

ALMA MATER STUDIORUM - UNIVERSITA' DI BOLOGNA  
SEDE DI CESENA  
FACOLTA' DI SCIENZE MATEMATICHE, FISICHE E NATURALI  
CORSO DI LAUREA IN SCIENZE DELL'INFORMAZIONE

CALCOLO AD ALTE PRESTAZIONI  
SU ARCHITETTURA IBRIDA CPU - GPU

**Tesi di Laurea in**  
Biofisica delle Reti Neurali e loro applicazioni

**Relatore**

Chiar.mo Prof. Renato Campanini

**Presentata da**

Luca Benini

**Co-Relatore**

Dott. Matteo Roffilli

Sessione II

Anno Accademico 2005/2006



*The best way to predict the future is to invent it.*

*Alan Kay*



# Ringraziamenti

Nulla di tutto ciò che è presente in questo volume sarebbe stato possibile senza la disponibilità del prof. Campanini e del dott. Roffilli che con la loro esperienza e disponibilità mi hanno permesso di realizzare tutto ciò.

Vorrei ringraziare tutti i ragazzi dell'ex RadioLab, perchè solo dal caos può nascere una stella danzante.



# Indice

<b>1</b>	<b>Calcolabilità e HPC</b>	<b>1</b>
1.1	Calcolabilità . . . . .	1
1.1.1	Entscheidungsproblem e la Macchina di Turing . . . . .	1
1.2	Tecnologie per il calcolo ad alte prestazioni . . . . .	3
1.2.1	Supercomputer . . . . .	4
1.2.2	Processori Vettoriali . . . . .	5
1.2.3	SIMD . . . . .	7
1.2.4	Grid computing . . . . .	10
<b>2</b>	<b>Graphics Processing Unit</b>	<b>15</b>
2.1	Evoluzione delle unità grafiche programmabili . . . . .	15
2.2	Passi per la realizzazione di una vista 3D . . . . .	16
2.2.1	Primitive . . . . .	16
2.2.2	Frammenti . . . . .	17
2.2.3	Shader . . . . .	18
2.3	Architettura di una GPU . . . . .	20
2.3.1	NVIDIA GeForce 7800 GTX . . . . .	20
<b>3</b>	<b>C for graphics</b>	<b>25</b>
3.1	Caratteristiche del linguaggio . . . . .	25
3.1.1	Vantaggi dello shading di alto livello . . . . .	25
3.1.2	Language Profile . . . . .	26
3.1.3	Input e Output . . . . .	27
3.2	Funzionalità del linguaggio . . . . .	28
3.2.1	Tipi di dato . . . . .	28
3.2.2	Flusso di Controllo e operatori . . . . .	30

3.3	Uso del linguaggio . . . . .	31
3.3.1	Cg Runtime . . . . .	31
3.3.2	Inizializzazione dell'ambiente . . . . .	31
3.3.3	Caricamento del programma . . . . .	32
3.3.4	Preparazione dell'input e dell'output . . . . .	33
3.3.5	Upload dell'input . . . . .	34
3.3.6	Esecuzione . . . . .	34
3.3.7	Download dell'output . . . . .	36
<b>4</b>	<b>Prestazioni Elementari</b>	<b>37</b>
4.1	Costi di caricamento e fattori di convenienza . . . . .	37
4.1.1	PCI Express e caratteristiche hardware . . . . .	38
4.2	Valutazione degli operatori elementari . . . . .	40
4.2.1	Shader . . . . .	42
4.2.2	Risultati . . . . .	44
<b>5</b>	<b>Scalatura di Immagini</b>	<b>49</b>
5.1	Aspetti teorici . . . . .	49
5.1.1	Campionamento e Quantizzazione . . . . .	49
5.1.2	Scalatura di immagini digitali . . . . .	50
5.2	Interpolazione Nearest Neighbour . . . . .	50
5.3	Interpolazione Bilineare . . . . .	51
5.4	Aspetti implementativi . . . . .	54
5.4.1	Filtri per il ridimensionamento texture . . . . .	54
5.4.2	Texture Mapping . . . . .	55
5.4.3	Codice . . . . .	59
5.4.4	Risultati . . . . .	59
<b>6</b>	<b>Calcolo Ibrido</b>	<b>61</b>
6.1	Ragioni di un'architettura ibrida . . . . .	61
6.2	Applicazioni Multithreading . . . . .	62
6.3	Thread Library . . . . .	63
6.4	Aspetti Implementativi . . . . .	64
6.5	Risultati . . . . .	66



