di interconnessione.

1.3 Stream Processing

Lo "*stream processing*" [26] è un paradigma di programmazione parallela di tipo SIMD con lo scopo di semplificare lo sviluppo di hardware e software restringendo le possibilità sulle tipologie di calcoli paralleli che possono essere svolti. In questo modello vengono considerati gli *stream* che sono collezioni di dati sui quali devono essere eseguiti gli stessi calcoli e *kernel* che sono funzioni da applicare ai dati da elaborare. Lo *streaming processor* si occupa di eseguire la funzione kernel su tutti gli elementi dello stream di input e producendo uno stream di output contenente i dati elaborati.

In questo modello la GPU può essere considerata uno streaming processor che esegue la funzione kernel, implementata attraverso gli shader, su tutti gli elementi dello stream in input, ovvero vertici e texture, producendo uno stream in output rappresentato da una seconda texture con i risultati dell'elaborazione degli shader sugli elementi dati in input. Questo tipo di elaborazione si mostra principalmente adatta ad implementare algoritmi che devono eseguire elevate quantità di calcoli logico-matematici su ogni dato piuttosto che accedere a locazioni variabili della memoria. Inoltre i calcoli si devono poter eseguire parallelamente, senza che sia presente una dipendenza tra i risultati di una computazione ed un'altra. In altri termini, il kernel deve potrer operare su dati locali ed indipendenti tra loro per poter essere efficacemente utilizzato in questo modello.

1.4 Confronto architettura CPU e GPU

Sebbene i microprocessori siano evoluti nel tempo raggiungendo prestazioni molto più elevate rispetto che in passato, la loro architettura fondamentale continua ad essere basata sul modello di Von Neumann. In questo modello la CPU è composta da due unità fondamentali: la CU (*Control Unit*) che si occupa di eseguire le istruzioni e dirigere le operazioni del processore, la ALU (*Arithmetic Logic Unit*) che si occupa di eseguire calcoli ed operazioni logiche sui dati. La CPU è connessa mediante un bus dati alla memoria RAM consentendo la lettura e la scrittura di dati. Inoltre, per velocizzare l'accesso ai dati durante l'eseucuzione di un programma, le CPU moderne possiedono delle memorie cache locali molto veloci per ridurre la latenza di accesso alla RAM. Secondo la tassonomia di Flynn, le CPU tradizionali sono classificate come SISD, un unico flusso di istruzioni che esegue calcoli su singoli dati alla volta.

Le GPU (*Graphics Processing Unit*) possiedono un'architettura differente dalle CPU, costituita da una griglia di ALU altamente specializzate per eseguire calcoli necessari alla grafica computerizzata, tipicamente caratterizzata da operazioni con numeri in virgola mobile su grandi quantità di dati. Ad ogni ALU vengono assegnate memorie cache per le operazioni sui dati al fine di ridurre il più possibile i tempi di accesso alla memoria. Tradizionalmente le GPU sono state introdotte in ausilio alla CPU con lo scopo di eseguire operazioni prefissate necessarie alla grafica 3D; non potevano essere programmate e pertanto venivano definite *fixed-function pipeline*. Con l'evoluzione delle tecniche grafiche si è reso necessario introdurre la possibilità di programmare (prima in parte, poi totalmente) le operazioni eseguite dalla scheda video. Il passaggio successivo fu quello di impiegare la scheda video per calcoli di varia natura e non solo legati alla grafica computerizzata, introducendo il concetto di **GPGPU** (*General Processing GPU*). La caratteristica della GPU è che consente di eseguire operazioni velocemente e parallelamente su grandi insiemi di dati, classificandosi come SIMD secondo la tassonomia di Flynn.

Generalmente le CPU sono idonee ad essere impiegate per scopi generici,

hanno una bassa latenza nell'accesso alla RAM e sono relativamente facili da programmare. In contrapposizione le GPU sono molto specializzate, adatte ad eseguire velocemente calcoli su grandi insiemi di dati cercando di massimizzare il *throughput*, ovvero la quantità di dati generati per unità di tempo, e la ramificazione del flusso di esecuzione del codice dovrebbe essere ridotto per quanto possibile al fine di garantire che tutte le unità di calcolo svolgano le stesse operazioni in sincronia.

1.5 Metriche di valutazione

Normalmente non è garantito che aumentare il numero di processori per risolvere un problema produca un aumento di prestazioni in maniera linearmente proporzioanle al numero di unità di calcolo impiegate. Alcuni problemi sono intrinsecamente seriali quindi aumentare il numero di unità di calcolo potrebbe non produrre alcun vantaggio, pertanto si rivela spesso necessario riformulare gli algoritmi affinché sfruttino efficacemente le potenzialità di calcolo a disposizione.

Per valutare il contributo derviante dall'impiego di un algoritmo parallelo occorrono delle metriche che consentono di stabilire l'aumento delle prestazioni in relazione all'analoga implementazione seriale. Una metrica principalmente utilizzata per valutarne la qualità è lo **speedup**:

$$S_p = \frac{T_1}{T_p}$$

dove T_1 è il tempo di esecuzione con un singolo processore mentre T_p è il tempo di esecuzione con p processori. Nel caso in cui valga $S_p = p$ allora lo speedup è definito lineare.

Un altro modo per valutare la qualità di un algoritmo parallelo è definito **efficienza** e viene calcolato come:

$$E = \frac{S_p}{p} = \frac{T_1}{p \cdot T_p}$$

ovvero viene messo in relazione il tempo impiegato da p processori per eseguire l'algoritmo rispetto un singolo processore. Se all'aumentare di p il valore di E risulta costante allora si dice che l'algoritmo è scalabile.

Un ulteriore modo per determinare il miglioramento atteso di un algoritmo parallelo viene espresso con la **legge di Amdahl** [32] secondo la quale il tempo di calcolo T_1 di un algoritmo eseguito su una singola unità di elaborazione può essere suddiviso nel tempo s necessario per il calcolo sequenziale non parallelizzabile e nel tempo p impiegato dai frammenti di codice che può essere parallelizzato. Questa osservazione unita alla definizione di speedup con n unità di calcolo da luogo alla legge di Amdahl:

$$S_p \leq \frac{s+p}{s+p/n} = \frac{1}{s+p/n}$$

dove $s + p = 1$ poiché corrisponde al tempo totale di esecuzione di un algoritmo sia nelle parti sequenziali che nelle parti parallele. Se si considerano infiniti processori, ovvero $n \to \infty$, si ha che il massimo speedup possibile sia $1/s$ ed è una importante implicazione di questa legge. Il limite massimo possibile per lo speedup non può essere che una frazione del tempo richiesto per il lavoro non parallelizzabile, anche utilizzando infiniti processori.

1.6 Tecnologie esistenti

Per implementare algoritmi paralleli esistono vari strumenti che ne agevolano lo sviluppo. Un metodo consiste nell'impiego di tecniche per sviluppare applicazioni multiprocesso (ad esempio con *fork()* nei sistemi GNU/Linux) e multithread. Queste tecniche sono disponibili nativamente nel sistema operativo e sono alla base della programmazione concorrente. L'impiego nativo di questi strumenti, sebbene utile in molti contesti, può comportare una maggiore difficoltà nella programmazione parallela, cioè garantire che ogni processo e ogni thread venga eseguito contemporaneamente se sono disponibili più unità di calcolo, compito normalmente delegato al sistema operativo. Inoltre il loro impiego in un contesto distribuito richiede di implementare

anche l'aspetto di comunicazione tra processi utilizzando *socket*. L'impiego degli strumenti nativi di sistema è certamente possibile tuttavia richiede un maggior lavoro di sviluppo rispetto l'impiego di strumenti e *framework* dedicati.

Per semplificare i problemi di implementazione e di debug sono stati realizzati degli strumenti che consentono di semplificare il lavoro di sviluppo, consentendo di focalizzare sui problemi legati al calcolo parallelo. Tra questi uno degli strumenti utilizzati è **MPI** (*Message Passing Interface*) [27] che consente di sviluppare algoritmi paralleli e distruibiti senza preoccuparsi dell'architettura sottostante, ovvero mediante apposite primitive è possibile eseguire facilmente codice parallelo su processore multicore come su più macchine distribuite in rete. I processi operano sui dati che possiedono nella propria memoria (non in memoria condivisa) e scambiano informazioni mediante funzioni *send()* e *receive()*.

Un altro strumento disponibile per lo sviluppo di applicazioni parallele è **OpenMP** [24] che consente di sviluppare applicazioni parallele su CPU multi-core e consente di sfruttare la memoria condivisa per l'accesso ai dati. Questo strumento consente di utilizzare direttive per il compilatore che genera automaticamente il codice per il parallelismo e ne definiscono il comportamento durante l'esecuzione, semplificando quindi le fasi di sviluppo.

Nell'interesse di sviluppare applicazioni parallele in grado di sfruttare le potenzialità della GPU altri strumenti sono stati sviluppati appositamente. Tra questi ricordiamo **OpenCL** [25], un framework per sviluppare applicazioni parallele su sistemi eterogenei, ovvero composti da più unità di elaborazione dalle caratteristiche differenti, ad esempio consente di utilizzare insieme un processore multi-core e una o più schede video. OpenCL fornisce un modo per accedere in maniera omogenea alle risorse offerte dal sistema, consentendo agli sviluppatori di algoritmi paralleli un modo per focalizzare lo sviluppo sulle operazioni di calcolo senza doversi occupare delle problematiche legate alla portabilità e delle problematiche implementative di basso livello.

Simile ad OpenCL, una soluzione alternativa (proprietaria) sviluppata da NVidea è **CUDA** [23]. Questo strumento si differenzia al precedente per avere caratteristiche legate al linguaggio di programmazione che lo rendono più semplice del precedente, di contro si tratta di uno strumento a disposizione solamente per le schede video NVidea, che ne limita pertanto l'applicabilità. Nonstante la limitazione del supporto hardware, CUDA è uno strumento diffuso e largamente impiegato anche nelle comunità scientifiche e molteplici articoli sono stati sviluppati con questo strumento. La caratteristica di essere più semplice da utilizzare lo rende particolarmente interessante in tutti quei contesti in cui il tempo di sviluppo deve essere contenuto e ci si vuole concentrare nell'analisi dei risultati ottenuti.

Un'altra soluzione disponibile per la programmazione parallela è **OpenACC** (*Open Accelerators*) [20] ed è uno standard per lo sviluppo di applicazioni parallele capaci di sfruttare architetture eterogenee CPU e GPU. Consente agli sviluppatori di produrre codice parallelo con semplici annotazioni che istruiscono il compilatore a generare codice parallelo, senza che lo sviluppatore si preoccupi di aggiungere istruzioni necessarie ad instanziare le risorse necessarie e gestire le differenti architetture parallele a disposizione.

Uno strumento alternativo per sviluppare applicazioni parallele su GPU fa uso di **OpenGL** [21][22], un framework impiegato per sviluppare applicazioni grafiche, dal rendering ai videogame. Utilizzare OpenGL per contesti di GPGPU può non essere semplice in quanto la programmazione deve affrontare problematiche legate all'impiego degli strumenti pensati per la computer graphics ed applicarli per scopi di calcolo parallelo. Questa tecnica appare per lo più utilizzata in tempi antecedenti gli altri strumenti più specializzati che consentono agli sviluppatori di concentrarsi sulle problematiche legate all'implementazione degli algoritmi senza doversi preoccupare degli altri aspetti. Tuttavia nei contesti in cui altri strumenti non sono disponibili, OpenGL risulta essere l'unica alternativa diffusa per gli sviluppatori che non vogliono programmare su GPU a livello di codice macchina. Come prima osservazione il codice macchina per ogni GPU può essere differente e quindi

il codice tende a non essere portatile; in secondo luogo la programmazione ed il debug di codice macchina non è certamente semplice, in modo particolare per la GPU, infine OpenGL con le sue molteplici varianti è uno strumento largamente diffuso in tutti i dispositivi che possiedono una scheda video.

Capitolo 2

Raspberry Pi

In questo capitolo viene descritto il Rasperry Pi, un piccolo calcolatore dal ridotto consumo energetico, le cui potenzialità di impiego sono state di interesse per appassionati e ricercatori, grazie ad una comunità attiva che ha consentito al prodotto di diffondersi. Descriveremo le principali caratteristiche tecniche dei vari modelli esistenti, alcuni degli impieghi possibili, infine le difficoltà e le possibilità in relazione al calcolo parallelo su questo dispositivo.

2.1 Descrizione

Il Raspberry Pi è un calcolatore di tipo *Single-Board* sviluppato con l'intento di promuovere l'insegnamento dell'informatica nelle scuole [33], riscuotendo anche l'interesse da parte di appassionati ed aziende per via dei costi contenuti e delle potenzialità di impiego per progetti di varia tipologia. Rilasciato inizialmente nel 2012, nel corso degli anni sono state prodotte diverse versioni di questo dispositivo.

Tra le caratteristiche che accomunano le diverse versioni si può osservare la CPU di tipo ARM, la scheda video *Broadcom VideoCore IV* integrata, RAM dai 256 MB ai 2 GB e slot per scheda SD usata sia per contenere il sistema operativo e come memoria di massa. Per il Raspberry Pi sono stati distribuiti diversi sistemi operativi GNU/Linux per architettura ARM. Tra

questi sistemi operativi ricordiamo i più rilevanti quali Raspbian (derivato da Debian), OSMC (*Open Source Media Center*) e Ubuntu.

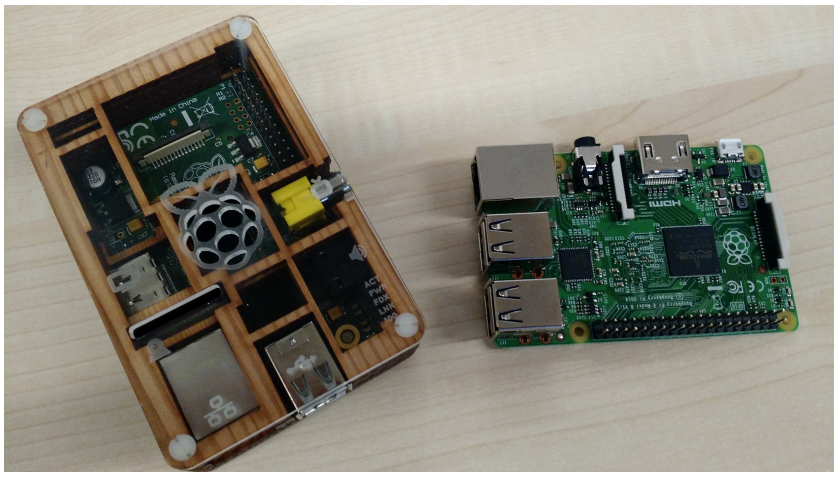


Figura 2.1: Raspberry Pi v.1B (sinistra) e v.2B (destra).

2.2 Caratteristiche

Nelle differenti versioni rilasciate del Raspberry Pi si può notare una modifica dell'hardware di cui questo è dotato. Le principali caratteristiche di ciascun modello vengono qui riportate:

Modello	SoC	CPU	GPU	RAM	Consumo
A	BCM2835	700 MHz single-core	VideoCore IV 24 GFLOPS	256 MB	300 mA (1.5 W)
A+	BCM2835	700 MHz single-core	VideoCore IV 24 GFLOPS	256 MB	200 mA (1 W)
B	BCM2835	700 MHz single-core	VideoCore IV 24 GFLOPS	512 MB	700 mA (3.5 W)
B+	BCM2835	700 MHz single-core	VideoCore IV 24 GFLOPS	512 MB	600 mA (3.0 W)
2B	BCM2836	900 MHz quad-core	VideoCore IV 24 GFLOPS	1 GB	800 mA (4.0 W)
Zero	BCM2835	1 GHz single-core	VideoCore IV 24 GFLOPS	512 MB	160 mA (0.8 W)

Tabella 2.1: Caratteristiche modelli Raspberry Pi

La tabella delle caratteristiche qui riportata è ovviamente ridotta agli aspetti essenziali per quanto riguarda questa analisi. Possiamo notare che a

livello hardware vi sono minime differenze e per quanto riguarda la GPU non vi è alcuna differenza. Il processore generalmente non ha prestazioni elevate ad eccezione del modello 2B che possiede un quad-core; gli altri possiedono solamente un core con frequenza poco elevata rispetto altre CPU ARM attualmente disponibili sul mercato (si pensi a molti degli smartphone in commercio). Per le versioni con un singolo core risulta evidentemente inutile eseguire applicazioni parallele.

Nonostante la potenza di calcolo ridotta è bene osservare che la GPU possiede delle caratteristiche interessanti se confrontate con le prestazioni dei rispettivi processori, infatti tutte le schede hanno a disposizione una GPU integrata Broadcom VideoCore IV a 24 GFLOPS. Le caratteristiche grafiche di un Raspberry Pi sono circa equivalenti a quelle offerte da una XBOX del 2001 [28], se in più si considera il basso consumo energetico di ogni scheda allora la valutazione complessiva di questo piccolo calcolatore può essere accetabile per alcuni utilizzi.

2.3 Motivazioni di impiego

Le motivazioni principali per le quali si preferisce utilizzare dispositivi Single-Board come il Raspberry Pi rispetto architetture più potenti sono inerenti i bassi consumi energetici e i bassi costi. Ovviamente il contesto di impiego gioca un ruolo fondamentale, nelle situazioni in cui i tempi di calcolo devono essere necesariamente contenuti entro limiti temporali probabilmente la scelta opportuna ricaderebbe nei sistemi HPC (*High Performance Computing*). Se invece si preferisce avere una buona potenza di calcolo con un rapporto prestazioni/prezzo accessibile allora l'impiego di architetture a basso consumo energetico possono essere preferibili.

Progetti innovativi puntano alla costruzione di mini-cluster di Raspberry Pi (o simili) al fine di poter costruire piccoli centri di calcolo parallelo. Sfruttare la potenza di calcolo unita di più calcolatori può essere effettivamente una soluzione adeguata per diversi contesti, in particolare quando si rivela

necessario elaborare grandi quantità di dati. Un esempio di questo impiego è presentato nel progetto *Raspèin* [29] sviluppato presso l'Università di Bologna, nell'ambito del quale è stato realizzato un cluster di Raspberry Pi con lo scopo di promuovere attività didattiche e di ricerca su applicazioni parallele.

2.4 GPGPU su Raspberry Pi

Attualmente programmare le GPU del Raspberry Pi risulta difficoltoso. La maggior parte degli strumenti impiegati dai programmatori per poter effettuare calcolo parallelo sfruttando le schede video non è disponibile per questo dispositivo. Non è possibile utilizzare CUDA in quanto si tratta di uno strumento sviluppato da NVidia per le proprie schede video mentre il Raspberry Pi utilizza una scheda video Broadcom, pertanto è ipotizzabile che il supporto non sarà mai disponibile. In alternativa a CUDA, uno degli strumenti maggiormente impiegati per lo sviluppo di applicazioni di calcolo parallelo è OpenCL, purtroppo questo non è in grado di sfruttare la GPU del Raspberry Pi [36] e si ipotizza che sia improbabile nel futuro ottenerne il supporto. Dalle informazioni che è stato possibile recuperare [35][34], si richiederebbe un considerevole investimento in termini di tempo e di sviluppatori necessari al porting di questo strumento per il dispositivo in questione e vi è uno scarso interesse da parte dei produttori nell'investire risorse in questo progetto a fronte di costi elevati ed uno stimato ridotto ritorno di investimento.

Altra limitazione legata all'impiego della GPU è dovuta alla scheda video presente ed ai driver disponibili. Infatti, sebbene il dispositivo in pochi anni si sia diffuso grazie anche all'interesse ed al contributo portato da appassionati con la condivisione di software libero, il driver video a disposizione si tratta di un blob binario proprietario[1] e le specifiche tecniche della scheda video non erano rilasciate. Il driver a disposizione possiede una interfaccia verso il kernel del sistema operativo che ne consente un uso limitato esclusivamente ad OpenGL ES 2 per il rendering, impedendo di fatto qualsiasi altro utilizzo

della GPU se non per gli usi consentiti con questo strumento. Non essendo disponibili le specifiche della scheda video era inoltre impossibile sviluppare un driver libero capace di sfruttare opportunamente l'hardware disponibile e con un supporto più esteso rispetto quello rilasciato da Broadcom.

Nel febbraio 2014 [1] venne rilasciata la documentazione tecnica relativa la scheda video al fine di consentire agli sviluppatori appassionati di poter rilasciare un driver libero per la GPU del Raspberry Pi, o comunque consentire un utilizzo più libero dell'hardware a disposizione per quali che siano gli scopi. Di fatto, a distanza di meno di un anno [2], sono stati sviluppati alcuni progetti che riuscivano ad utilizzare la scheda video per effettuare calcoli di diverso tipo. Questi software condividono un aspetto implementativo, ossia l'impiego della scheda video per calcoli di tipo GPGPU avviene a livello di linguaggio macchina con le istruzioni direttamente interpretabili dai processori video per la loro esecuzione. Tra alcuni esempi notevoli per questo contesto di utilizzo citiamo l'implementazione dell'algoritmo di hashing SHA-256 su GPU [5], il calcolo della FFT sviluppato da Andrew Holme [3] e Pete Warden con una applicazione di *deep learning* per il riconoscimento di immagini [4]. Questi software hanno consentito lo sviluppo di primi assemblatori [6] che tuttavia richiedono un tipo di programmazione estremamente a basso livello, detta anche "*bare metal programming*". Solitamente questo tipo di programmazione è considerata molto difficile per gli sviluppatori e ne segue che la realizzazione di applicazioni che operano a tale livello di granularità sia ostacolata dalla difficoltà stessa. Ad un livello leggermente superiore rispetto il linguaggio macchina può essere interessante considerare la libreria *PyVideoCore* [10] che consente di scrivere applicazioni in linguaggio Python e contestualmente definire del codice che deve essere eseguito sulla gpu, la libreria si occupa quindi di consegnare dati e codice e ricevere i risultati che vengono consegnati all'applicazione. Il codice da eseguire sulla GPU deve essere scritto in linguaggio macchina, tuttavia è possibile impiegare le funzionalità offerte da Python per una più agevole gestione dei dati da elaborare.

Parallelamente rispetto lo sviluppo di queste prime applicazioni GPGPU, la disponibilità delle specifiche tecniche della scheda video ha visto l'interesse di altri sviluppatori in merito la realizzazione di un driver libero per il Raspberry Pi [8, 9] e capace di supportare non solo OpenGL ES 2 ma anche la versione più completa disponibile nei desktop, OpenGL 2. Lo sviluppo di questo driver è in corso ma dai primi risultati sono state citate buone prestazioni ed un supporto quasi completo [7]. Tale driver è stato rilasciato come sperimentale nella verisione di Febbraio 2016 per il sitema operativo Raspian [37].

La disponibilità di OpenGL 2 rispetto OpenGL ES 2 risulta molto importante in merito lo sviluppo di applicazioni di tipo GPGPU. Il supporto esteso offerto dalla versione desktop consente di utilizzare degli strumenti che possono essere impiegati per sfruttare le potenzialità di calcolo della scheda video e consentendo lo sviluppo di applicazioni ad alto livello, quindi semplificando notevolemente rispetto le anaologhe implementazioni a livello codice macchina. La disponibilità di OpenGL ES 2, sebbene limitata per certi aspetti come il formato di rappresentazione dei dati ed altre caratteristiche implementative, consente di sviluppare applicazioni capaci di sfruttare la GPU per semplici calcoli paralleli, un approccio relativamente complesso ma che tuttavia consente di ottenere risultati soddisfacienti per alcune tipologie di applicazioni.

Capitolo 3

Calcolo Parallelo con OpenGL

In questo capitolo si vuole presentare la tecnica per utilizzare le librerie OpenGL al fine di eseguire calcolo parallelo su GPU. Da una prima presentazione delle OpenGL e OpenGL ES, evidenziandone le caratteristche e le rispettive differenze, si descriveranno gli elementi principali relativi alla GPGPU con queste librerie, quali gli shader, il FrameBuffer Object e le texture. Vengono inoltre descritte alcune delle modalità in cui questi strumenti possono essere utilizzati per implementare algoritmi paralleli adatti ad essere eseguiti sulla GPU.

3.1 OpenGL ES

La libreria *OpenGL* è uno strumento utilizzato per il rendering nelle applicazioni grafiche 3D, fornisce la possibilità di sfruttare efficacemente l'hardware grafico disponibile nel sistema [38]. La versione *OpenGL for Embedded Systems* (ES) è una variante ridotta di OpenGL pensata specificatamente per essere impiegata nei calcolatori con capacità hardware e software limitate rispetto le tradizionali macchine desktop [39], ad esempio è impiegata negli smartphone e nel Raspberry Pi. Le maggiori differenze presenti tra le varie versioni riguardano le funzionalità implementate nella *rendering pipeline*, rimuovendo le funzionalità deprecate e/o aggiungendo nuovi strumenti.

In OpenGL ES il criterio adottato per lo sviluppo è quello di rimuovere le funzionalità meno utili o il cui utilizzo può essere sostituito con altri strumenti, fornendo un'interfaccia leggera per l'utilizzo ma talvolta insufficiente a coprire tutte le necessità.

In OpenGL ES 1.x, ad esempio, non sono disponibili gli strumenti che consentono di utilizzare le funzionalità della pipeline programmabile, pertanto questa versione non può essere impiegata per la programmazione GPGPU. A partire da OpenGL ES 2.0 questo aspetto è stato cambiato ed è possibile utilizzare la pipeline programmabile mediante l'utilizzo degli *shaders* in modo simile a quanto possibile OpenGL 2.0 [40].

Gli *shader* hanno lo scopo di specificare il comportamento che la scheda video deve adottare in alcuni stage della pipeline grafica. La loro programmazione avviene mediante l'impiego del linguaggio GLSL (*OpenGL Shading Language*) compilato a run-time, fornisce una sintassi "C-like" per definire i costrutti base di programmazione (come *if*, *for* , ...). Per impiegare gli shader nella programmazione in OpenGL ES 2 è necessario definire almeno 2 shader di base: il *vertex shader* che si occupa di eseguire operazioni di trasformazione sui vertici dei poligoni e il *fragment shader* che si occupa di eseguire le operazioni relative ai colori dei frammenti che costituiscono un'immagine, detti *pixel* se relativi lo schermo oppure *texel* per le texture.

L'output degli shader è normalmente indirizzato nei buffer in memoria destinati allo schermo in modo da proiettare velocemente l'immagine risultate sul video, in alternativa è possibile reindirizzare l'output verso altri buffer, ad esempio utilizzando i FBO (Frame-Buffer Object) che definiscono aree di memoria predisposte alla ricezione del rendering, consentendo all'applicazione di riutilizzare i risultati prodotti.

OpenGL non supporta direttamente le funzionalità di GPGPU per realizzare programmi paralleli, tuttavia gli strumenti offerti per la *computer graphics* possono essere sfruttati con alcuni accorgimenti.

3.2 Impiego di OpenGL per GPGPU

Scrivere applicazioni che utilizzino opportunamente la scheda video per eseguire calcoli in parallelo non è generalmente un compito semplice in quanto richiede una buona comprensione sia degli algoritmi paralleli che degli strumenti esistenti per l'effettiva implementazione. Utilizzare OpenGL introduce un livello di difficoltà in più in quanto si richiede una buona comprensione della pipeline grafica riguardo le fasi di rendering, l'uso delle texture, degli shader, sia a livello concettuale che a livello implementativo, aggiungendo una difficoltà di comprensione per chi non ha mai avuto familiarità con gli aspetti di grafica 3D. Per di più, scrivere applicazioni che utilizzino OpenGL ES 2 è per certi aspetti più complicato rispetto utilizzare OpenGL 2 in quanto molte delle funzionalità utili presenti nella versione desktop non sono disponibili nella versione embedded, vincolandovincolando la realizzazione delle applicazioni e a volte rendendo il porting di un software scritto per la versione desktop sulla versione embedded un compito non banale.

Senza prendere in esame i dettagli implementativi, le operazioni concettuali che consentono di sfruttare le librerie grafiche per scopi di GPGPU possono essere descritte a prescindere dalla specifica libreria grafica impiegata. Gli unici requisiti necessari per l'impiego di questa tecnica è avere a disposizione una scheda grafica programmabile (non "*fixed-function pipeline*" ormai obsolete), avere la possibilità di utilizzare texture e poter eseguire il rendering su texture piuttosto che su schermo. Questi requisiti sono essenziali e dovrebbero essere tutti disponibili per le attuali schede grafiche, anche non necessariamente quelle di ultima generazione, tuttavia schede video particolarmente datate potrebbero non avere supporto ad alcune di queste funzionalità, in quel caso risulterebbero inutilizzabili. La scheda video del Raspberry Pi è in grado di supportare queste funzionalità quindi la tecnica in questione può essere applicata.

Per poter descrivere al meglio questa tecnica sono necessarie alcune associazioni tra i concetti di *computer graphics* ed i concetti di calcolo parallelo [11].

- **Texture = Dati**. Una delle modalità comunemente impiegate per la memorizzazione dei dati sui quali deve operare un algoritmo è quella di utilizzare vettori o matrici accedute per mezzo di indici. Analogamente, le texture sono matrici i cui elementi esprimono colori nei formati RGB, RGBA o altri formati disponibili a seconda delle implementazioni. Ogni cella della texture, chiamata *texel*, rappresenta il dato associato ad uno dei punti che compongono un'immagine e vengono acceduti mediante le relative coordinate. L'obiettivo è quindi individuare una rappresentazione dei dati idonea per essere espressa sotto forma di texture, l'input della procedura di calcolo parallelo. L'output deve essere analogamente una texture e questo può essere eseguito mediante l'impiego del *Frame Buffer Object* (FBO), un componente impiegato per far si che una scena resa non venga proiettata nello schermo ma venga mantenuta in memoria. Questo componente viene impiegato, ad esempio, se la scena resa deve essere salvata su file oppure deve essere impiegata come texture per la simulazione del riflesso di uno specchio, modificando opportunamente il punto di vista della scena. Nel contesto GPGPU invece vogliamo che i dati ritornino al controllo dell'applicazione dopo essere stati elaborati, sarà poi l'applicazione stessa a trattarli opportunamente.

- **Shader = Kernel**. Lo shader è il programma che viene eseguito sulla GPU, il kernel è invece la funzione che deve essere eseguita parallelamente su tutti i dati da elaborare. Gli shader comunemente considerati nel rendering sono il *vertex shader* che si occupa di eseguire operazioni su ogni vertice ed il *fragment shader* che si occupa di eseguire operazioni su ogni texel, principalmente per produrre il colore che ciascun texel deve assumere (ad esempio, in ambito grafico può definire il contributo della luce in un texel durante la resa di una scena). In ambito di GPGPU, poiché i dati sono rappresentati nelle componenti di colore di una o più texture, siamo interessati ad operare nel fragment shader per eseguire calcoli di vario tipo sugli input e produrre un output

rappresentato dai texel della scena resa.

- **Rendering = Calcolo**. Per poter eseguire le operazioni di calcolo su tutti gli elementi della texture occorre che gli shader eseguano l'operazione di rendering di tutta la scena, ovvero ogni texel deve essere elaborato nel fragment shader e quindi reso. Il modo più semplice di eseguire questa operazione è disegnare un quadrato che ricopra interamente le dimensioni della viewport. Nel fragment shader quindi ad ogni coordinata del texel generato in output corrisponderà la rispettiva coordinata della texture in input.

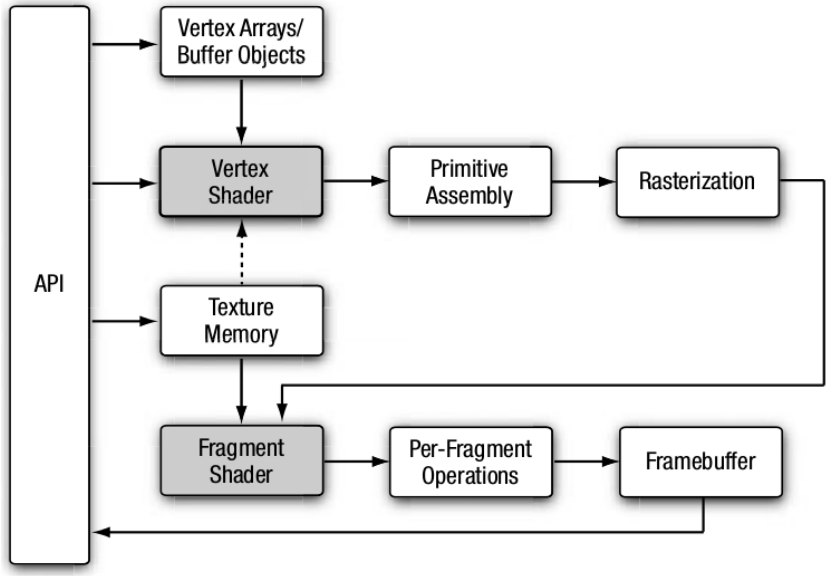


Figura 3.1: Pipeline grafica in OpenGL ES 2 (fonte: *OpenGL ES 2.0 Programming Guide*, p.4 [46]).

Riassumendo, per poter eseguire calcoli GPGPU con OpenGL è necessario che i dati in input ed output siano memorizzati in delle texture, il fragment shader si occuperà di effettuare il calcolo opportuno per ogni texel sfruttando le componenti di colore di ognuno di questi. Per richiamare l'esecuzione della GPU sulla texture sarà sufficiente disegnare un quadrato della dimensione della viewport dopo aver assicurato che non avvenga perdita di

informazione in trasformazioni window-viewport, che non vengano generati artefatti durante la fase di resa e che il vertex shader si limiti ad eseguire una proiezione ortogonale del quadrato disegnato sulla texture mantenendo un rapporto di 1:1 tra le coordinate dei texel e quelle dei frammenti. Questa tecnica consente una singola istanza del kernel su tutte le informazioni date in input mediante la texture, riutilizzando questo approccio per ogni istanza di calcolo parallelo si possono sviluppare algoritmi paralleli per gpu in assenza del supporto di appositi framework.

3.3 Limitazioni a GPGPU in OpenGL

La tecnica presentata è stata impiegata in passato in assenza di strumenti più idonei al calcolo parallelo come CUDA o OpenCL, pertanto già verificata in studi antecedenti [12]. Alcuni aspetti devono tuttavia essere presi in considerazione per il suo impiego. In particolare occorre considerare alcune delle difficoltà legate all'impiego di OpenGL che potrebbe apparire piuttosto ostile a chi si cimenta per le prime volte alla programmazione, chi non possiede le basi di *computer graphics* e chi non ha mai sviluppato codice utilizzando OpenGL.