<b>7 Conclusioni</b>	<b>69</b>
<b>A Sistema di Testing</b>	<b>71</b>
<b>B YAW - Yet Another Wrapper</b>	<b>75</b>
<b>Bibliografia</b>	<b>78</b>



# Elenco delle tabelle

3.1	Semantiche Associative . . . . .	28
3.2	Tipi di dato scalari . . . . .	28
3.3	Tipi di dato vettoriali . . . . .	29
4.1	Prestazioni dei più noti BUS . . . . .	41



# Elenco delle figure

1.1	Earth Simulator 4 . . . . .	3
1.2	CDC 6600 . . . . .	4
1.3	Cray-1 . . . . .	5
1.4	Architettura simd . . . . .	7
1.5	Il primo processore dotato di istruzione SSE . . . . .	8
1.6	MMX e SSE a confronto . . . . .	9
1.7	Mappa della diffusione dei sistemi BOINC . . . . .	12
1.8	IBM Blue Gene . . . . .	12
2.1	Un esempio di applicazione di un Vertex Program . . . . .	17
2.2	Un esempio di applicazione di un Fragment Shader . . . . .	18
2.3	Il rendering di una scena 3D. . . . .	19
2.4	L'architettura della scheda 7800 GTX . . . . .	21
2.5	Architettura di un vertex shader. . . . .	22
2.6	Architettura di un pixel shader. . . . .	23
3.1	Architettura del runtime Cg . . . . .	27
4.1	Ciclo di Esecuzione . . . . .	38
4.2	Dettaglio dei connettori PCI Express . . . . .	39
4.3	Confronto per <code>.=</code> . . . . .	45
4.4	Confronto per <code>.cos</code> . . . . .	45
4.5	Confronto per <code>.sin</code> . . . . .	46
4.6	Costo di esecuzione per <code>.cos</code> . . . . .	47
4.7	Costo di esecuzione per <code>.sin</code> . . . . .	47
5.1	Interpolazione Nearest Neighbour . . . . .	51

5.2	Interpolazione Bilineare . . . . .	52
5.3	Interpolazione Bilineare con fattori di scala 16, 8, 4 e l'immagine originaria, successivamente riscalate con il filtro di Lanczos, per ottenere le medesime dimensioni. . . . .	53
5.4	MipMapping (sinistra) e Nearest Neighbour (destra) . . . . .	55
5.5	Processo del texture mapping. . . . .	56
5.6	Nearest Neighbour . . . . .	60
5.7	Interpolazione Bilineare . . . . .	60
6.1	Tempi per l'esecuzione sequenziale . . . . .	66
6.2	Tempi per l'esecuzione parallela . . . . .	67
A.1	Nvidia 7800 GT . . . . .	71
A.2	Nvidia 7300 . . . . .	72
A.3	Nvidia 5200 . . . . .	73

# Capitolo 1

## Calcolabilità e HPC

*The idea behind digital computers may be explained by saying  
that these machines are intended to carry out any operations  
which could be done by a human computer.*

*Alan Turing*

### 1.1 Calcolabilità

Durante il *Secondo Congresso Internazionale di Matematica* tenutosi nel 1900 a Parigi il matematico tedesco David Hilbert presentò una lista di 23 problemi di grande interesse le cui soluzioni dovevano essere un obiettivo primario per la comunità scientifica.

In particolare il secondo problema di Hilbert riguardava l'Assiomatizzazione dell'aritmetica e proprio a partire da questo problema e dai lavori a esso correlati si è sviluppata l'attuale teoria della calcolabilità.

#### 1.1.1 Entscheidungsproblem e la Macchina di Turing

Nel 1931 Kurt Gödel, in risposta al problema sollevato da Hilbert, presentò il *I Teorema di Incompletezza* dimostrando che in ogni sistema assiomatico sufficientemente espressivo, cioè da contenere almeno l'aritmetica, si può costruire una sentenza sui numeri naturali la quale:

- o non può essere né provata né refutata all'interno del sistema (*sistema incompleto*);

- o può essere provata e refutata all'interno del sistema (*sistema inconsistente*).