All'atto pratico implementare in questo modo algoritmi paralleli, anche piuttosto semplici, può risultare difficoltoso ed il debugging spesso non banale. Alcuni degli errori di implementazione potrebbero non essere immediatamente segnalati se non eseguendo opportuni controlli a run-time, ad esempio per l'utilizzo di funzionalità non supportate dal sistema per le quali il calcolo procede nella sua esecuzione trascurando il problema, per poi non generare nessun output oppure producendo risultati differenti rispetto quelli attesi. Un esempio è la scelta della dimensione delle texture: il corretto mapping sulle coordinate delle texture è garantito sempre per texture quadrate con dimensioni potenze di 2 (ovvero 1x1, 2x2, 4x4, 8x8, 16x16, ...) fino alla dimensione massima che nel caso di Raspberry Pi è 1920x1200 [41]. Dimensioni differenti rispetto a quelle supportate potrebbero causare un mapping non

corretto sulle coordinate delle texture, invalidando i risultati ottenuti dall'elaborazione. Per questioni legate all'allineamento dei dati in memoria, ovvero come queste vengono memorizzate e trasferite alla gpu, texture non quadrate producono risultati attendibili purché ciascuna dimensione della texture sia potenza di 2 (ad esempio una texture di dimensioni 1x256 viene elaborata correttamente). Questo significa che è scorretto elaborare una matrice di dimensioni 6x6 in quanto produrrebbe risultati errati, mentre una matrice 8x8 sarebbe invece corretta. Per questo particolare errore non viene generato nessun messaggio o segnalazione, rendendo problematico individuare la causa degli errori di elaborazione ottenuti in output. Le dimensioni delle texture sono soggette ai vincoli descritti in base alle versioni di OpenGL impiegate ed in base alle caratteristiche della scheda video, pertanto questo aspetto non può essere generalizzato in tutti i casi, sebbene le texture quadrate con dimensioni potenze di due, dette *texture POT*, sono lo standard generalmente sempre supportato. Nel caso del Raspberry Pi ogni versione fino ad ora rilasciata integra la stessa GPU quindi eventuali ottimizzazioni specifiche possono essere riutilizzate.

Un altro aspetto che limita l'impiego della tecnica per il calcolo parallelo su GPU è il formato di dati supportato dalle texture in OpenGL ES 2, ovvero i valori dei texel non supportano il tipo float. Stando alla documentazione i tipi di dato accettati per i valori dei texel sono: GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT_5_6_5, GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_5_5_5_1. In pratica questo significa che possiamo utilizzare solo valori interi positivi contenuti in 8 bit (0-255) per GL_UNSIGNED_BYTE. I valori per GL_UNSIGNED_SHORT_* non aiutano molto in generale, almeno per quanto riguarda l'aspetto di GPGPU, ovvero ogni numero che segue nel tipo "short" non mappa ogni componente di colore in 16 bit e non ci consente di avere a disposizione valori compresi tra 0 e $2^{16} - 1$, bensì i 16 bit vengono suddivisi tra le singoli componenti di colore (ad esempio GL_UNSIGNED_SHORT_5_6_5 in una texture RGB utilizza 5 bit per il rosso, 6 bit per verde e 5 bit per il blu). L'impiego di

questi formati di dato può essere vantaggioso in casi particolari legati alle applicazioni di *computer graphics*. Per quanto ci riguarda possiamo trarre maggior vantaggio utilizzando texture nel formato RGB o RGBA e tipo di dato GL_UNSIGNED_BYTE, ovvero memorizzare in ogni texel 3 o 4 componenti di colore e ciascuno dei quali con 8 bit a disposizione, per un massimo di 24 o 32 bit per texel. Non avere a disposizione valori negativi o in virgola mobile può essere molto limitativo e costringe a cercare soluzioni appropriate a seconda dei casi.

Nonostante le limitazioni presenti nei formati dei dati delle texture, quindi gli input e gli output del processo di calcolo parallelo, occorre ricordare che invece gli shader operano in virgola mobile ed è possibile consegnare dei singoli valori in virgola mobile facendo uso delle funzioni `glUniform`. Queste funzioni consentono di consegnare valori agli shader al di fuori delle texture, quindi si possono specificare parametri che lo shader potrà utilizzare per effettuare calcoli, ad esempio indicare le dimensioni delle texture o l'offset da utilizzare per accedere ai valori dei texel vicini. In altri termini, i tipi di dato in input ed output sono soggetti a limitazioni tuttavia le operazioni di calcolo possono utilizzare valori interi a virgola mobile, costringendo a sviluppare algoritmi che fanno un uso attento sia delle limitazioni che delle potenzialità.

Un altro aspetto che occorre tenere presente è che sia le dimensioni delle texture, sia i valori delle componenti dei texel, sono convertiti in valori in virgola mobile compresi tra 0 ed 1. Quindi, per operare opportunamente negli shader, si rivela necessario convertire i valori in input, ad esempio se un texel ha le componenti RGB=(0, 128, 255) nello shader lo stesso texel avrà componenti RGB=(0, 128/255, 1). Analogamente una texture di dimensione 128x128 nello shader avrà dimensioni 1x1, questo aspetto è da tenere bene in considerazione se si ha bisogno di confrontare il valore di un texel con quello dei suoi vicini, occorre calcolare l'offeset che deve essere associato alle coordiante per poter accedere alle locazioni corrette.

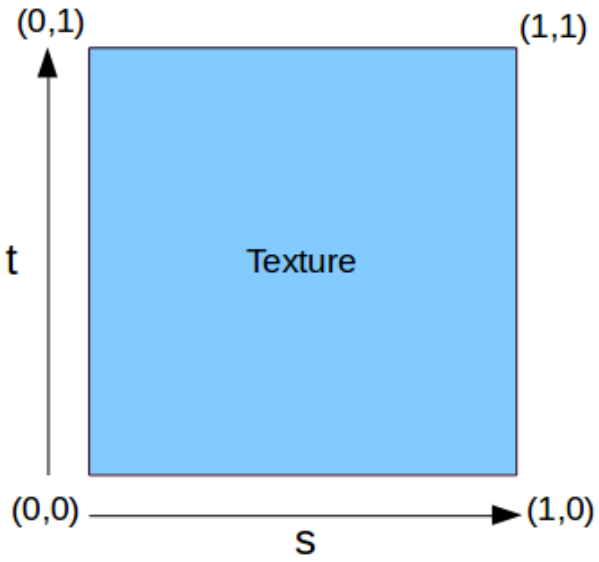


Figura 3.2: Coordinate texture.

Le limitazioni e le difficoltà riportate sono alcune tra le principali, tuttavia anche altre limitazioni sono presenti in OpenGL ES 2 rispetto OpenGL 2, ad esempio non è possibile disegnare un quadrato (GL_QUADS) come figura geometrica primitiva, elemento utilizzato per eseguire la procedura di calcolo sulla texture, inoltre non sono disponibili alcuni buffer di output che potrebbero essere utilizzati per ottenere ulteriori valori in uscita dal fragment shader. Queste limitazioni sono dovute a scelte implementative che eliminano il supporto ad elementi deprecati o non necessari in favore delle modalità considerate più idonee per effettuare rendering 3D. Queste limitazioni comportano principalmente che codice precedentemente sviluppato per OpenGL 2 non sia immediatamente compatibile con OpenGL ES 2 ma richiede un porting attento e in alcuni casi non semplice. Nel caso del disegno del quadrato è infatti possibile in un unico passo generare 2 triangoli (GL_TRIANGLE_STRIP) che coprano la stessa superficie del quadrato evitando sovrapposizioni, ottenendo quindi lo stesso risultato. Si deduce quindi che implementare algoritmi paralleli in OpenGL ES 2 non sia un compito semplice ma nemmeno impossibile, richiede un'attenta valutazione dell'algo-

ritmo che si vuole implementare, considerare tutte le limitazioni e disporre di una buona conoscenza di come programmare utilizzando questo strumento.

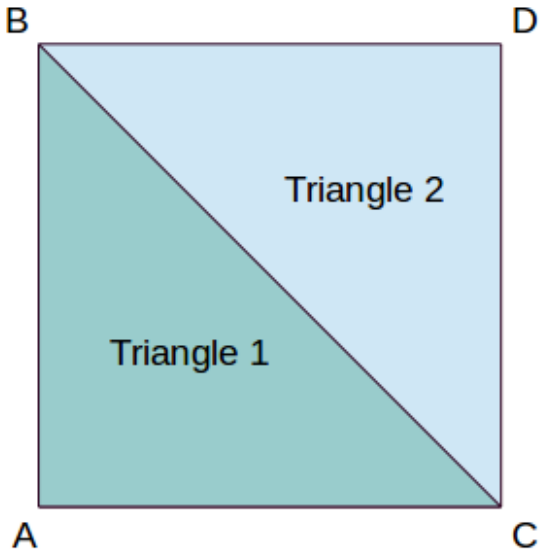


Figura 3.3: Un quadrato con GL_TRIANGLE_STRIP.

3.4 Algoritmi paralleli in OpenGL

La tecnica di eseguire calcolo parallelo sulle GPU utilizzando shader e texture può essere molto utile nelle situazioni in cui non sono presenti strumenti più evoluti per GPGPU, come CUDA o OpenCL. In queste situazioni, se si vuole evitare di scrivere direttamente in codice macchina le istruzioni che devono essere eseguite sulla GPU, questa tecnica può essere una buona alternativa in grado di offrire un compromesso tra facilità e potenzialità, nonostante le limitazioni descritte.

Questa tecnica se utilizzata in un singolo passo può essere limitativa e può non essere sufficiente coprire tutti gli scopi necessari, ad esempio perché le stesse istruzioni vengono eseguite parallelamente su tutto l'insieme di dati, sia per il vincolo imposto sul numero di valori in output, sia per il controllo limitato che ogni unità di calcolo della GPU possiede sui dati in input. In

questo caso il problema può essere risolto eseguendo l'istanza di calcolo su GPU in più passi, cambiando le istruzioni da eseguire, cambiando i valori in input oppure utilizzando in input i valori di output di una precedente elaborazione. Usando più passi di elaborazione possono essere realizzati algoritmi paralleli come composizione di operazioni semplici su insiemi di dati [12].

Un buon modello di astrazione per il calcolo su GPU fa riferimento allo *stream processing*, utilizzando insiemi di dati detti *stream* ed operazioni fondamentali eseguite su questi, dette *kernel*: *map*, *reduce*, *scatter*, *gather*, *scan*, *filter*, *sort*, *search*. A tale proposito fa riferimento alle texture come a degli stream, in quanto entrambi rappresentano collezioni di dati che devono essere elaborati.

3.4.1 Map

La funzione *map* è la più semplice da implementare in quanto richiede che una data funzione venga eseguita su tutti i valori dello stream in input. In questo caso è sufficiente implementare la funzione da eseguire all'interno del fragment shader e questa verrà eseguita su tutti i texel della texture, cioè su tutti gli elementi dello stream. L'output verrà reso sulla texture in memoria e pertanto l'operazione richiede un singolo passo di calcolo parallelo per gli n valori in input, quindi richiedendo tempo $O(1)$ se il numero di unità di calcolo p della GPU sia $p \geq n$, altrimenti richiederà tempo $O(\frac{n}{p})$.

3.4.2 Reduce

Questa operazionie consiste nel ridurre un insieme di dati in un singolo valore, ad esempio nella media, nel massimo o nella somma dei valori in input. Senza ulteriori assunzioni sui valori in input normalmente queste operazioni richiedono $O(n)$ passi di esecuzione per un algoritmo sequenziale. Operando in parallelo possiamo dividere la texture in input su due texture di dimensioni dimezzate, confrontare i due valori presenti nelle rispettive coordinate e produrre un unico valore nella texture in output. Questa operazione richiede

$\log n$ passi di esecuzione parallela, quindi se il numero delle unità di calcolo p a disposizione della GPU sia $p \geq \frac{n}{2}$ allora il tempo richiesto sarà $O(\log n)$, altrimenti richiederà tempo $O(\frac{n}{p} \log n)$. Per diminuire ulteriormente il numero di passi paralleli necessari alla computazione è possibile suddividere la texture in 4 parti (o un'altra potenza di 2), riducendo le dimensioni della texture più rapidamente rispetto che dividendola per 2. Aumentare il livello di suddivisione della texture ovviamente comporta per lo shader di eseguire più calcoli e confronti per texel, tuttavia richiede anche meno operazioni di trasferimento dei dati alla memoria, pertanto occorre una valutazione attenta anche in base alle prestazioni della GPU per decidere il livello migliore di suddivisione della texture al fine di ottenere un miglioramento di prestazioni (ad esempio, nel caso degenere una singola unità di calcolo della GPU esegue il test su tutta la texture, riconducendosi effettivamente nel comportamento a quello della versione sequenziale).

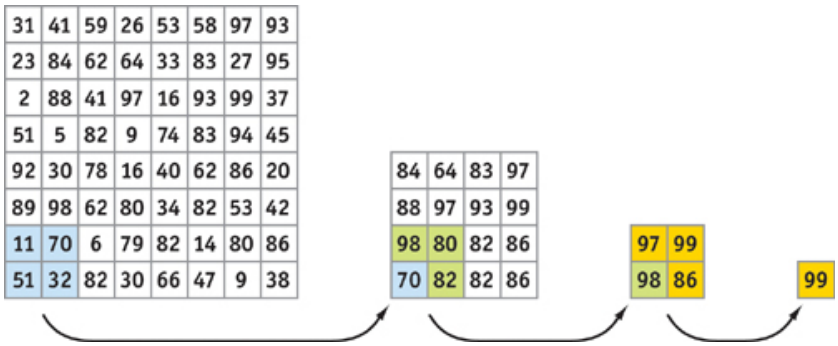


Figura 3.4: Passi di applicazione della reduce per valore massimo (fonte: *GPU Gems*, ch. 37 [43]).

3.4.3 Scatter e Gather

Le operazioni *scatter* e *gather* possono assumere un significato differente a seconda del contesto in cui vengono utilizzate. In ambito di operazioni relative l'algebra lineare, ad esempio in BLAST [13], assumono un significato di accesso indiretto ad un'area di memoria mediante l'impiego di indici: l'operazione gather può essere esemplificata in `x[i] = y[idx[i]]` mentre

l'operazione scatter in `y[idx[i]] = x[i]`, dove x e y sono vettori di dati mentre *idx* è un vettore di indirizzi di memoria. In questo contesto l'operazione di gather risulta essere immediatamente disponibile con il passaggio dell'indirizzo di memoria dei dati nella costruzione della texture, quindi trasferendo la texture nella GPU. L'operazione di scatter è invece più complessa da realizzare in quanto per ogni texel il fragment shader è vincolato in scrittura nelle coordinate di riferimento della texture di lettura, sebbene alcune soluzioni consentono di risolvere il problema riducendo l'operazione di scatter in un'operazione di gather [14].

Altro modo di considerare le operazioni scatter e gather sono previste in MPI e sono intese come funzioni per la comunicazione dei dati tra più unità di calcolo. In questo contesto, scatter ha il compito di comunicare porzioni di dati da elaborare a più unità di calcolo in modo che tutti i processi dispongano di una parte dell'insieme, gather invece ha il compito di raccogliere i risultati dell'elaborazione di più processi. Si può quindi notare che da queste definizioni, le operazioni di scatter e gather sono immediatamente disponibili mendiante il trasferimento delle texture dalla memoria centrale alla GPU, e viceversa. Simile a scatter, altra funzione utile disponibile in MPI è *broadcast* che si occupa di propagare un unico valore su più unità di calcolo. Nel caso degli shader l'analoga funzione disponibile è *uniform* che permette esattamente lo stesso tipo di comportamento.

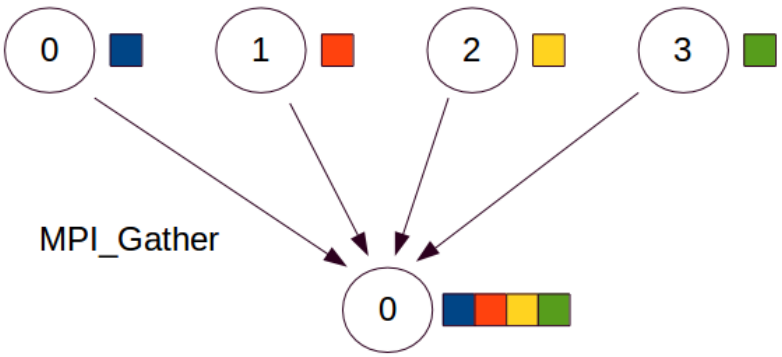


Figura 3.5: Gather in MPI.

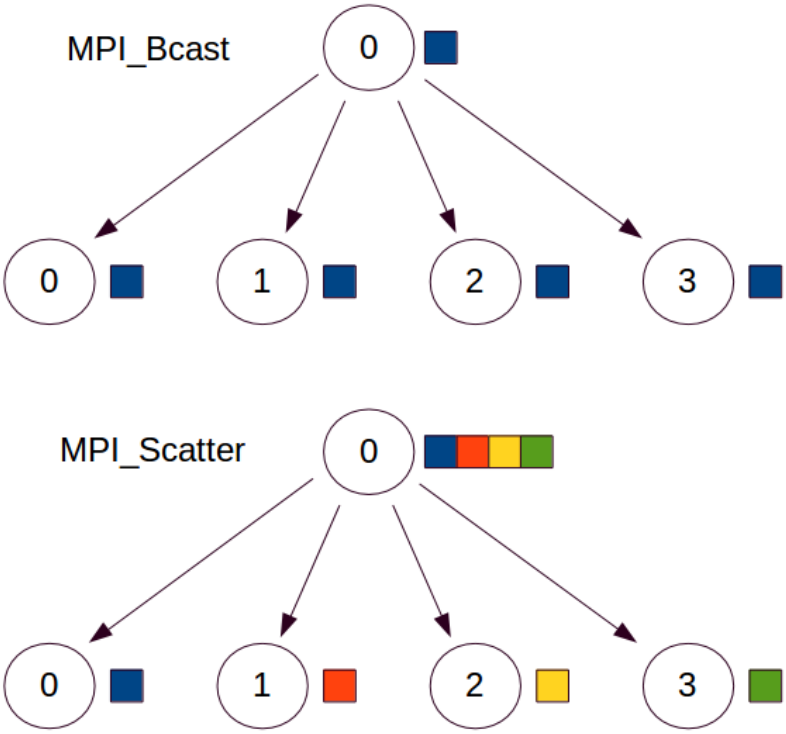


Figura 3.6: Broadcast e Scatter in MPI.

3.4.4 Scan

L'operazione *scan* è una generalizzazione della somme prefisse, ovvero dato un vettore x di n elementi viene generato un secondo vettore y di n elementi in cui vale la relazione di ricorrenza $y_i = f(y_{i-1}, x_i)$ e $y_0 = x_0$ dove f è una funzione binaria applicata mediante la scan agli elementi del vettore. Nel caso delle somme prefisse la funzione applicata è la somma, quindi vale la relazione $y_i = y_{i-1} + x_i$.

Nell'implementazione parallela della scan può risultare non banale garantire un effettivo miglioramento delle prestazioni, infatti questa esibisce delle dipendenze sui dati da elaborare. La funzione da applicare può avere delle caratteristiche che non ne consentono una parallelizzazione efficiente. Se invece presenta un operatore con proprietà associativa in tal caso può

essere facilmente parallelizzata. A titolo esemplificativo si osservi la figura 3.7 che rappresenta graficamente la versione semplificata della somma prefissa. Questa versione esegue $O(n \log_2 n)$ operazioni di addizione sfruttando n processori. Si possono tuttavia individuare varianti di questo algoritmo più efficienti affinché svolgano meno calcoli per ottenere il risultato desiderato.

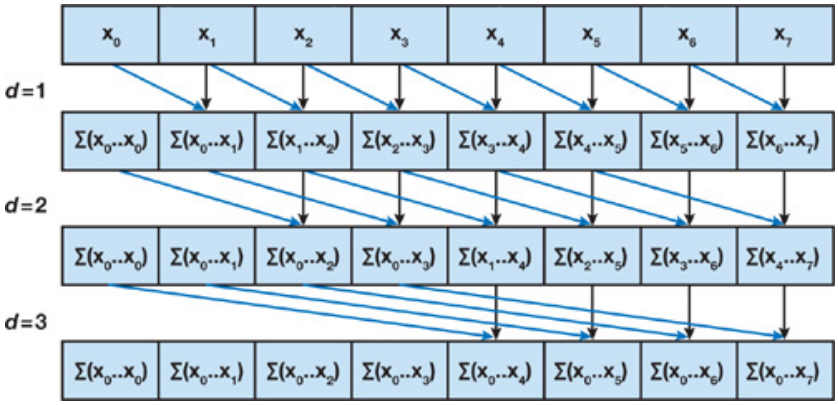


Figura 3.7: Scan per somme prefisse (fonte: *GPU Gems 3*, ch. 39 [45]).

3.4.5 Filter

La funzione *filter* consiste nella riduzione di un insieme di elementi in modo tale da ottenere solo quelli che soddisfano una certa proprietà. Utilizzando una combinazione di *scan* e *search*, questa operazione può essere eseguita in $O(\log n)$ passi di esecuzione parallela [15].

3.4.6 Sort

Il *sort* consiste nell'ordinamento dei valori di un vettore, si tratta di un algoritmo classico per CPU e del quale sono state implementate numerose varianti (tra le più note merge sort, quick sort, bubble sort, ...). Gli algoritmi di ordinamento per CPU presentano delle caratteristiche che difficilmente riescono ad essere implementate su GPU mantenendo delle buone prestazioni, si pensi al problema dello scatter inteso come l'accesso indicizzato in scrittura ad una posizione di memoria differente da quella in cui l'unità di calcolo stà

operando. Per garantire un efficiente impiego delle risorse della GPU sono necessari degli algoritmi che non richiedono l'impiego di operazioni di scatter.

Molti degli algoritmi di ordinamento per GPU, per evitare il problema del'accesso indicizzato in scrittura, sono basati su reti di ordinamento. L'idea principale di questi algoritmi consiste nell'impiego di queste reti e di effettuare un ordinamento in un numero finito di passi. I nodi della rete hanno un pattern di comunicazione prefissato che consente di ricondurre il problema legato all'utilizzo dello scatter sostituendolo con il gather, più facile ed efficiente da implementare. In questo modo si possono realizzare algoritmi di ordinamento per GPU che possiedono generalmente una complessità di calcolo nell'ordine di $O(n \log^2 n)$, dei quali l'esempio più significativo è il *Bitonic Merge Sort* [16].

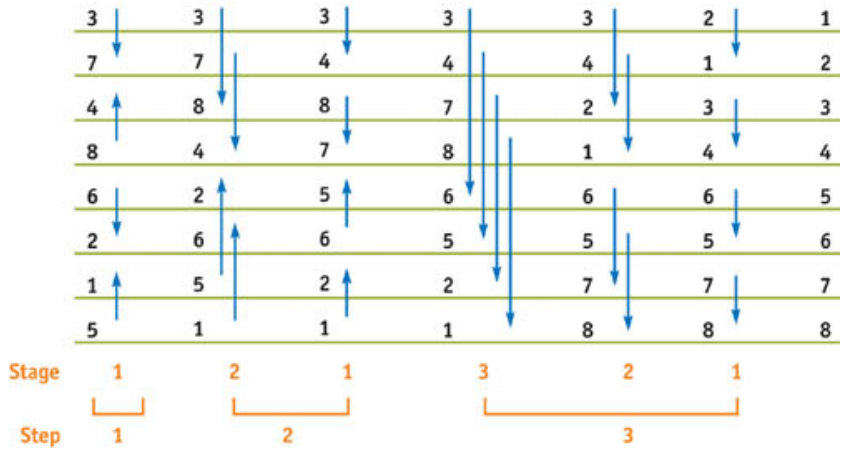


Figura 3.8: Bitonic Merge Sort con 8 elementi (fonte: *GPU Gems*, ch. 37 [43]).

3.4.7 Search

L'ultima operazione fondamentale considerata per operare su stream di dati è la *search*, la quale consiste nella ricerca di un valore in un insieme di dati. Gli algoritmi di ricerca implementati possono essere valutati in base alle collezioni dati utilizzati, ad esempio in presenza di un vettore ordinato

di valori l'algoritmo comunemente più utilizzato è quello di ricerca binaria. In questo caso la tecnica sequenziale consiste nel confrontare l'elemento centrale della lista con il valore cercato, a seconda del risultato del confronto la ricerca procede nella prima o nella seconda metà della lista, oppure termina in caso di elemento trovato. Nella versione parallela è sufficiente suddividere la lista in più frammenti pari al numero di unità di calcolo disponibili sulla GPU, quindi confrontare l'elemento ricercato con tutti i valori centrali dei frammenti assegnati. Nella versione sequenziale si ha che la ricerca binaria possiede costo computazionale di $O(\log n)$, similimente la versione parallela con p unità di calcolo richiede un costo pari a $O(\log_p n)$.

Se i dati in input non sono ordinati si ha che la ricerca binaria non può essere applicata, in questo caso si può procedere con una variante della reduce che punta a ridurre la dimensione dei dati in input ad ogni passo di esecuzione parallella fino ad ottenere un solo valore. Al termine della computazione sarà sufficente confrontare il valore ottenuto con il valore cercato, la ricerca avrà avuto successo se questi due valori saranno uguali.

Capitolo 4

Seam Carving

In questo capitolo verrà presentato l'algoritmo di Seam Carving nelle sue implementazioni sequenziale e parallela.

4.1 Descrizione dell'algoritmo

Per valutare l'applicabilità delle tecniche descritte ad un caso reale e poter evidenziare sia le potenzialità che le difficoltà derivanti dal loro impiego, si è deciso di implementare l'algoritmo di *Seam Carving* [17] come caso di studio; questo algoritmo consente il ridimensionamento delle immagini in modo consapevole del contenuto.

Ipotizziamo di prendere un'immagine e volerne ridurre la larghezza di 50 pixel. Utilizzando una tecnica di scalatura si otterrà che tutta l'immagine verrà ristretta, alterando le proporzioni degli oggetti rappresentati in essa. Una tecnica di taglio invece eliminerà interamente una parte dell'immagine, le proporzioni verranno mantenute ma alcune informazioni importanti potranno perdersi. In contrapposizione, Seam Carving punta ad individuare delle cuciture all'interno dell'immagine la cui rimozione comporterà la minima perdita di informazione.

(a) Immagine originale

(b) Immagine scalata

(c) Immagine ritagliata

(d) Seam Carving

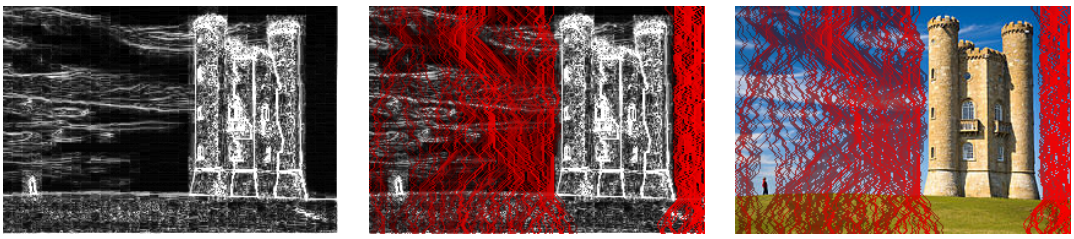
Figura 4.1: Confronto tra le tecniche di ridimensionamento (fonte: Wikipedia, *Seam Carving* [42]).

Per far questo occorre valutare l'importanza di ogni pixel dell'immagine in base al contrasto di colore con i pixel vicini, tale valutazione può avvenire con una qualsiasi funzione in grado di misurare l'energia, il peso, la densità (va bene qualsiasi funzione capace di assegnare un valore di rilevanza per ciascun pixel). Dopo aver individuato i pesi occorre individuare la cucitura di peso minore, ottenuta sommando tutti i pesi dei pixel che compongono le cuciture. Ogni pixel di coordinate (i, j) può appartenere a 3 cuciture verticali con i pixel di coordinate $(i-1, j-1)$, $(i-1, j)$ o $(i-1, j+1)$. Dopo aver individuato la cucitura di peso minore si può procedere con la sua rimozione. L'equazione di ricorrenza nel calcolo delle cuciture è la seguente:

$$W[i, j] = E[i, j] + \min\{W[i-1, j-1], W[i-1, j], W[i-1, j+1]\}$$

Un algoritmo sequenziale efficiente per il calcolo delle cuciture viene im-

plementato con una tecnica di programmazione dinamica. Consideriamo l'immagine come una matrice di pixel e ad ogni elemento assegniamo il valore di energia corrispondente, l'algoritmo valuta in ordine dalla seconda all'ultima riga tutta la matrice. La prima riga può essere trascurata in quanto non vi sono righe precedenti, invece per le righe successive si valuta ogni cella e per ciascuna di queste viene individuato il costo minimo tra le 3 celle superiori adiacenti. Per la prima e l'ultima cella di ogni riga vengono valutate solo le due celle possibili. Ad ogni cella esaminata viene sommato il costo della cella attuale con il minimo costo tra le tre possibili celle candidate. Al termine dell'analisi della matrice, l'ultima riga possederà il costo minore per ogni cucitura che termina in ciascuno dei pixel. Sarà quindi sufficiente individuare la cella con costo minore e procedere a ritroso da questa individuando i costi minori tra le 3 celle superiori fino a raggiungere la prima riga della matrice. Il percorso individuato corrisponderà alla cucitura di costo minimo relativa all'immagine originale e si potrà procedere con la rimozione dei pixel associati, riducendo la larghezza dell'immagine originale di 1 pixel. La procedura può essere ripetuta n volte al fine di rimuovere le n cuciture di costo minore.



(a) Calcolo dei pesi (c) Cuciture minime (d) Cuciture sulla foto

Figura 4.2: Passi intermedi dell'algoritmo di Seam Carving (fonte: Wikipedia, *Seam Carving* [42])

L'algoritmo presentato può essere utilizzato sia per rimuovere cuciture orizzontali che cuciture verticali, sarà sufficiente ruotare l'immagine oppure modificare l'ordine di esplorazione della matrice affinché venga esaminata nella direzione desiderata. Seam Carving può essere utilizzato per aumentare

la dimensione interessata di un immagine invece che restringerla, in questo caso sarà sufficiente calcolare la cucitura di costo minimo ed inserire pixel mancanti che possono essere calcolati come media dei valori di colore dei pixel adiacenti.

Per una immagine di larghezza w e di altezza h, il costo dell'algoritmo sequenziale così come descritto è $O(wh)$. Infatti, data una immagine qualsiasi rappresentata come matrice di valori (ad esempio ogni cella può rappresentare le componenti RGB del relativo pixel), calcolare il valore di energia di ogni pixel mediante una funzione che prende in esame solamente i pixel adiacenti costerà $O(wh)$, ovvero si tratta di applicare una funzione con costo $O(1)$ su tutti i $w \times h$ pixel. Analoga situazione si verifica nel calcolo delle cuciture di costo minimo con programmazione dinamica, infatti si esanineranno $h-1$ righe e w celle per riga, per ogni cella si esegue una somma ed il calcolo del minimo tra 3 valori che ha costo computazionale $O(1)$, quindi anche in questo caso il costo complessivo sarà $O((h-1) \times w) = O(wh)$. Calcolare il minimo tra w valori ha costo computazionale $O(w)$ e quindi determinare ricorsivamente la cucitura di costo minimo a partire dal valore individuato avrà costo $O(h)$. Complessivamente il costo computazionale dell'algoritmo sequenziale è dunque $O(wh)$.

4.2 Algoritmo parallelo

L'implementazione parallela del Seam Carving è stata svolta individuando gli aspetti parallelizzabili del rispettivo algoritmo sequenziale e cercando di trarre il massimo vantaggio possibile dall'impiego della GPU. In questo caso l'aspetto più facile da parallelizzare riguarda il calcolo dell'energia di ogni pixel in quanto ciascun calcolo può essere eseguito indifferentemente dagli altri con approccio "data parallel". In questo modo, ipotizzando di elaborare un'immagine di altezza h e larghezza w, il costo computazionale con infiniti processori sarà $O(1)$ in quanto gli $h \times w$ pixel potranno essere elaborati parallelamente; avendo a disposizione p processori il costo sarà $O(wh/p)$.

Per calcolare tutte le possibili cuciture dell'immagine non è invece possibile procedere in parallelo su tutti i pixel dell'immagine ma occorre valutare riga per riga ed utilizzare sia l'output della computazione precedente che i valori di energia di ogni pixel della riga attuale. In altri termini, è possibile effettuare calcolo parallelo su tutti i pixel della riga i utilizzando i valori di costo minimo delle cuciture alla riga $i-1$, l'output di questo calcolo sarà l'input al momento di valutare la riga $i+1$. In questo caso il costo computazionale sarà $O(h)$ in quanto è possibile elaborare parallelamente w pixel per ogni riga dell'immagine.

Dopo aver elaborato tutte le cuciture dell'immagine occorrerà individuare la cucitura di costo minimo. La ricerca del minimo è uno degli algoritmi di base e parallelizzando opportunamente si ottiene un costo computazionale pari a $O(\log w)$ in quanto la ricerca interesserà solamente l'ultima riga della matrice con i costi delle cuciture.

Infine, l'ultimo passaggio consiste nell'individuare tutte le coordinate dei pixel che compongono la cucitura e rimuoverli dall'immagine. Questa operazione purtroppo è strettamente sequenziale poiché ad ogni passo di elaborazione si valuta a partire dalle coordinate di un solo pixel e il passo seguente potrà iniziare dalle coordinate di uno dei 3 dei possibili pixel adiacenti candidati. In questo caso si ha che il costo computazionale non può essere ridotto usando unità di calcolo parallele e continuerà ad essera $O(h)$.

Da queste osservzioni si deduce che il costo computazionale utilizzando p unità di calcolo parallele sia $O(wh/p)$. Questa implementazione del Seam Carving parallelo risulta più efficiente per immagini particolarmente larghe piuttosto che per immagini di altezza elevata. Questa caratterisitca è dovuta essenzialmente dal calcolo della cucitura minima parallelizzando il processo sulla larghezza dell'immagine, attendendo che il calcolo di una riga sia terminato prima di procedere alla successiva.

4.3 Implementazione dell'algoritmo parallelo

Data la descrizione formale dell'algoritmo parallelo che consente di considerare il problema da un punto di vista più astratto, l'implementazione deve affrontare le difficoltà legate all'impiego di OpenGL ES 2. L'impiego delle texture vincola fortemente il tipo di dati che possiamo consegnare ed ottenere agli shader. Se le texture sono di tipo RGB allora ogni elemento della matrice potrà contenere al massimo 3 valori di tipo "unsigned char", ovvero interi compresi nel range $0-255$. Nonostante il limite legato al trasferimento dei dati, gli shader possono eseguire operazioni in virgola mobile.

Come si è descritto, possiamo individuare tre elementi parallelizzabili:

1. Il calcolo dei valori di similitudine per ogni pixel rispetto i vicini.

2. Il calcolo delle cuciture.

3. Il calcolo della cucitura minima.

Ipotizziamo di avere in input la sequente immagine:

Figura 4.3: Una foto in input per Seam Carving.

Il calcolo parallelo viene effettuato nel fragment shader, tuttavia occorre definire anche il vertex shader per poter impiegare la pipeline grafica. Per fare questo è sufficiente definire la semplice proiezione ortogonale del quadrato nella scena, sul quale verrà eseguito il calcolo mediante fragment shader con i valori della texture.

```
attribute highp vec4 aPos;
attribute highp vec2 aTexCoord;
varying highp vec2 vTexCoord;

void main() {
        gl_Position = aPos;
        vTexCoord = aTexCoord;
}
```

Successivamente nel fragment shader verrano definiti i calcoli per determinare la similitudine dei pixel facendo uso di una funzione di energia con gradiente duale:

$$E(x, y) = \Delta_x^2(x, y) + \Delta_y^2(x, y)$$

con

$$\Delta_x^2(x, y) = R_x(x, y)^2 + G_x(x, y)^2 + B_x(x, y)^2$$

$$\Delta_y^2(x, y) = R_y(x, y)^2 + G_y(x, y)^2 + B_y(x, y)^2$$

I valori di $R_x(x, y)$, $G_x(x, y)$ e $B_x(x, y)$ sono le differenze assolutute tra le componenti R, G o B dei pixel adiacenti $(x - 1, y)$ e $(x + 1, y)$. Analogmante per $\Delta_y^2(x, y)$ i calcoli vengono effettuati con le componenti adiacenti rispetto la coordinata y, ovvero $(x, y - 1)$ e $(x, y + 1)$. Occorre solamente prestare attenzione quando si elaborano i punti che sono sul bordo dell'immagine evitando di considerare valori che sono fuori dai margini, in tal caso si considerano invece i valori delle componenti presenti sul texel in esame.