In altre parole ogni sistema assiomatico sufficientemente espressivo o è inconsistente o è incompleto. Un'altra formulazione informale è che non tutte le sentenze vere sono teoremi (cioè derivabili dagli assiomi usando le regole di inferenza del sistema). Gödel dimostrò inoltre (II Teorema di Incompletezza) che ogni sistema assiomatico sufficientemente espressivo non può provare la propria consistenza, risolvendo così in negativo il secondo problema di Hilbert.

Sul solco delle riflessioni inerenti gli aspetti logico-fondazionali della Matematica sollevata dal lavoro di Gödel, si sviluppò quella corrente di pensiero che riuscì in seguito a delineare il nucleo fondante dell'Informatica, cioè la Teoria della Computabilità, intesa come studio, modellizzazione e individuazione dei limiti relativi all'approccio computazionale basato sulle procedure effettive.

Di nuovo lo spunto iniziale partì da Hilbert, che nel 1928 scrisse con W. Ackermann il libro *Grundzüge der theoretischen Logik*[13]; in quest'opera compare per la prima volta l'enunciazione del famoso Entscheidungsproblem (Problema della decisione) per la Logica del Primo Ordine. In particolare all'interno del volume veniva presentato l'Entscheidungsproblem che richiedeva di trovare una procedura algoritmica per decidere se una qualunque formula nella logica dei predicati è valida. Il problema fu risolto indipendentemente da Alonzo Church, che pubblicò nel 1936 un articolo intitolato *An Unsolvability Problem in Elementary Number Theory*[6], e da Alan Turing, che nello stesso anno pubblicò l'articolo *On Computable Numbers, with an Application to the Entscheidungsproblem*[19]. Essi dimostrarono, con argomentazioni molto diverse, la non esistenza di un siffatto algoritmo.

Pertanto, in particolare, è impossibile decidere algebricamente se una qualunque sentenza sui numeri naturali è vera o meno.

Il lavoro di Church fu l'atto di nascita di un formalismo matematico, denominato  $\lambda$ -calcolo, che costituisce un vero e proprio modello di computazione. L'approccio di Turing, basato su un semplice dispositivo, chiamato Macchina di Turing (MdT), che oggi riconosciamo come la descrizione del primo modello formale di calcolatore, faceva riferimento al problema della fermata della macchina di Turing, dimostrando che, assegnata una qualunque MdT, non è possibile decidere algebricamente se essa si fermerà o meno a partire da certe condizioni iniziali. Il successivo concetto di macchina di Turing Universale, cioè di una macchina che sia in grado di simulare la





Figura 1.1: Earth Simulator 4

computazione di qualunque altra macchina, getta poi le basi teoriche del calcolatore programmabile, realizzato in seguito da Von Neumann

## 1.2 Tecnologie per il calcolo ad alte prestazioni

Il lavoro teorico di Von Neumann e le successive implementazioni hanno portato alla realizzazione, attraverso il corso degli anni, di un insieme di tecnologie e metodologie in grado di realizzare generazioni di calcolatori sempre più performanti.

Oggi con poche migliaia di euro è possibile acquistare un personal computer con migliori prestazioni, più memoria, sia principale che secondaria, di un computer acquistato negli anni '70 per un miliardo di lire.

Qual è il vantaggio di costruire computer sempre più veloci? Perché non potremmo utilizzare una macchina che impiega un mese o una settimana invece di un giorno o di un'ora? Per molti problemi si può anche fare così. Ma il fatto è che oggi stiamo cominciando a raggiungere la potenza di calcolo sufficiente per capire il comportamento di sistemi con migliaia o milioni di variabili; le macchine più veloci sono già in grado di prevedere ciò che può succedere. Si prenda, per esempio, i gas a effetto serra e le modalità con cui stanno modificando il clima globale, uno dei problemi per i quali è stato costruito Earth Simulator4. Con i computer abbastanza veloci da predire con precisione i cambiamenti climatici, possiamo conoscere con un certo grado di precisione quale percentuale di anidride carbonica nell'atmosfera determina lo scioglimento delle calotte polari.