Il calcolo dell'energia con gradiente duale è un problema "*embarassingly parallel*" cioè può essere effettuato parallelamente ad ogni elemento della matrice. Poiché l'immagine da elaborare non necessariamente ha le dimensioni

di una texture quadrata con lato di lunghezza pari ad una potenza di 2, occorre consegnare allo shader due texure: una contenente l'immagine da elaborare e un'altra contenente valori che indicano se il texel è da elaborare, se è sul margine o se è fuori dall'immagine. La scelta adottata prevede che la seconda texture contenga la componente y a 1 se il texel corrente è da elaborare, 0 altrimenti. Se $y = 1$ allora si verifica anche la componente x che può possedere il valore 0 se indica un texel generico, oppure i seguenti valori come riferimento per l'immagine:

```
#define CORNER_TL 1
#define EDGE_TOP 2
#define CORNER_TR 4
#define EDGE_RIGHT 8
#define CORNER_BR 16
#define EDGE_BOTTOM 32
#define CORNER_BL 64
#define EDGE_LEFT 128
```

Per l'esecuzione del calcolo parallelo si fa uso dello shader, il cui codice semplificato è riportato di seguito. Per le funzioni ausiliarie implementate si faccia riferimento all'appendice.

```
varying highp vec2 vTexCoord;
uniform highp sampler2D uInputTex;
uniform highp sampler2D uInputTexCoords;
uniform highp vec2 coords;

void main() {
    highp vec4 near[4];
    highp vec2 T = gl_FragCoord.xy;

    if (computeTexel()) {
        //right edge
        if (right()) {
```

```
13              near[0] = texture2D(uInputTex, T*coords);
14          }
15          else {
16              near[0] = texture2D(uInputTex, vec2(T.x + 1.0,
                    T.y)*coords);
17          }
18
19          //bottom edge
20          if (bottom()) {
21              near[1] = texture2D(uInputTex, T*coords);
22          }
23          else {
24              near[1] = texture2D(uInputTex, vec2(T.x, T.y +
                    1.0)*coords);
25          }
26
27          //left edge
28          if (left()) {
29              near[2] = texture2D(uInputTex, T*coords);
30          }
31          else {
32              near[2] = texture2D(uInputTex, vec2(T.x - 1.0,
                    T.y)*coords);
33          }
34
35          //up edge
36          if (top()) {
37              near[3] = texture2D(uInputTex, T*coords);
38          }
39          else {
40              near[3] = texture2D(uInputTex, vec2(T.x, T.y -
                    1.0)*coords);
41          }
```

```
42
43          highp vec4 deltay = abs(near[0] - near[2]);
44          highp vec4 deltax = abs(near[1] - near[3]);
45
46          gl_FragColor = deltax*deltax + deltay*deltay;
47      }
48      else {
49          gl_FragColor = vec4(0,0,0,0);
50      }
51  }
```

Occorre ricordare che i valori dati in input nel range $(0 - 255)$ vengono convertiti automaticamente nel range $(0..1)$ in virgola mobile, occorre quindi eseguire delle operazioni di conversione per ottenere i valori di input.

Per eseguire gli shader occorre quindi definire nel codice dell'applicazione tutti i parametri per utilizzare correttamente la tecnica di calcolo parallelo. La semplificazione di questo aspetto viene descritto qui di seguito, per le funzioni ausiliarie si veda l'appendice.

```
1  glUseProgram( esContext.programObject[0] );
2
3  glClear(GL_COLOR_BUFFER_BIT);
4  glViewport(0, 0, texSize, texSize);
5
6  setCommonsTexture(GL_TEXTURE1, &inputTex1);
7  loadTextureData(GL_TEXTURE1, texSize, texSize, dataX);
8
9  setCommonsTexture(GL_TEXTURE3, &processTex1);
10 loadTextureData(GL_TEXTURE3, texSize, texSize, dataZ);
11
12 setOutputTexture(GL_TEXTURE0, &outputTex, texSize, texSize, &fb);
13
14 glFinish();
15
```

```
16 glUniform2f(glGetUniformLocation(esContext.programObject[0],
       "coords"), 1.0/texSize, 1.0/texSize);
17 glUniform1i(glGetUniformLocation(esContext.programObject[0],
       "uInputTex"), 1);
18 glUniform1i(glGetUniformLocation(esContext.programObject[0],
       "uInputTexCoords"), 3);
19
20 computeGPU(esContext.programObject[0]);
21
22 glReadPixels(0, 0, texSize, texSize, GL_RGB, GL_UNSIGNED_BYTE,
       dataY);
23
24 glDeleteTextures(1, &inputTex1);
25 glDeleteTextures(1, &processTex1);
26 glDeleteTextures(1, &outputTex);
27 glDeleteFramebuffers(1, &fb);
```

Possiamo individuare tre strutture dati principali in questo frammento di codice: `dataX` contenente i valori delle componenti di colore per ogni texel dell'immagine, `dataZ` contenente le indicazioni sui margini e sui texel da elaborare dell'immagine, `dataY` per raccogliere i valori di output dell'elaborazione.

Il risultato delle energie calcolate per ogni pixel è mostrato in figura 4.4.

Il passo successivo consiste nel calcolo delle cuciture per il seam carving. Il principale problema da affrontare è ancora una volta nella limitazione imposta dal formato dei dati. La funzione di ricorrenza per calcolare le cuciture è definita come segue, sempre prestando attenzione a non considerare valori fuori dai margini:

$$W[0, j] = E[0, j]$$

$$W[i, j] = E[i, j] + \min\{W[i-1, j-1], W[i-1, j], W[i-1, j+1]\}$$

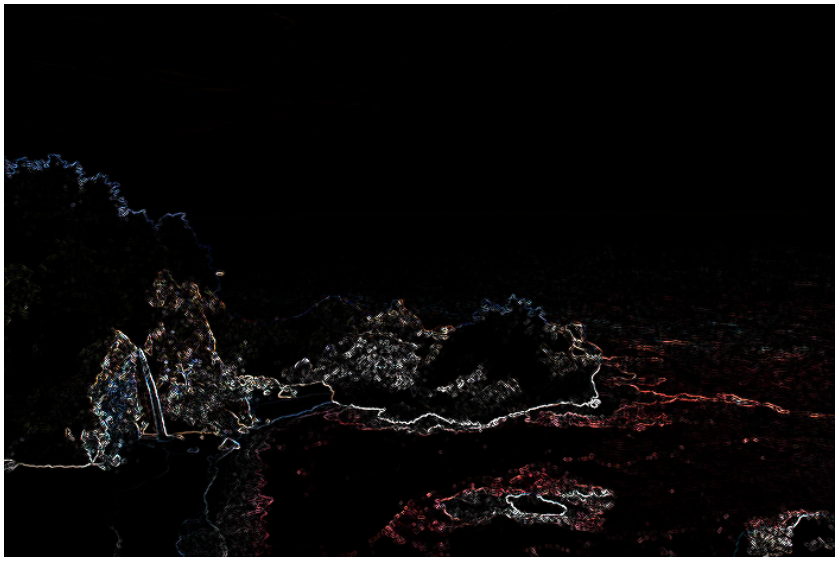


Figura 4.4: Calcolo dell'energia come gradiente duale.

dove E è la matrice con i valori delle energie per ogni texel, W è la matrice con i pesi delle cuciture.

Il calcolo effettuato nel passo precedente di elaborazione definisce il valore per ogni componente RBG tra 0 e 255, quindi è plausibile che la somma comporterebbe un overflow in pochi passi di elaborazione. L'idea è dunque quella di considerare non 3 valori da 8 bit, bensì 1 valore da $3 * 8 = 24$ bit per ciascun texel. Per questo approccio sono sufficienti le operazioni di moltiplicazione, divisione e resto, completamente fruibili sia sulla CPU che attraverso lo shader sulla GPU.

Un numero intero n compreso tra 0 e $2^{24} - 1$ può essere rappresentato in 3 componenti nel seguente modo:

$$v[2] = \lfloor n/65536 \rfloor$$

$$v[1] = \lfloor (n \bmod 65536)/256 \rfloor$$

$$v[0] = v[1] \bmod 256$$

Analogamente, per ottenere il valore di n date le tre componenti è sufficiente eseguire l'operazione di calcolo inversa:

$$n = v[0] + v[1] * 256 + v[2] * 65536$$

Sono quindi utilizzati 2 fragment shader per il calcolo delle cuciture: uno per la conversione del formato numerico della prima riga della matrice, l'altro per l'efftivo calcolo delle cuciture sulla matrice restante. Dato che la funzione di energia ha determinato 3 componenti di colore, ora il passo da svlogere è sommare le componenti in un unico valore e memorizzarlo come descritto.

```
varying highp vec2 vTexCoord;
uniform highp sampler2D uInputTex;
uniform highp sampler2D uInputTexCoords;
uniform highp vec2 coords;

void main() {
    if (computeTexel()) {
        gl_FragColor =
            getFragValue(getSimpleValue(vec3(texture2D(uInputTex,
            vTexCoord))));
    }
    else {
        gl_FragColor = vec4(0,0,0,0);
    }
}
```

Le operazioni da eseguire sul codice per eseguire lo shader sono semplici in quanto anche in questo caso si tratta di risolvere un problema "*embarrassing parallel*". La texture in questo caso può non essere quadrata in quanto non occorre accedere ai texel vicini, l'importante è che la dimensione dei lati continui ad essere una potenza di 2.

```
glUseProgram( esContext.programObject[1] );

glClear(GL_COLOR_BUFFER_BIT);
glViewport(0, 0, texSize, 1);
```

```
5
6  setCommonsTexture(GL_TEXTURE1, &inputTex1);
7  loadTextureData(GL_TEXTURE1, texSize, 1, dataY);
8
9  setCommonsTexture(GL_TEXTURE3, &processTex1);
10 loadTextureData(GL_TEXTURE3, texSize, 1, dataZ);
11
12 setOutputTexture(GL_TEXTURE0, &outputTex, texSize, 1, &fb);
13
14 glFinish();
15
16 glUniform2f(glGetUniformLocation(esContext.programObject[1],
       "coords"), 1.0/texSize, 1.0);
17 glUniform1i(glGetUniformLocation(esContext.programObject[1],
       "uInputTex"), 1);
18 glUniform1i(glGetUniformLocation(esContext.programObject[1],
       "uInputTexCoords"), 3);
19
20 computeGPU(esContext.programObject[1]);
21
22 glReadPixels(0, 0, texSize, 1, GL_RGB, GL_UNSIGNED_BYTE, dataY);
23
24 glDeleteTextures(1, &inputTex1);
25 glDeleteTextures(1, &processTex1);
26 glDeleteTextures(1, &outputTex);
27 glDeleteFramebuffers(1, &fb);
```

Dopo questa fase di conversione di dati occorre calcolare le cuciture. Per questo scopo è sufficiente richiamare il calcolo su GPU riga per riga, utilizzando ad ogni iterazione i valori ottenuti nell'iterazione precedente.

```
1 void main() {
2     highp vec3 sx, dx, cx, test;
3     highp int v1, v2, v3, vt, min;
```

```
4
5     if (computeTexel()) {
6         cx = vec3(texture2D(uInputTex, vTexCoord));
7
8         if (first()) {
9             sx = vec3(texture2D(uInputTex, vTexCoord));
10        }
11        else {
12            sx = vec3(texture2D(uInputTex, vec2(vTexCoord.x -
                  coords.x, vTexCoord.y)));
13        }
14
15        if (last()) {
16            dx = vec3(texture2D(uInputTex, vTexCoord));
17        }
18        else {
19            dx = vec3(texture2D(uInputTex, vec2(vTexCoord.x +
                  coords.x, vTexCoord.y)));
20        }
21
22        test = vec3(texture2D(uInputTex2, vTexCoord));
23        vt = getSimpleValue(test);
24
25        v1 = getValue(sx);
26        v2 = getValue(cx);
27        v3 = getValue(dx);
28
29        if (v1 < v2) {
30            if (v1 < v3) {
31                min = v1;
32            }
33            else {
34                min = v3;
```

```
35              }
36          }
37          else {
38              if (v2 < v3) {
39                  min = v2;
40              } else {
41                  min = v3;
42              }
43          }
44
45          gl_FragColor = getFragValue(min + vt);
46      }
47      else {
48          gl_FragColor = vec4(0,0,0,0);
49      }
50  }
```

Lo shader può dunque essere richiamato normalmente in un ciclo che elabori secondo l'altezza dell'immagine.

```
1  glUseProgram( esContext.programObject[2] );
2
3  glClear(GL_COLOR_BUFFER_BIT);
4  glViewport(0, 0, texSize, 1);
5
6  setCommonsTexture(GL_TEXTURE1, &inputTex1);
7  setCommonsTexture(GL_TEXTURE2, &inputTex2);
8
9  setCommonsTexture(GL_TEXTURE3, &processTex1);
10 loadTextureData(GL_TEXTURE3, texSize, 1, dataZ);
11
12 setOutputTexture(GL_TEXTURE0, &outputTex, texSize, 1, &fb);
13
14 int i;
```

```
15 for (i = 0; i < (height-1); i++) {
16     loadTextureData(GL_TEXTURE1, texSize, 1, &dataY[i * texSize *
           TEXEL_SIZE]);
17     loadTextureData(GL_TEXTURE2, texSize, 1, &dataY[(i+1) * texSize
           * TEXEL_SIZE]);
18
19     glFinish();
20
21     glUniform2f(glGetUniformLocation(esContext.programObject[2],
           "coords"), 1.0/texSize, 1.0);
22     glUniform1i(glGetUniformLocation(esContext.programObject[2],
           "uInputTex"), 1);
23     glUniform1i(glGetUniformLocation(esContext.programObject[2],
           "uInputTex2"), 2);
24     glUniform1i(glGetUniformLocation(esContext.programObject[2],
           "uInputTexCoords"), 3);
25
26     computeGPU(esContext.programObject[2]);
27
28     glReadPixels(0, 0, texSize, 1, GL_RGB, GL_UNSIGNED_BYTE,
           &dataY[(i+1) * texSize * TEXEL_SIZE]);
29 }
30
31 glDeleteTextures(1, &inputTex1);
32 glDeleteTextures(1, &inputTex2);
33 glDeleteTextures(1, &processTex1);
34 glDeleteTextures(1, &outputTex);
35 glDeleteFramebuffers(1, &fb);
```

Il risultato delle cuciture calcolate è mostrato in 4.5.

Infine occorre calcolare la cucitura minima. In questo caso il problema risulta non banale nelle limitazioni imposte da OpenGL ES 2; siamo interessati all'indice del valore minimo, non al minimo stesso. Procedere in parallelo

Figura 4.5: Calcolo delle possibili cuciture.

richiede di suddividere il vettore da esaminare in più texture quindi esaminare queste ricorsivamente fino ad ottenre un unico valore. In questo caso il problema è che gli indici vengono normalizzati sempre tra 0 ed 1 qualsiasi sia la dimensione della texture. In pratica il riferimento al dato puntato viene perduto a meno che non si suddivida la texture dei dati come la texture degli indici ad ogni passo di iterazione. In alternativa si può decidere di non dividere la texture e procedere con passi di elaborazione costanti che tuttavia richiedono di elaborare sempre una stessa quantità di dati, quindi con sprecando molta della potenza di calcolo in passi inutili.

La soluzione proposta prevede di effettuare i test di confronto su GPU ed ottenere una risposta booleana per ogni confronto, quindi vengono lette le risposte e si copiano i valori minori con i rispettivi indici, dimezzando ad ogni iterazione la dimensione dei dati da analizzare. Il vantaggio principale di questo approccio consiste nell'effettuare i calcoli di conversione tra i valori su GPU.

L'implementazione dello shader è la seguente:

```
1 void main() {
```

```
    highp int index_a, index_b;
    highp float real_index_a, real_index_b;
    highp int a, b;
    highp vec2 T = gl_FragCoord.xy;

    bool empty1 = voidTexel(uInputTexCoords);
    bool empty2 = voidTexel(uInputTexCoords2);

    if (empty1 && empty2) {
        gl_FragColor = vec4(0,0,0,0);
    }
    else if (!empty1 && empty2) {
        gl_FragColor = vec4(255,0,0,0);
    }
    else if (empty1 && !empty2) {
        gl_FragColor = vec4(0,255,0,0);
    }
    else {
        a = getValue(vec3(texture2D(uInputTex, vTexCoord)));
        b = getValue(vec3(texture2D(uInputTex2, vTexCoord)));

        if (a < b) {
            gl_FragColor = vec4(255,0,0,0);
        }
        else {
            gl_FragColor = vec4(0,255,0,0);
        }
    }
}
```

Mentre l'implementazione complementare su CPU è la seguente:

```
glUseProgram( esContext.programObject[3] );
glClear(GL_COLOR_BUFFER_BIT);
```

```
3  setCommonsTexture(GL_TEXTURE1, &inputTex1);
4  setCommonsTexture(GL_TEXTURE2, &inputTex2);
5  setCommonsTexture(GL_TEXTURE3, &processTex1);
6  setCommonsTexture(GL_TEXTURE4, &processTex2);
7  setOutputTexture(GL_TEXTURE0, &outputTex, texSize, 1, &fb);
8
9  for (i = 0; i < width; i++) {
10    div_t q = div(i, 65536);
11    binarySearchIndex[i*3 + 2] = q.quot;
12    q = div(q.rem, 256);
13    binarySearchIndex[i*3 + 1] = q.quot;
14    binarySearchIndex[i*3] = q.rem;
15 }
16
17 for (i = 0; i < width; i++) {
18    binarySearchValue[i*3] = dataY[OFFSET(i, height-1, texSize)];
19    binarySearchValue[i*3 + 1] = dataY[OFFSET(i, height-1, texSize) +
          1];
20    binarySearchValue[i*3 + 2] = dataY[OFFSET(i, height-1, texSize) +
          2];
21 }
22
23 for (q = texSize >> 1; q > 0; q >>= 1) {
24    glViewport(0, 0, q, 1);
25
26    loadTextureData(GL_TEXTURE1, q, 1, binarySearchValue);
27    loadTextureData(GL_TEXTURE2, q, 1, &binarySearchValue[q *
          TEXEL_SIZE]);
28    loadTextureData(GL_TEXTURE3, q, 1, binarySearchIndex);
29    loadTextureData(GL_TEXTURE4, q, 1, &binarySearchIndex[q *
          TEXEL_SIZE]);
30
31    glFinish();
```

```
32
33    glUniform2f(glGetUniformLocation(esContext.programObject[3],
          "coords"), 1.0/q, 1.0);
34    glUniform1i(glGetUniformLocation(esContext.programObject[3],
          "uInputTex"), 1);
35    glUniform1i(glGetUniformLocation(esContext.programObject[3],
          "uInputTex2"), 2);
36    glUniform1i(glGetUniformLocation(esContext.programObject[3],
          "uInputTexCoords"), 3);
37    glUniform1i(glGetUniformLocation(esContext.programObject[3],
          "uInputTexCoords2"), 4);
38
39    computeGPU(esContext.programObject[3]);
40
41    glReadPixels(0, 0, q, 1, GL_RGB, GL_UNSIGNED_BYTE,
          binarySearchResult);
42
43    for (i = 0; i < q; i++) {
44       if (binarySearchResult[i*TEXEL_SIZE + 1] == 255) {
45          binarySearchValue[i*TEXEL_SIZE] =
                binarySearchValue[i*TEXEL_SIZE + q*TEXEL_SIZE];
46          binarySearchValue[i*TEXEL_SIZE + 1] =
                binarySearchValue[i*TEXEL_SIZE + q*TEXEL_SIZE + 1];
47          binarySearchValue[i*TEXEL_SIZE + 2] =
                binarySearchValue[i*TEXEL_SIZE + q*TEXEL_SIZE + 2];
48
49          binarySearchIndex[i*TEXEL_SIZE] =
                binarySearchIndex[i*TEXEL_SIZE + q*TEXEL_SIZE];
50          binarySearchIndex[i*TEXEL_SIZE + 1] =
                binarySearchIndex[i*TEXEL_SIZE + q*TEXEL_SIZE + 1];
51          binarySearchIndex[i*TEXEL_SIZE + 2] =
                binarySearchIndex[i*TEXEL_SIZE + q*TEXEL_SIZE + 2];
52       }
```

```
53    }
54 }
55
56 int pivot = binarySearchIndex[0] + binarySearchIndex[1]* 256 +
       binarySearchIndex[2] * 65536;
```

Dopo aver individuato l'indice del valore minimo in *pivot* l'ultima parte per completare l'algoritmo di Seam Carving consiste nel rimuovere l'elemento dalla matrice che rappresenta l'immagine e risalire in altezza seguendo la cucitura composta per ogni elemento in (x, y) dal minimo valore tra $(x-1, y-1)$, $(x, y-1)$ o $(x+1, y-1)$. Questo aspetto è strettamente sequenziale e non può essere ottimizzato dal calcolo parallelo.

```
1  for (i=height-1; i>=0; i--) {
2     for (j = pivot; j < width-1; j++) {
3       dataX[OFFSET(j, i, texSize)] = dataX[OFFSET(j+1, i, texSize)];
4       dataX[OFFSET(j, i, texSize) + 1] = dataX[OFFSET(j+1, i,
            texSize) + 1];
5       dataX[OFFSET(j, i, texSize) + 2] = dataX[OFFSET(j+1, i,
            texSize) + 2];
6     }
7
8     if (i==0) break;
9
10    int next = pivot;
11
12    if (pivot > 0 && dataY[OFFSET(pivot-1, i-1, texSize)] <
          dataY[OFFSET(next, i-1, texSize)]) {
13      next = pivot - 1;
14    }
15
16    if (pivot < width-1 && dataY[OFFSET(pivot+1, i-1, texSize)] <
          dataY[OFFSET(next-1, i-1, texSize)]) {
17      next = pivot + 1;
```

```
18      }
19
20      pivot = next;
21  }
22
23  width--;
```

Al termine di queste operazioni è possibile ottenere l'immagine risultante in `dataY` ridotta di 1 pixel. Se l'operazione viene eseguita n volte allora n pixel sarebbero rimossi.

Figura 4.6: Immagine dopo 100 cuciture rimosse con Seam Carving.

Capitolo 5

Valutazione dei risultati ottenuti

In questo capitolo analizzeremo i risultati ottenuti dall'impiego della tecnica di programmazione parallela su GPU con impiego delle OpenGL ES 2. Le prove eseguite vogliono evidenziare le prestazioni temporali per varie tipologie di task, evidenziati per l'implentazione dell'algoritmo di Seam Carving. Si vuole inoltre valutare il rapporto tra le prestazioni dell'algoritmo parallelo e quello sequenziale. Inoltre si volgiono valutare gli impatti sulle prestazioni dati dai tempi di trasferimento dei dati tra la memoria RAM e la memoria della GPU.

5.1 Descrizione delle prove sperimentali

Al fine di poter valutare se l'impiego delle tecniche di programmazione parallela usando OpenGL siano effettivamente utili per un concreto impiego nello sviluppo di software per Raspberry Pi occorre raccogliere informazioni relative le reali prestazioni ottenute nell'impiego di queste. Ci si domanda pertanto se è vero che si riduca il tempo di calcolo utilizzando la GPU, e se si quali sono i casi in cui risulta più conveniente o altrimenti quali siano le limitazioni che inducano a non impiegare l'approccio descritto. Per poter rispondere obiettivamente a questi interrogativi è necessario condurre le opportune indagini, eseguendo prove sperimentali, raccogliendo dati ed ana-

lizzando i risultati ottenuti. Nel caso specifico si intende raccogliere i tempi impiegati nell'esecuzione dei singoli task che compongono l'algoritmo di Seam Carving parallelo e confrontarli con gli analoghi task che compongono l'algoritmo equenziale. Per la fase di testing si è fatto uso di un Rasperry Pi B e sono stati raccolti dati per 3 iterazioni di ciascuna prova. Si rivela inoltre necessario considerare che l'algoritmo parallelo richiede che i dati vengano consegnati, elaborati e poi recuperati dalla GPU, quindi è necessario valutare l'impatto che il trasferimento dei dati possiede relativamente le prestazioni complessive dell'algoritmo.

5.2 Prestazioni del Seam Carving parallelo

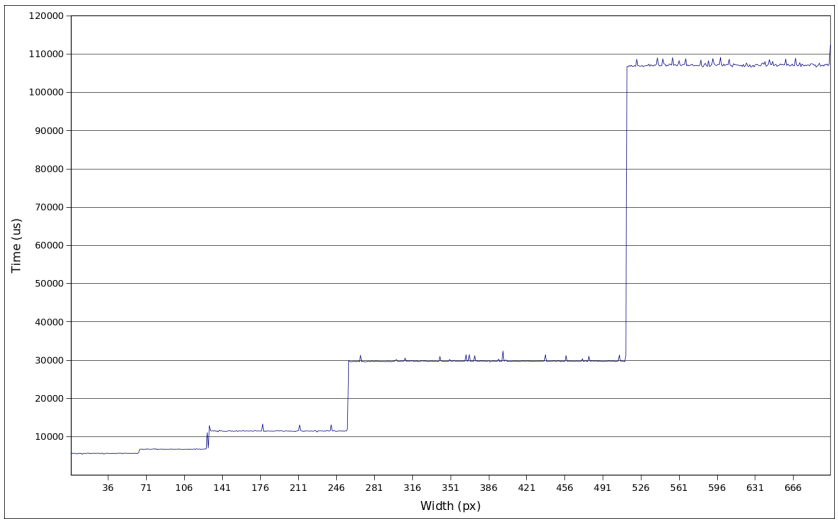


Figura 5.1: Tempo impiegato nel calcolo delle energie dei pixel.

Il grafico riportato in figura 5.1 raccoglie i tempi di esecuzione per il primo task dell'algoritmo parallelo, ovvero il calcolo delle energie per ogni pixel che compone l'immagine. Questa operazione è relativamente semplice in quanto richiede di eseguire operazioni di calcolo su tutti gli elementi, quindi *data parallel*. Sull'asse delle ascisse è riportato il numero di pixel che compongono la larghezza dell'immagine data in input, sull'asse delle ordinate invece

il tempo espresso in microsecondi. Come prima osservazione notiamo che l'impiego di texture POT (*Power Of Two*), ovvero texture quadrate con lato di dimensione pari ad una potenza di 2, influisce sulle prestazioni del task indipendentemente dalle dimensioni dell'immagine in essa contenuta. Questo significa che il tempo per elaborare su GPU in data parallel dipende dalla dimensione della texture contenente i dati indipendentemente da quanti dati sono contenuti in essa. Da questo si deduce che, se possibile, è opportuno utilizzare una texture le cui dimensioni siano le minime per contenere i dati da elaborare.

Tuttavia, qualora le dimensioni dei dati da elaborare non siano identiche a quelle della texture, una leggera perdita di prestazioni risulta inevitabile. Per esaminare meglio questo aspetto consideriamo i valori raccolti dalle simulazioni e confrontiamoli rispetto la larghezza dell'immagine. I risultati ottenuti possono essere evidenziati in figura 5.2.

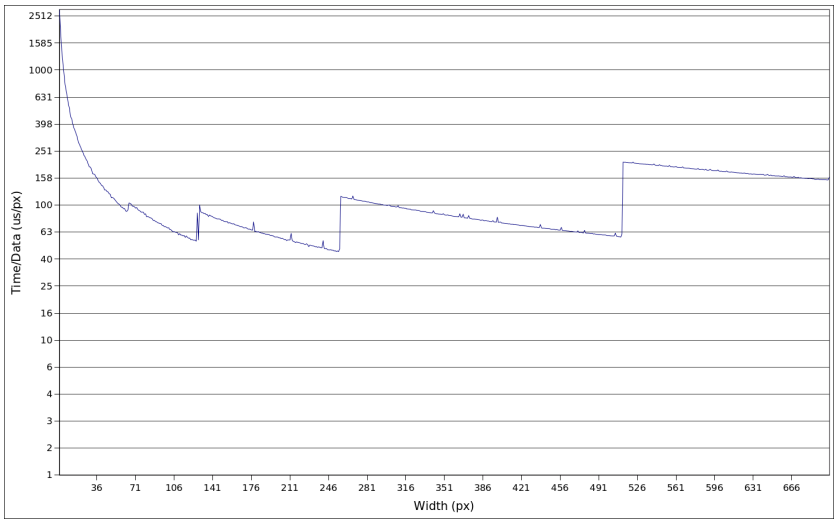


Figura 5.2: Rapporto larghezza immagine per tempo impiegato.

Possiamo notare che i valori di prestazione migliori sono quelli che impiegano meno tempo per quantità di dati da elaborare, ovvero non si produce uno spreco di tempo per esaminare texel "vuoti" i cui dati non sono rilevanti per il calcolo ma semplice spazio vuoto. Poiché i tempi di elaborazione del-

la texture sono pressoché costanti in base alla dimensione di questa, risulta chiaro che l'elaborazione di meno dati comporta una perdita nelle prestazioni. Quello che risulta tuttavia interessante è notare che per alcune texture, ad esempio quelle di lato 64, le prestazioni di tempo/pixel sono meno ottime rispetto quelle delle texture più grandi, ad esempio quelle di lato 255. Questo perdita di prestazione è plausibile poiché anche il trasferimento di dati verso e dalla GPU richiede di delegare le OpenGL al compito, il quale può occupare il sistema per del tempo ed introducendo un overhead non trascurabile per le prestazioni.

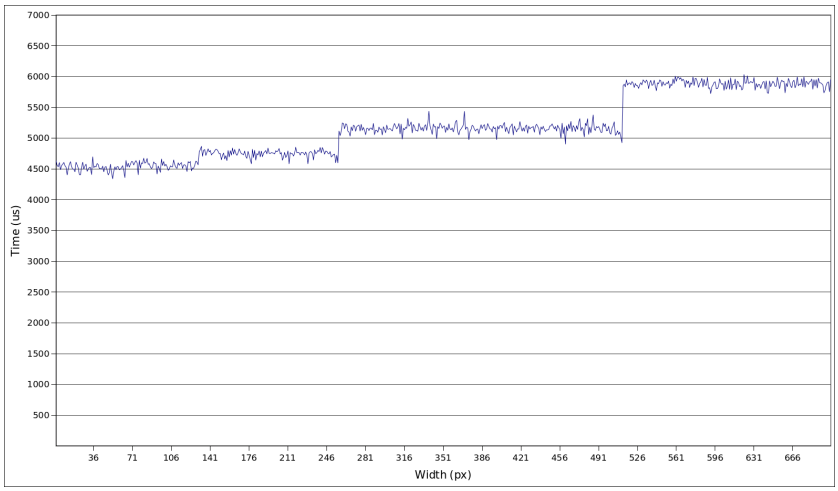


Figura 5.3: Conversione dei valori della prima riga di texel.

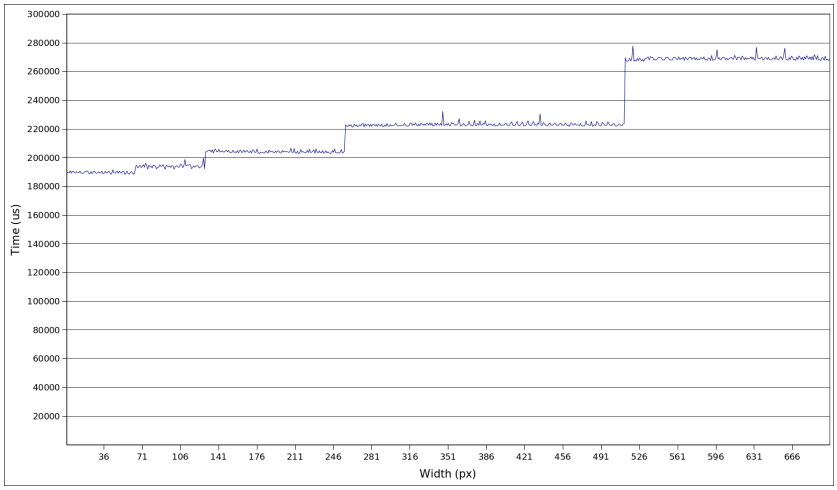


Figura 5.4: Calcolo delle possibili cuciture.

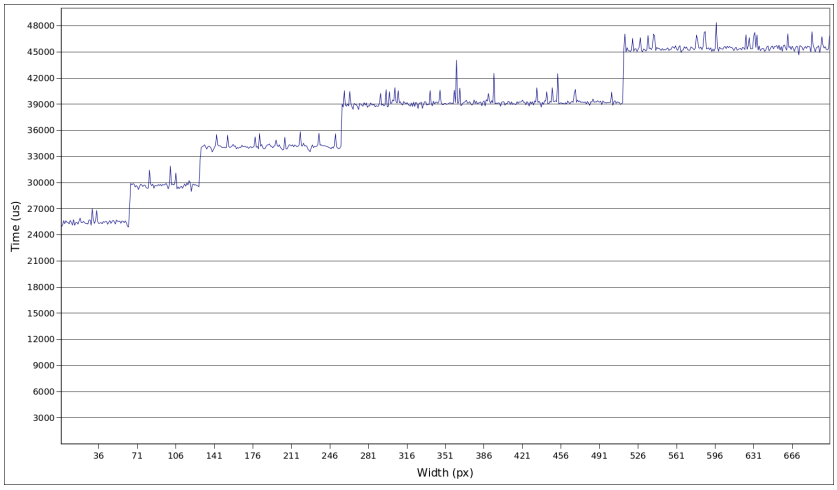


Figura 5.5: Calcolo della cucitura minima.

Per gli altri task eseguiti su GPU si ottengono comportamenti analoghi in quanto il tempo di elaborazione risulta dipendente dalla dimensione della texture, come mostrato dai grafici 5.3, 5.4 e 5.5. Per approfondire l'indagine in merito gli aspetti legati al confronto delle prestazioni tra calcolo parallelo su GPU e calcolo sequenziale su CPU si possono confrontare i tempi di esecuzione tra le due implementazioni dell'algoritmo.

5.3 Seam Carving parallelo e sequenziale

In questa analisi si confrontano i tempi ottenuti utilizzando una semplice variante dell'algoritmo, oltre a ridurre la larghezza dell'immagine di un pixel rimuovendo una cucitura si rimuove anche l'ultima riga di pixel dall'immagine. In questo modo ad ogni istanza dell'algoritmo si ha che la dimensione dell'immagine viene ridotta sia in larghezza che in altezza. Nei test si fornisce in input un'immagine quadrata di dimensioni 1024x1024 che consente di essere contenuta in una texture POT sempre più piccola all'aumentare delle cuciture rimosse.