Analogamente, dato che Earth Simulator costruisce modelli del clima del pianeta a un livello incredibile di granularità, è in grado di eseguire simulazioni che tengano

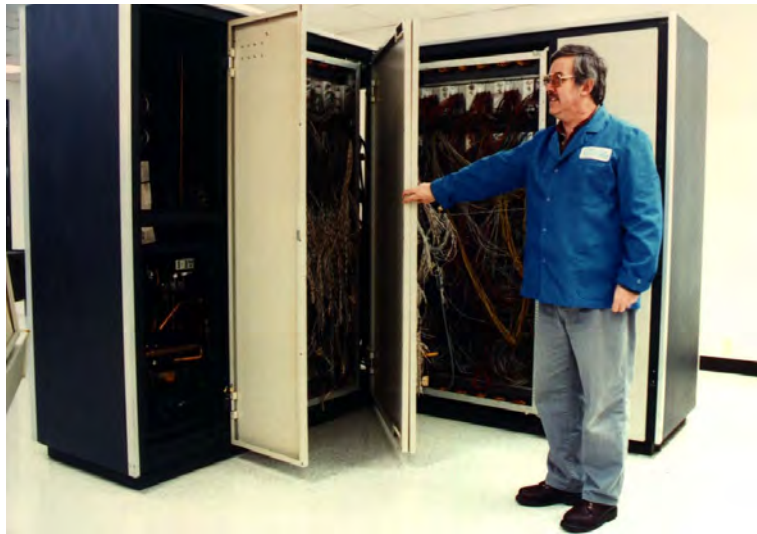


Figura 1.2: CDC 6600

conto di fenomeni su scala locale come i temporali. Tali fenomeni possono riguardare aree estese solo 10 chilometri, contro i 30-50 chilometri di risoluzione delle griglie adottate normalmente per i modelli meteorologici.

Più recentemente è stato mostrato [17] come anche il campo del Machine Learning, in special modo nell'ambito delle Support Vector Machine, possa trarre beneficio da forti incrementi prestazionali [4], [17],[2]. Per queste necessità di potenza di calcolo si sono seguite numerose strade ognuna delle quali può essere impiegata in congiunzione con le altre andando a delineare la complessità del campo del calcolo ad alte prestazioni, da un lato come continua ricerca di nuovi approcci e architetture, dall'altro come individuazione degli equilibri più performanti tra i vari approcci possibili.

### 1.2.1 Supercomputer

Tra gli anni '60 e la metà degli anni '70 la società CDC (*Control Data Corporation*) con i suoi supercomputer fu l'azienda leader nel settore HPC. I tecnici della CDC per ottenere elevate prestazioni di calcolo svilupparono diverse soluzioni tecnologiche come l'utilizzo di processori specializzati per i diversi compiti (CDC 6600), l'utilizzo di pipeline (CDC 7600) e l'utilizzo di processori vettoriali (CDC STAR-100). Scelte strategiche della CDC rischiarono di far fallire la società e alcuni tecnici insoddisfatti dei dirigenti della società abbandonarono la CDC per tentare nuove strade. Tra

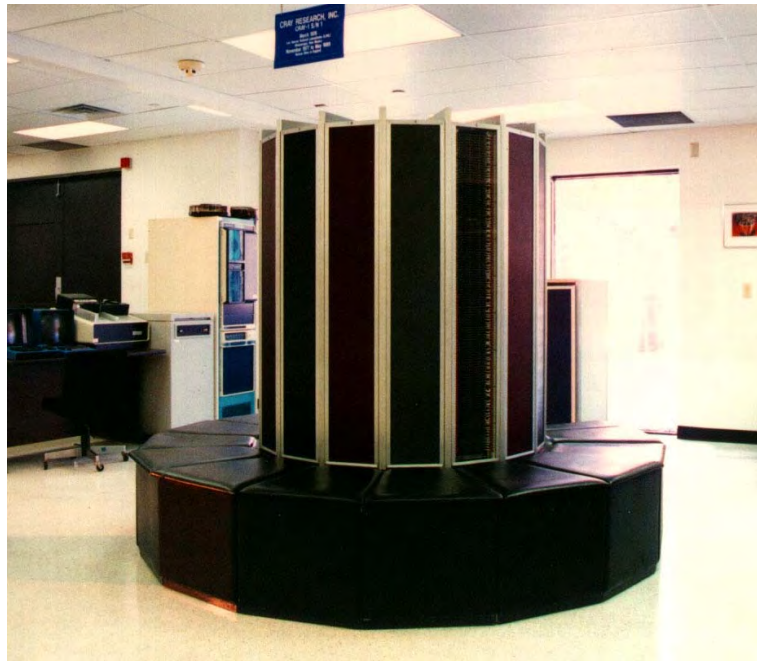


Figura 1.3: Cray-1

questi il più famoso fu Seymour Cray che con il Cray-1 segnò una nuova strada per le soluzioni tecnologiche utilizzate e per l'elevata potenza di calcolo sviluppata.