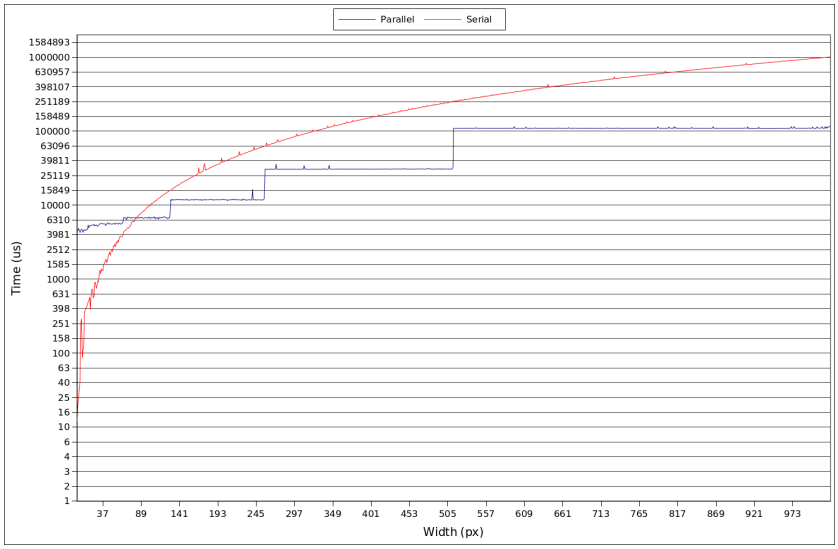


Figura 5.6: Confronto prestazioni calcolo energia.

Come possiamo notare dai risultati relativi le prestazioni del primo task in 5.6, per l'immagine di 1024x1024 pixel si ottiene un aumento delle prestazioni fino a quasi 10 volte migliori. Il miglioramento delle prestazioni è variabile in base all'dimensione dell'immagine da elaborare e diminuisce con il diminuire del numero di elementi da elaborare. Per l'immagine di 82x82 pixel le prestazioni della versione parallela e della versione sequenziale si equivalgono, e per immagini ancora più piccole la versione parallela fornisce prestazioni inferiori rispetto la versione sequenziale.

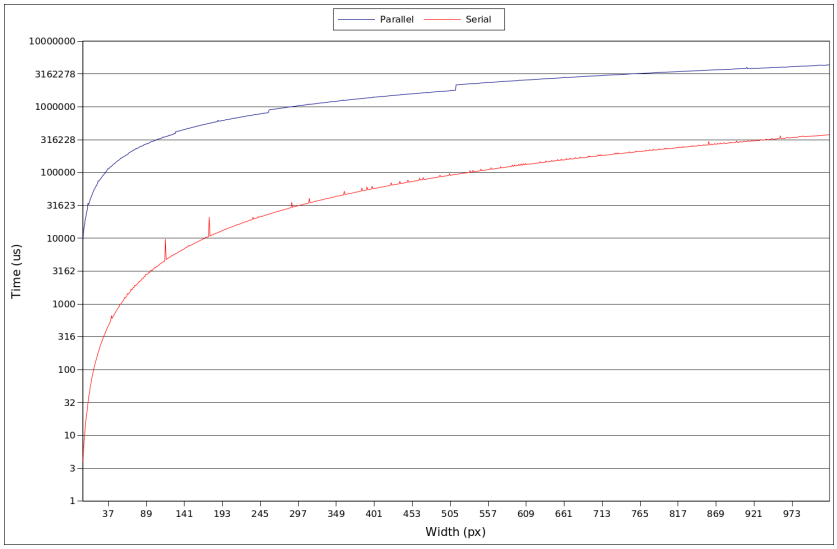


Figura 5.7: Confronto prestazioni calcolo delle cuciture.

Per il secondo task, come evidenziato in 5.7, il calcolo delle possibili cuciture, occorre notare una differenza nelle implementazioni tra la versione sequenziale e la versione parallela. La versione parallela presentata richiede di essere svolta in due passi: il primo che converte la prima riga di texel rappresentati da valori RGB nel formato numerico, come descritto in relazione alla problematica della limitazione sui formati dei valori possibili dei texel in OpenGL ES 2, mentre il secondo passo consiste nella ricerca delle somme minime tra i texel delle successive $n-1$ righe che compongono le cuciture. Questo task è suddiviso in due passaggi per consentire che gli shader eseguano le stesse operazioni su tutti i valori dati in input. Nella versione sequenziale questi passi possono essere riuniti in uno solo in quanto non abbiamo il problema relativo il formato dei valori da memorizzare utilizzando matrici di interi invece delle texture. Tuttavia, poiché l'operazione non può essere eseguita in data parallel su tutti i texel, occorre consegnare alla GPU riga per riga i valori dei texel che compongono l'immagine.

Come possiamo notare dai risultati ottenuti in questo caso è presente una significativa riduzione delle prestazioni rispetto il caso precedente. L'algoritmo sequenziale risulta migliore rispetto l'implementazione parallela. Ana-

logamente anche la ricerca del minimo risulta molto più efficiente nel caso sequenziale rispetto che in quello parallelo, come evidenziato in 5.8.

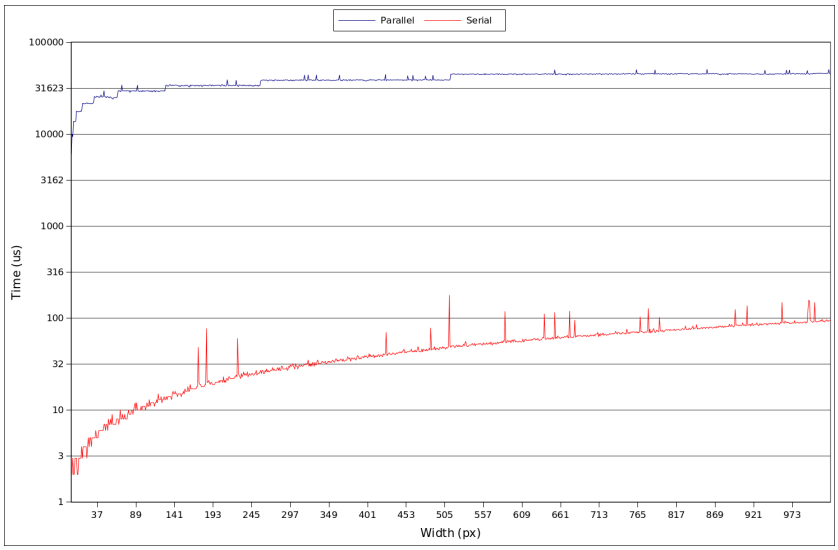


Figura 5.8: Confronto prestazioni ricerca minimo.

Da tutte queste osservazioni risulta opportuno individuare anche quali siano i tempi necessari per il trasferimento di una texture su GPU, la sua elaborazione e la sua rilettura. Per valutare questo comportamento si utilizza sempre la versione precedentemente descritta dell'algoritmo di Seam Carving e si raccolgono i tempi presentati. Per un'analisi completa si valutano sia i tempi richiesti rispettivamente sia da un'intera texture quadrata che da una riga di elementi che compongono la texture.

5.4 Prestazioni sul trasferimento dati

I grafici riportati in 5.9 e 5.10 mostrano rispettivamente i tempi raccolti per il trasferimento dei dati alla GPU, la loro elaborazione e la loro lettura. Il primo grafico fa riferimento al trasferimento di una matrice quadrata, il secondo invece fa riferimento ad una sola riga di texel. I tempi di elaborazione presi in considerazione sono relativi al calcolo delle energie dei texel per il

primo grafico ed alla conversione dei valori dei texel per il secondo grafico. Da questi risultati possiamo notare che la scrittura verso la GPU risulta più veloce rispetto che la lettura dei risultati, mentre il calcolo richiede più tempo rispetto al trasferimento dei dati.

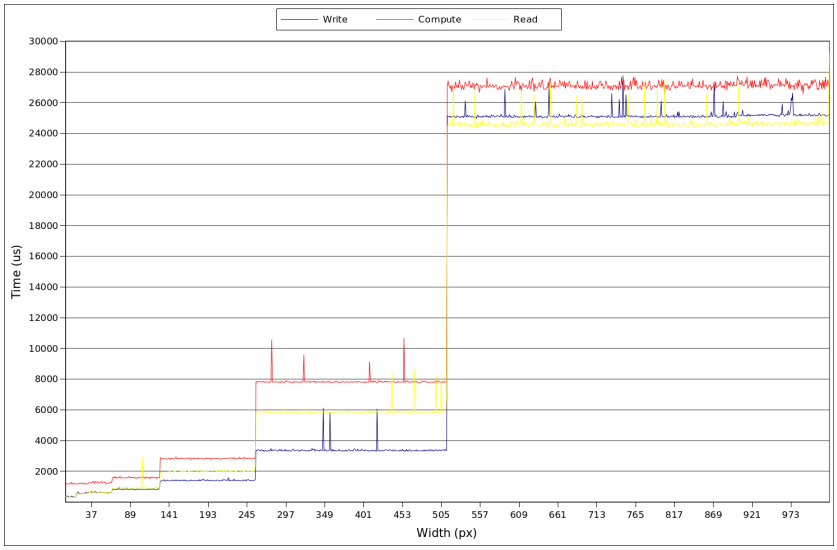


Figura 5.9: Trasferimento texture.

Figura 5.10: Trasferimento vettore di texel.

Da questi risultati si può notare che il trasferimento dei dati richiede tempo proporzioanle alla quantià di dati da trasferire. Tuttavia eseguire molte operazioni di trasferimento dati produce un overhead significativo sulle prestazioni, vanifincando quindi il miglioramento dei tempi di prestazione eventualmente ottenuti dal calcolo parallelo su GPU. Pertanto è dunque preferibile che i calcoli eseguiti su GPU siano relativi a grandi insiemi di dati, i cui calcoli risultino indipendenti tra loro e che non richiedano di essere trasferiti più volte tra la memoria centrale e la memoria della GPU.

Conclusioni

In questo lavoro abbiamo mostrato come la GPU del Rasperry Pi possa essere impiegata al fine di eseguire calcoli in parallelo per scopi differenti dalla *Computer Graphics*. Sebbene l'implementazione di algoritmi paralleli utilizzando OpenGL ES risulti essere particolarmente complicata, il miglioramento delle prestazioni è evidente per alcune tipologie di problemi. Il vantaggio principale che si può ottenere con l'impiego di questa tecnica è riscontrabile nei tempi di esecuzione qualora si necessiti di elaborare in modalità *data parallel* grandi quantità di dati. Stando ai risultati ottenuti, come evidenziato in 5.6, il miglioramento delle prestazioni è infatti riscontrato elaborando in data parallel matrici di dimensioni pari o superiori ad 82x82, ovvero elaborando almeno 6724 valori. Questo limite inferiore deve essere valutato anche in base alle tipologia di calcoli da eseguire, ovvero risulta che la quantità di operazioni matematiche da eseguire per singolo valore può influire sulla soglia di miglioramento, preferendo quindi le operazioni dette "*data intensive*". La principale perdita di prestazioni riscontrate nell'impiego di questa tecnica è dovuta alla necessità di trasferire i dati tra la memoria interna e quella della GPU, operazione che richiede tempo non trascurabile. Infatti, poiché il miglioramento è stato riscontrato al momento di elaborare almeno 6724 valori, risulta plausibile che negli altri casi di test non siano state raccolte prestazioni migliori. Nel Seam Carving, mentre il calcolo delle energie per ogni pixel poteva essere eseguito parallelamente, l'algoritmo per il calcolo delle cuciture richiedeva di procedere nell'elaborazione della matrice riga per riga. Poiché la larghezza massima dell'immagine fornita in input è pari a

1024 pixel, la quantità dei dati da elaborare parallelamente risulta inferiore alla soglia minima stimata.

Questi importanti risultati ci consentono di avere una visione delle prestazioni degli algoritmi non solo in relazione alla loro complessità, ma anche in relazione alla quantità di dati da elaborare. Per ottenere un reale 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 data parallel. Pertanto, al fine di ottenere un reale beneficio nelle prestazioni, la scelta migliore risulta essere un compromesso tra le tipologie di calcolo, mantenendo su GPU i calcoli data parallel per oltre una soglia minima di valori e su CPU i calcoli sequenziali o su piccole quantità di dati.

Come è stato inoltre evidenziato al momento dell'implementazione dell'algoritmo parallelo, le limitazioni indotte dall'impiego di OpenGL ES 2 sono principalmente date dalla tipologia di dati memorizzabili nelle texture. Con la recente versione di Raspbian ed i recenti driver liberi sperimentali, il Rasperry Pi è finalmente in grado di supportare le librerie OpenGL 2 tipiche dei computer desktop. Queste librerie possiedono meno restrizioni sulle tipologie di impiego dei driver e supportano maggiori formati di dati per le texture, come i float ed analoghi. Senza le limitazioni della versione embedded l'implementazione degli algoritmi risulta essere certamente semplificata, inoltre anche le possibilità ed i contesti d'uso possono essere maggiori in relazione al formato di dati ammissibili.

Appendice A

Context OpenGL ES 2

Si ringrazia Ben O'Steen per le parti di codice rilasciato pubblicamente e sul quale questo lavoro si è basato per lo sviluppo iniziale. Per riferimento si veda il suo blog [47].

```
1  /// esCreateWindow flag - RGB color buffer
2  #define ES_WINDOW_RGB        0
3  /// esCreateWindow flag - ALPHA color buffer
4  #define ES_WINDOW_ALPHA      1
5  /// esCreateWindow flag - depth buffer
6  #define ES_WINDOW_DEPTH      2
7  /// esCreateWindow flag - stencil buffer
8  #define ES_WINDOW_STENCIL    4
9  /// esCreateWindow flat - multi-sample buffer
10 #define ES_WINDOW_MULTISAMPLE 8
11
12 typedef struct _escontext {
13   GLuint* programObject;
14   int programCount;
15
16   GLint      width;
17   GLint      height;
18
```

```
19    EGLNativeWindowType hWnd;
20    EGLDisplay eglDisplay;
21    EGLContext eglContext;
22    EGLSurface eglSurface;
23  }
24  ESContext;
25
26  //  esInitContext()
27  //  Initialize ES utility context. This must be called before
        calling any other functions.
28  void ESUTIL_API esInitContext( ESContext *esContext, int count )
29  {
30    bcm_host_init();
31
32    if ( esContext != NULL ) {
33      memset( esContext, 0, sizeof( ESContext) );
34      esContext->programObject = calloc(count, sizeof(GLuint));
35      esContext->programCount = 0;
36    }
37  }
38
39  // esLogMessage()
40  // Log an error message to the debug output for the platform
41  void ESUTIL_API esLogMessage( const char *formatStr, ... )
42  {
43    va_list params;
44    char buf[BUFSIZ];
45    va_start ( params, formatStr );
46    vsprintf ( buf, formatStr, params );
47    printf ( "%s", buf );
48    va_end ( params );
49  }
50
```

```
51 // CreateEGLContext()
52 // Creates an EGL rendering context and all associated elements
53 EGLBoolean CreateEGLContext ( EGLNativeWindowType hWnd, EGLDisplay*
       eglDisplay, EGLContext* eglContext, EGLSurface* eglSurface,
       EGLint attribList[])
54 {
55   EGLint numConfigs;
56   EGLint majorVersion;
57   EGLint minorVersion;
58   EGLDisplay display;
59   EGLContext context;
60   EGLSurface surface;
61   EGLConfig config;
62   EGLint contextAttribs[] = { EGL_CONTEXT_CLIENT_VERSION, 2,
         EGL_NONE };
63
64   // Get Display
65   display = eglGetDisplay(EGL_DEFAULT_DISPLAY);
66   if ( display == EGL_NO_DISPLAY ) {
67     return EGL_FALSE;
68   }
69
70   // Initialize EGL
71   if ( !eglInitialize(display, &majorVersion, &minorVersion) ) {
72     return EGL_FALSE;
73   }
74
75   // Get configs
76   if ( !eglGetConfigs(display, NULL, 0, &numConfigs) ) {
77     return EGL_FALSE;
78   }
79
80   // Choose config
```

```
81    if ( !eglChooseConfig(display, attribList, &config, 1,
          &numConfigs) ) {
82      return EGL_FALSE;
83    }
84
85    // Create a surface
86    surface = eglCreateWindowSurface(display, config,
          (EGLNativeWindowType)hWnd, NULL);
87    if ( surface == EGL_NO_SURFACE ) {
88      return EGL_FALSE;
89    }
90
91    // Create a GL context
92    context = eglCreateContext(display, config, EGL_NO_CONTEXT,
          contextAttribs );
93    if ( context == EGL_NO_CONTEXT ) {
94      return EGL_FALSE;
95    }
96
97    // Make the context current
98    if ( !eglMakeCurrent(display, surface, surface, context) ) {
99      return EGL_FALSE;
100   }
101
102   *eglDisplay = display;
103   *eglSurface = surface;
104   *eglContext = context;
105
106   return EGL_TRUE;
107 }
108
109 //  WinCreate() - RaspberryPi, direct surface (No X, Xlib)
110 //  This function initialized the display and window for EGL
```

```
111 EGLBoolean WinCreate(ESContext *esContext, const char *title)
112 {
113   int32_t success = 0;
114
115   static EGL_DISPMANX_WINDOW_T nativewindow;
116
117   DISPMANX_ELEMENT_HANDLE_T dispman_element;
118   DISPMANX_DISPLAY_HANDLE_T dispman_display;
119   DISPMANX_UPDATE_HANDLE_T dispman_update;
120   VC_RECT_T dst_rect;
121   VC_RECT_T src_rect;
122
123   uint32_t display_width;
124   uint32_t display_height;
125
126   success = graphics_get_display_size(0, &display_width,
        &display_height);
127   if ( success < 0 ) {
128     return EGL_FALSE;
129   }
130
131   display_width = 1;
132   display_height = 1;
133
134   dst_rect.x = 0;
135   dst_rect.y = 0;
136   dst_rect.width = display_width;
137   dst_rect.height = display_height;
138
139   src_rect.x = 0;
140   src_rect.y = 0;
141   src_rect.width = display_width << 16;
142   src_rect.height = display_height << 16;
```

```
143
144   dispman_display = vc_dispmanx_display_open( 0 );
145   dispman_update = vc_dispmanx_update_start( 0 );
146   dispman_element = vc_dispmanx_element_add( dispman_update,
          dispman_display, 0, &dst_rect, 0, &src_rect,
          DISPMANX_PROTECTION_NONE, 0, 0, 0);
147
148   nativewindow.element = dispman_element;
149   nativewindow.width = display_width;
150   nativewindow.height = display_height;
151
152   vc_dispmanx_update_submit_sync( dispman_update );
153
154   esContext->hWnd = &nativewindow;
155
156   return EGL_TRUE;
157 }
158
159 //  esCreateWindow()
160 //  title - name for title bar of window
161 //  width - width of window to create
162 //  height - height of window to create
163 //  flags  - bitwise or of window creation flags
164 //  ES_WINDOW_ALPHA     - specifies that the framebuffer should have
        alpha
165 //  ES_WINDOW_DEPTH     - specifies that a depth buffer should be
        created
166 //  ES_WINDOW_STENCIL   - specifies that a stencil buffer should be
        created
167 //  ES_WINDOW_MULTISAMPLE - specifies that a multi-sample buffer
        should be created
168 GLboolean ESUTIL_API esCreateWindow ( ESContext *esContext, const
        char* title, GLint width, GLint height, GLuint flags )
```

```
169 {
170   EGLint attribList[] = {
171      EGL_RED_SIZE,     5,
172      EGL_GREEN_SIZE,   6,
173      EGL_BLUE_SIZE,    5,
174      EGL_ALPHA_SIZE,   (flags & ES_WINDOW_ALPHA) ? 8 : EGL_DONT_CARE,
175      EGL_DEPTH_SIZE,   (flags & ES_WINDOW_DEPTH) ? 8 : EGL_DONT_CARE,
176      EGL_STENCIL_SIZE, (flags & ES_WINDOW_STENCIL) ? 8 :
           EGL_DONT_CARE,
177      EGL_SAMPLE_BUFFERS, (flags & ES_WINDOW_MULTISAMPLE) ? 1 : 0,
178      EGL_NONE
179   };
180
181   if ( esContext == NULL ) {
182     return GL_FALSE;
183   }
184
185   esContext->width = width;
186   esContext->height = height;
187
188   if ( !WinCreate ( esContext, title) ) {
189     return GL_FALSE;
190   }
191
192   if ( !CreateEGLContext( esContext->hWnd, &esContext->eglDisplay,
         &esContext->eglContext, &esContext->eglSurface, attribList) ) {
193     return GL_FALSE;
194   }
195
196   return GL_TRUE;
197 }
198
199 // Create a shader object, load the shader source, and compile the
```

```
    shader.
200 GLuint LoadShader ( GLenum type, const char *shaderSrc )
201 {
202   GLuint shader;
203   GLint compiled;
204
205   // Create the shader object
206   shader = glCreateShader ( type );
207
208   if ( shader == 0 ) {
209     return 0;
210   }
211
212   // Load the shader source
213   glShaderSource ( shader, 1, &shaderSrc, NULL );
214
215   // Compile the shader
216   glCompileShader ( shader );
217
218   // Check the compile status
219   glGetShaderiv ( shader, GL_COMPILE_STATUS, &compiled );
220
221   if ( !compiled ) {
222     GLint infoLen = 0;
223
224     glGetShaderiv ( shader, GL_INFO_LOG_LENGTH, &infoLen );
225
226     if ( infoLen > 1 ) {
227       char* infoLog = malloc (sizeof(char) * infoLen );
228
229       glGetShaderInfoLog ( shader, infoLen, NULL, infoLog );
230       esLogMessage ( "Error compiling shader:\n%s\n", infoLog );
231
```

```
232       free ( infoLog );
233     }
234
235     glDeleteShader ( shader );
236
237     return 0;
238   }
239
240   return shader;
241 }
242
243 GLuint initShader(GLenum type, char* shaderFile) {
244   return LoadShader(type, readShaderSource(shaderFile));
245 }
246
247 // Initialize the shader and program object
248 int Init ( ESContext *esContext, GLuint vertexShader, GLuint
      fragmentShader )
249 {
250   GLuint programObject;
251   GLint linked;
252
253   // Create the program object
254   programObject = glCreateProgram ( );
255
256   if ( programObject == 0 ) {
257     return 0;
258   }
259
260   glAttachShader ( programObject, vertexShader );
261   glAttachShader ( programObject, fragmentShader );
262
263   // Bind vPosition to attribute 0
```

```
264   glBindAttribLocation ( programObject, 0, "vPosition" );
265
266   // Link the program
267   glLinkProgram ( programObject );
268
269   // Check the link status
270   glGetProgramiv ( programObject, GL_LINK_STATUS, &linked );
271
272   if ( !linked ) {
273     GLint infoLen = 0;
274
275     glGetProgramiv ( programObject, GL_INFO_LOG_LENGTH, &infoLen );
276
277     if ( infoLen > 1 ) {
278       char* infoLog = malloc (sizeof(char) * infoLen );
279
280       glGetProgramInfoLog ( programObject, infoLen, NULL, infoLog );
281       esLogMessage ( "Error linking program:\n%s\n", infoLog );
282
283       free ( infoLog );
284     }
285
286     glDeleteProgram ( programObject );
287
288     return GL_FALSE;
289   }
290
291   // Store the program object
292   esContext->programObject[esContext->programCount] = programObject;
293   esContext->programCount++;
294
295   glClearColor ( 0.0f, 0.0f, 0.0f, 1.0f );
296
```

```
297   return GL_TRUE;
298 }
```