Dalla metà degli anni '70 fino alla fine degli anni '80 la Cray Research rappresentò il leader nel settore del supercalcolo. Cray estese le innovazioni utilizzate nelle macchine della CDC portandole al loro estremo ed utilizzando soluzioni innovative come il raffreddamento a liquido o delle strutture a torre ove alloggiare le schede con le unità di calcolo in modo da ridurre la lunghezza media delle connessioni.

### 1.2.2 Processori Vettoriali

L'idea dei processori vettoriali nacque originariamente alla fine degli anni '60 presso la Westinghouse durante il progetto Solomon che mirava ad incrementare drasticamente le prestazioni matematiche utilizzando un elevato numero di semplici coprocessori matematici (o ALU) sotto il controllo di una singola CPU. La CPU avrebbe dovuto inviare l'operazione matematica comune a tutte le ALU, le ALU avrebbero prelevato i dati e dopo averli elaborati li avrebbero salvati. Questo avrebbe consentito lo svolgimento di un singolo algoritmo su molti dati contemporaneamente. Nel 1962 la Westinghouse decise di cancellare il progetto ma alcuni ricercatori fuoriusciti dalla società convinsero l'University of Illinois a riavviare le ricerche e a sviluppare

l'ILLIAC IV. Inizialmente la macchina doveva essere dotata di 256 processori elementari e doveva sviluppare 1 Gigaflops ma nel 1972 quando venne presentata la macchina questa era dotata di 64 processori e sviluppava solo 100 Megaflops (150 megaflops di picco). Nonostante l'ILLIAC IV sia considerato un insuccesso nelle applicazioni che trattavano elevate quantità di dati come la fluidodinamica computazionale la macchina era il più veloce supercomputer del pianeta.

La prima implementazione vincente dei processori vettoriali si ebbe con la CDC STAR-100 e con il Texas Instruments ASC (*Advanced Scientific Computer*). L'ALU base dell'ASC utilizzava un'architettura a pipeline per eseguire le operazioni scalari e vettoriali ed era in grado di sviluppare 20 Megaflops di picco durante l'esecuzione delle operazioni vettoriali. ALU estese gestivano due o quattro pipeline e quindi le prestazioni si raddoppiavano o si quadruplicavano. La banda fornita dal sottosistema di memoria era in grado di soddisfare le ALU estese. Invece la STAR-100 era più lenta delle altre macchine della CDC come il CDC 7600 dato che nel caso dell'esecuzione di istruzioni scalari la sua decodifica era lenta. Nel caso dell'esecuzione delle istruzioni vettoriali invece la macchina era veloce nella decodifica e quindi questo compensava la lentezza delle operazioni scalari. La tecnica vettoriale venne sfruttata in pieno dal famoso Cray-1.

Questo computer a differenza dello STAR-100 o dell'ASC non manteneva i dati esclusivamente in memoria ma aveva anche 8 registri vettoriali a 64 bit nella CPU. Le istruzioni vettoriali erano eseguite sui dati memorizzati nei registri e quindi erano molto più veloci di quelle svolte accedendo alla memoria principale. In aggiunta il progetto prevedeva diverse pipeline per le diverse operazioni e quindi la pipeline di somma/sottrazione era diversa dalla pipeline delle moltiplicazioni. Quindi più istruzioni vettoriali potevano essere concatenate ed eseguite simultaneamente. Il Cray-1 era in grado di eseguire 80 MIPS ma concatenando fino a tre serie di istruzioni si poteva arrivare a 240 MIPS.

Mentre lo sviluppo dei supercomputer si diresse verso l'impiego del parallelismo sia a livello di microprocessore che di calcolatore, le idee alla base dei processori vettoriali vennero introdotte anche nei personal computer con l'introduzione, a partire dal Pentium PRO, dell'architettura SIMD (*Single Instruction, Multiple Data*).

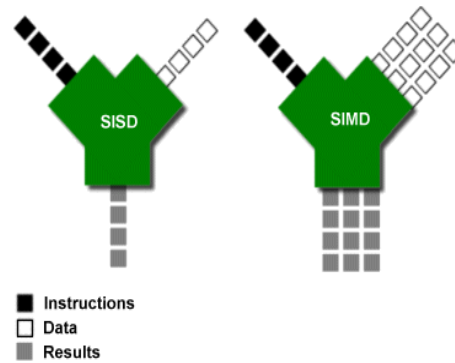


Figura 1.4: Architettura simd

### 1.2.3 SIMD

SIMD è un'architettura in cui più unità elaborano dati diversi in parallelo. Questa architettura viene utilizzata da processori vettoriali o da processori che funzionano in parallelo. In passato venivano prodotti un numero elevato di dispositivi dedicati allo svolgimento di compiti specifici. Usualmente questi dispositivi erano DSP (*Digital Signal Processor*) opportunamente programmati. La differenza fondamentale tra le istruzioni SIMD e i DSP è che questi sono dotati di un set di istruzioni completo e quindi sono in grado di svolgere teoricamente qualsiasi compito. Al contrario le istruzioni SIMD sono progettate per manipolare elevate quantità di dati in parallelo e per le usuali operazioni si appoggiano ad un altro insieme di istruzioni usualmente gestito dal microprocessore. Inoltre i DSP tendono a includere un certo numero di istruzioni dedicate ad elaborare tipi specifici di dati come possono essere i dati audio o video mentre le istruzioni SIMD vengono utilizzate per elaborare dati generici. Nell'elaborazione di dati multimediali spesso si incontrano algoritmi che possono avvantaggiarsi di un'architettura SIMD. Per esempio per cambiare la luminosità di un'immagine un microprocessore dovrebbe caricare ogni pixel che compone l'immagine nei suoi registri, effettuare la modifica della luminosità e poi salvare i risultati in memoria. Un processore SIMD eseguirebbe prima un'istruzione che caricherebbe con un'unica operazione un certo numero di pixel (il numero preciso dipende dall'architettura) poi il processore modificherebbe tutti i dati in parallelo e in seguito li salverebbe tutti contemporaneamente in memoria. Eseguire le operazioni a blocchi invece che agire sui singoli pixel rende le operazioni molto più efficienti dato che i moderni computer sono progettati per trasferire i dati a blocchi e sono inefficienti

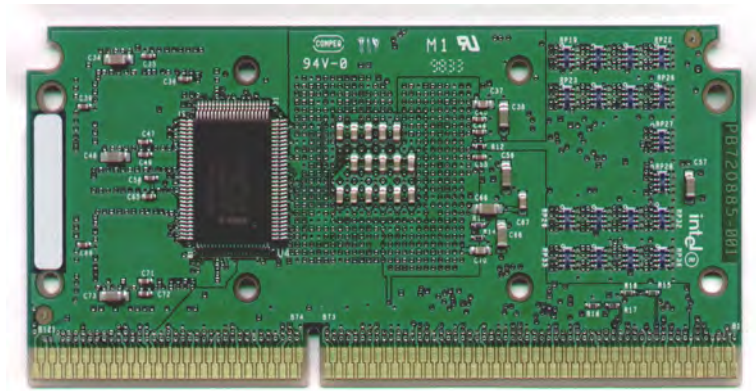


Figura 1.5: Il primo processore dotato di istruzione SSE

nei singoli accessi alla memoria. Un altro vantaggio deriva dal fatto che tipicamente le istruzioni SIMD sono sempre in grado di manipolare tutti i dati caricati contemporaneamente: quindi se un processore SIMD è in grado di caricare 8 dati, questo sarà anche in grado di processarli tutti contemporaneamente. La prima architettura SIMD a essere disponibile commercialmente fu l'architettura MMX realizzata da Intel.

Le istruzioni MMX tuttavia fallirono l'obiettivo di portare i benefici del paradigma SIMD nei comuni personal computer. E questo si era avuto per due motivazioni principali: lo scarso supporto fornito da Intel agli sviluppatori e la mancanza di istruzioni utilizzabili nell'emergente mondo della grafica 3D. Le istruzioni MMX manipolano numeri interi e quindi non sono in grado di gestire le trasformazioni geometriche dei videogiochi perchè in questo compito sono richieste operazioni floating point; inoltre l'utilità di un set di istruzioni capace di accelerare tali calcoli si è resa visibile in maniera pesante con l'affermazione di API dedicate alla gestione del 3D quali le Direct 3D e OpenGL, capaci di astrarre lo sviluppatore dal codice ottimizzato vero e proprio.