Appendice B

GPGPU per OpenGL ES 2

```
1  void setCommonsTexture(GLenum texture, GLuint* id) {
2    glActiveTexture(texture);
3    glGenTextures(1, id);
4    glBindTexture(GL_TEXTURE_2D, *id);
5
6    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
7    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
8    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
         GL_CLAMP_TO_EDGE);
9    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
         GL_CLAMP_TO_EDGE);
10 }
11
12 void loadTextureData(GLenum texture, int w, int h, unsigned char*
       ptr) {
13   glActiveTexture(texture);
14   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, w, h, 0, GL_RGB,
         GL_UNSIGNED_BYTE, ptr);
15 }
16
```

```
17 void setOutputTexture(GLenum texture, GLuint* id_texture, int w,
       int h, GLuint* fb) {
18   setCommonsTexture(texture, id_texture);
19
20   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, w, h, 0, GL_RGB,
         GL_UNSIGNED_BYTE, NULL);
21
22   glGenFramebuffers(1, fb);
23   glBindFramebuffer(GL_FRAMEBUFFER, *fb);
24
25   glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
         GL_TEXTURE_2D, *id_texture, 0);
26 }
27
28 void computeGPU(GLuint programObject) {
29   const GLfloat vertexBuf[4*2] = {
30     -1, -1,
31      1, -1,
32     -1,  1,
33      1,  1 }; // vertex data buffer for a quad
34
35   const GLfloat texCoordBuf[4*2] = {
36     0, 0,
37     1, 0,
38     0, 1,
39     1, 1 }; // texture coordinate data buffer for a quad
40
41   GLint shParamAPos = glGetAttribLocation(programObject, "aPos");
42   GLint shParamATexCoord = glGetAttribLocation(programObject,
         "aTexCoord");
43
44   glEnableVertexAttribArray(shParamAPos);
```

```
45   glVertexAttribPointer(shParamAPos, 2, GL_FLOAT, GL_FALSE, 0,
         vertexBuf);
46
47   glVertexAttribPointer(shParamATexCoord, 2, GL_FLOAT, GL_FALSE, 0,
         texCoordBuf);
48   glEnableVertexAttribArray(shParamATexCoord);
49
50   glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
51 }
```

Appendice C

Funzioni ausiliare GLSL

```
1  bool computeTexel() {
2          highp vec4 position = texture2D(uInputTexCoords,
               gl_FragCoord.xy*coords);
3          return (position.y == 1.0/255.0);
4  }
5
6  bool voidTexel(sampler2D texture) {
7      highp vec4 position = texture2D(texture, vTexCoord);
8      return (position.x == 0.0 && position.y == 0.0 && position.z ==
           0.0);
9  }
10
11 bool right() {
12         highp vec4 position = texture2D(uInputTexCoords,
               gl_FragCoord.xy*coords);
13         return (position.x == 4.0/255.0 || position.x == 8.0/255.0
               || position.x == 16.0/255.0);
14 }
15
16 bool left() {
```

```
17          highp vec4 position = texture2D(uInputTexCoords,
                gl_FragCoord.xy*coords);
18          return (position.x == 1.0/255.0 || position.x == 64.0/255.0
                || position.x == 128.0/255.0);
19  }
20
21  bool top() {
22          highp vec4 position = texture2D(uInputTexCoords,
                gl_FragCoord.xy*coords);
23          return (position.x == 1.0/255.0 || position.x == 2.0/255.0
                || position.x == 4.0/255.0);
24  }
25
26  bool bottom() {
27          highp vec4 position = texture2D(uInputTexCoords,
                gl_FragCoord.xy*coords);
28          return (position.x == 16.0/255.0 || position.x == 32.0/255.0
                || position.x == 64.0/255.0);
29  }
30
31  bool first() {
32          highp vec4 position = texture2D(uInputTexCoords, vTexCoord);
33          return (position.x == 1.0/255.0);
34  }
35
36  bool last() {
37          highp vec4 position = texture2D(uInputTexCoords, vTexCoord);
38          return (position.x == 4.0/255.0 );
39  }
40
41  int getSimpleValue(vec3 v) {
42          return int(v.x*255.0 + v.y*255.0 + v.z*255.0);
43  }
```

```
44
45 int getValue(vec3 v) {
46         return int(v.x*255.0 + v.y*255.0*256.0 + v.z*255.0*65536.0);
47 }
48
49 vec4 getFragValue(int n) {
50         highp vec4 v = vec4(0.0, 0.0, 0.0, 0.0);
51         highp float q = float(n);
52
53         if (q >= 65536.0) {
54                 v.z = q / (65536.0*255.0);
55                 q = q - 65536.0 * floor(q / 65536.0);
56         }
57
58         if (q >= 256.0) {
59                 v.y = q / (256.0*255.0);
60                 q = q - 256.0 * floor(q / 256.0);
61         }
62
63         v.x = float(q / 255.0);
64
65         return v;
66 }
```

Bibliografia

[1] Eben Upton (28th Feb 2014), "*A birthday present from Broadcom*", Official Blog Raspberry Pi. https://www.raspberrypi.org/blog/a-birthday-present-from-broadcom

[2] Eben Upton (2nd Jan 2015), "*New QPU macro assembler*", Official Blog Raspberry Pi. https://www.raspberrypi.org/blog/new-qpu-macro-assembler

[3] Eben Upton (30th Jan 2014), "*Accelerating Fourier transforms using the GPU*", Official Blog Raspberry Pi. https://www.raspberrypi.org/blog/accelerating-fourier-transforms-using-the-gpu

[4] Eben Upton (8th Aug 2014), "*More QPU magic from Pete Warden*", Official Blog Raspberry Pi. https://www.raspberrypi.org/blog/more-qpu-magic-from-pete-warden

[5] Herman H Hermitage (May 2014), "*Hacking the GPU for fun and profit*", Raspberry Pi Playground. https://rpiplayground.wordpress.com

[6] Marcel Müller (Dec 2014), "*Raspberry Pi BCM2835 QPU macro assembler*", VC4ASM - macro assembler for Broadcom VideoCore IV aka Raspberry Pi GPU. http://maazl.de/project/vc4asm/doc/index.html

[7] Xerxes Rånby (10th July 2015), "*Processing 3 is running for the first time on a Raspberry Pi using Eric Anholt's Mesa3D VC4 driver*", Zafena Development. http://labb.zafena.se/?category_name=jogamp

[8] Eric Anholt (14th Dec 2015), "*VC4 Status Update*", Anholt's Live Journal. http://anholt.livejournal.com

[9] Eric Anholt, "*VC4 Driver*", accessed on 23th Feb 2016. http://dri.freedesktop.org/wiki/VC4

[10] Koichi Nakamura, "*PyVideoCore*", accessed on 23th Feb 2016. https://github.com/nineties/py-videocore

[11] Dominik Göddeke (2005, 2006), "*GPGPU::Basic Math / FBO Tutorial*", accessed on 23th Feb 2016. http://www.seas.upenn.edu/ cis565/fbo.htm

[12] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, Timothy J. Purcell (March 2007), "*A Survey of General-Purpose Computation on Graphics Hardware*", Computer Graphics Forum, Volume 26, number 1 pp. 80–113, DOI: 10.1111/j.1467-8659.2007.01012.x

[13] BLAST Forum (21th Aug 2001), "*Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard*", Capitolo 3.

[14] Buck I. (March 2005), "*Taking the plunge into GPU computing*", In GPU Gems 2, Addison Wesley, chapter 32, pp. 509–519.

[15] Horn D. (March 2005), "*Stream reduction operations for GPGPU applications*", In GPU Gems 2, Addison Wesley, chapter 36, pp. 573–589.

[16] Batcher K. E. (April 1968), "*Sorting networks and their applications*". In Proceedings of the AFIPS Spring Joint Computing Conference, vol. 32, pp. 307–314.

[17] Shai Avidan, Ariel Shamir (2007), "*Seam Carving for Content-Aware Image Resizing*", In Proceeding of the ACM SIGGRAPH 2007, DOI: 10.1145/1275808.1276390.

[18] Peter Pacheco (2011), "*An introduction to parallel programming*", Elsevier.

[19] Anath Grama, George Karypis, Vipin Kumar, Anshul Gupta (2003), "*Introduction to parallel computing*, 2nd ed., Pearson Education.

[20] OpenACC, http://www.openacc.org

[21] OpenGL, https://www.khronos.org/opengl

[22] Dominik Göddeke (2005, 2007), "*GPGPU::Basic Math Tutorial*", accessed on 23th Feb 2016. http://www.mathematik.tu-dortmund.de/ goeddeke/gpgpu/tutorial.html

[23] CUDA, http://www.nvidia.it/object/cuda-parallel-computing-it.html

[24] OpenMP, http://openmp.org

[25] OpenCL, https://www.khronos.org/opencl

[26] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, Pat Hanrahan (2004), "*Brook for GPUs: Stream Computing on Graphics Hardware*", In Proceedings of ACM SIGGRAPH 2004, pp. 777-786, DOI: 10.1145/1186562.1015800.

[27] OpenMPI, https://www.open-mpi.org

[28] The Pi Hut, "*The Raspberry Pi*", accessed on 23th Feb 2016. http://thepihut.com/pages/the-raspberry-pi

[29] Parallel and Distributed Simulation Research Group, "*Raspèin Project*", University of Bologna, accessed on 23th Feb 2016. http://pads.cs.unibo.it/doku.php?id=raspein:index

[30] Gordon E. Moore (1998), "*Cramming More Components onto Integrated Circuits*", In Proceedings of the IEEE. Vol. 86. p. 4.

[31] Michael J. Flynn (1972), "*Some Computer Organizations and Their Eectiveness*", In IEEE Transactions on Computers 21.

[32] Gene Amdahl, "*Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*", In proceeding of AFIPS Joint Computer Conference, pp.483-485.

[33] Raspbery Pi, https://www.raspberrypi.org

[34] Raspberry Pi Forum, "*GPU Processing API*", accessed on 23th Feb 2016. https://www.raspberrypi.org/forums/viewtopic.php?f=33&t=6188

[35] Reddit, "*Is there an OpenCL implementation for the Pi being worked on?*", accessed on 23th Feb 2016. https://redd.it/3fi2of

[36] Elias Konstantinidis (28th Oct 2015), "*OpenCL on the Raspberry PI 2*", Parallel++. http://parallelplusplus.blogspot.it/2015/10/opencl-on-raspberry-pi-2.html

[37] Michael Larabel (9th Feb 2016), "*Raspbian Now Ships With Experimental Support For The New VC4 OpenGL Driver*", Phoronix. http://www.phoronix.com/scan.php?page=news_item&px=Raspbian-Feb2016-Update

[38] Toby Rufinus, Eric Engeström, Elliott Sales de Andrade, Aaron Hamilton, "*OpenGL*", accessed on 23th Feb 2016. https://open.gl

[39] OpenGL ES, https://www.khronos.org/opengles

[40] Aaftab Munshi, Dan Ginsburg, Dave Shreiner (2009), "*OpenGL ES 2.0 Programming Guide*", Addison-Wesley.

[41] eLinux.org, "*RPiconfig*", accessed on 23th Feb 2016. http://elinux.org/RPiconfig

[42] Wikipedia, "*Seam Carving*", accessed on 23th Feb 2016. https://en.wikipedia.org/wiki/Seam_carving

[43] Randima Fernando (2004), "*GPU Gems. Programming Techniques, Tips and Tricks for Real-Time Graphics*", Pearson Higher Education.

[44] Matt Pharr, Randima Fernando (2005), "*GPU Gems 2. Programming Techniques for High-Performance Graphics and General-Purpose Computation*", Addison-Wesley Professional.

[45] Hubert Nguyen (2007), "*GPU Gems 3*", Addison-Wesley Professional.

[46] Aaftab Munshi, Dan Ginsburg, Dave Shreiner (2009), *OpenGL ES 2.0 Programming Guide*, Addison-Wesley.

[47] Ben O'Steen (27th April 2012), "*Using OpenGL ES 2.0 on the Raspberry Pi without X windows*", Random Hacks Blog, accessed on 23th Feb 2016. https://benosteen.wordpress.com/2012/04/27/using-opengl-es-2-0-on-the-raspberry-pi-without-x-windows/

Ringraziamenti

Ringrazio in particolare il prof. Moreno Marzolla per tutta l'attenzione e la disponibilità che ha avuto nel seguirmi in questo percorso di tesi di laurea magistrale. Ringrazio inoltre tutti i compagni di corso con i quali ho condiviso molte ore di studio e lavorato ogni giorni fianco a fianco in questi ultimi anni. Infine, senza il cui supporto non sarei riuscito a raggiungere questo risultato, ringrazio i miei genitori per tutto il resto.