Questo fatto ha portato Intel (così come AMD con le sue 3DNow!) a sviluppare una estensione del set di istruzioni, in maniera analoga a quanto fatto con gli interi, anche per le operazioni in virgola mobile a singola precisione chiamata Internet SSE (Internet Streaming SIMD Extension) o più semplicemente SSE. L'obiettivo propostosi dai progettisti Intel era di raggiungere un incremento delle prestazioni floating point tra il 70% e il 100%, ritenuto sufficiente a rendere percettibile la differenza e quindi competitivo il prodotto, al minor costo possibile in termini di incremento

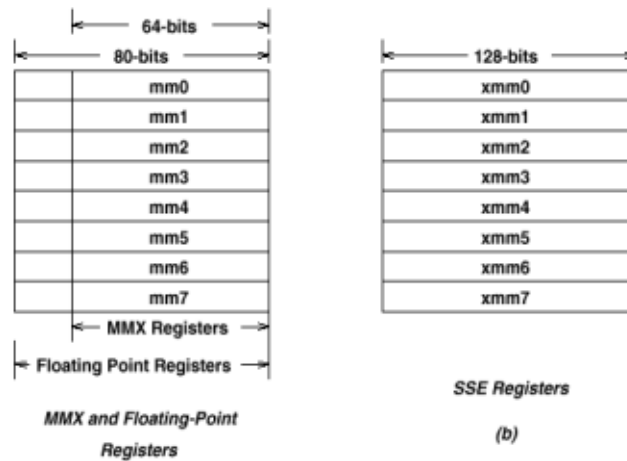


Figura 1.6: MMX e SSE a confronto

della complessità e aumento delle dimensioni del die.

Nel contempo si decise di estendere la tecnologia MMX (quale ad esempio istruzioni per facilitare le codifiche in tempo reale di tipo MPEG-2) e di introdurre istruzioni per mascherare la latenza che deriva dalle notevoli dimensioni, in termini di memoria, dei dati implicati in applicazioni video. Il termine *Streaming* si riferisce appunto alla presenza di istruzioni che permettono il prefetch di dati simultaneamente all'elaborazione di altri già disponibili velocizzando il flusso (stream) dei dati in ingresso e in uscita dal processore nascondendo nel tempo di esecuzione la latenza del fetch. Una delle scelte basilari nella definizione di un'architettura SIMD consiste nel definire su quanti dati contemporaneamente si vuole operare in modo da raggrupparli in un vettore di dimensioni adeguate, che costituirà il nuovo tipo di dato cui faranno riferimento le istruzioni SIMD.

Il team di sviluppatori di Intel ritenne che la computazione parallela di 4 floating point a singola precisione (32 bit) e quindi di un data-type SSE da 128 bit consentisse un raddoppio complessivo delle performance senza aggiungere eccessiva complessità essendo tendenzialmente ottenibile con un doppio ciclo della esistente architettura a 64 bit. La scelta di operare su 2 floating point non avrebbe consentito di ottenere paragonabili prestazioni mentre l'adozione di un datapath di 256 bit (8 FP da 32 bit) avrebbe determinato un impatto maggiore in termini di complessità. Mentre i 128 bit possono essere separati in 2 istruzioni da 64 bit che possono essere eseguite, come vedremo, in un ciclo, con 256 bit si sarebbe dovuto, per mantenere lo stesso

throughput, raddoppiare la larghezza delle unità di esecuzione e quindi la banda di memoria per alimentarle. Stabilito il datapath di 128 bit si poneva la domanda se implementare i registri a 128 bit nei registri MMX/x87 esistenti oppure definire un nuovo stato con registri appositi. La prima scelta, analoga a quella attuata con l'estensione alla tecnologia MMX, avrebbe comportato il vantaggio della piena compatibilità con il sistema operativo ma lo svantaggio di dover condividere i registri, già penalizzanti, della architettura IA-32. La seconda scelta avrebbe comportato il problema di dover adattare i sistemi operativi, problema poco sentito da Intel data la sua forza contrattuale, ma avrebbe avuto il vantaggio di facilitare i programmatori e la possibilità di eseguire contemporaneamente istruzioni MMX, x87 o SIMD-FP. I progettisti Intel optarono per aggiungere un nuovo stato architetturale, per la prima volta dai tempi dell'aggiunta di quello x87 ai tempi del i386 nel 1985, con la definizione di 8 nuovi registri da 128 bit (chiamati registri XMM), cosa che non era stata fatta con l'introduzione delle istruzioni MMX che operavano sugli stesi registri fisici della Floating Point Unit.

Benché le istruzioni SSE abbiano introdotto un utile parallelismo delle istruzioni floating point permettendo incrementi di prestazioni in molti settori del processing multimediale, ci sono ancora tipologie di operazioni non supportate come le moltiplicazioni tra interi a 32bit importanti per il processing audio di qualità. Per risolvere tutti questi problemi, Intel ha introdotto nel suo Pentium4 ben 144 nuove istruzioni SIMD che prendono il nome di SSE2, che hanno colmato buona parte delle mancanze presenti in SSE.

Sempre Intel ha introdotto, a partire dalla generazione Prescott, le istruzioni SSE3 che senza andare a incrementare sostanzialmente le potenzialità delle architetture SIMD ne hanno semplificato l'utilizzo.

### **1.2.4 Grid computing**

Grid computing (letteralmente, calcolo a griglia) indica un'infrastruttura distribuita per consentire l'utilizzo di risorse di calcolo e di storage provenienti da un numero indistinto di calcolatori eterogenei (anche e soprattutto di potenza non particolarmente elevata) interconnessi da una rete (solitamente, ma non necessariamente, Internet). L'idea del Grid computing, di cui recentemente si sente spesso parlare come la prossima rivoluzione dell'informatica (come a suo tempo fu il World Wide



Web), risale però a circa metà degli anni Novanta. Le 'griglie di calcolo' vengono prevalentemente utilizzate per risolvere problemi computazionali di larga scala in ambito scientifico e ingegneristico. Sviluppatesi originariamente in seno alla fisica delle alte energie (in inglese HEP), il loro impiego è già da oggi esteso alla biologia, all'astronomia e in maniera minore anche ad altri settori. I maggiori player dell'IT in ambito commerciale hanno già da tempo cominciato ad interessarsi al fenomeno, collaborando ai principali progetti grid world-wide con sponsorizzazioni o sviluppando propri progetti grid in vista di un utilizzo finalizzato al mondo del commercio e dell'impresa[16]. Una grid è in grado di fornire agli utenti di un gruppo scalabile senza una particolare caratterizzazione geografica (gergalmente detto VO ossia Virtual Organization) la potenzialità di accedere alla capacità di calcolo e di memoria di un sistema distribuito, garantendo un accesso coordinato e controllato alle risorse condivise e offrendo all'utente la visibilità di un unico sistema di calcolo logico cui sottomettere i propri job. L'idea del Grid computing è scaturita dalla constatazione che in media l'utilizzo delle risorse informatiche di una organizzazione è pari al 5% della sua reale potenzialità. Le risorse necessarie sarebbero messe a disposizione da varie entità in modo da creare un'organizzazione virtuale con a disposizione un'infrastruttura migliore di quella che la singola entità potrebbe sostenere. Il progetto SETI@home[1], lanciato nel 1999 da Dan Werthimer, è un esempio molto noto di un progetto, seppur semplice, di Grid computing. SETI@Home è stato seguito poi da tanti altri progetti simili nel campo della matematica e della microbiologia.

Attualmente, la più importante grid europea è quella del CERN di Ginevra che ora si chiama gLite[8] (precedentemente LCG e prima ancora DataGrid), sviluppata da un team italo-ceco e prevalentemente presso l'INFN, l'Istituto Nazionale di Fisica Nucleare con la collaborazione di Datamat spa. A differenza di quella utilizzata da SETI@Home, attualmente una grid viene concepita prevedendo un livello di middleware fra le risorse di calcolo e memoria e gli utenti della grid stessa. Lo scopo principale del middleware è quello di effettuare il cosiddetto match-making, ossia l'accoppiamento tra le risorse richieste e quelle disponibili in modo da garantire il dispatching dei job nelle condizioni migliori avendo sempre visibilità dello stato dell'intera grid. L'architettura GRID presenta dei punti di forza e di debolezza che vanno compresi a fondo prima di intraprendere la realizzazione di una simile architettura. L'utilizzo di un insieme di calcolatori di media potenza anziché di un

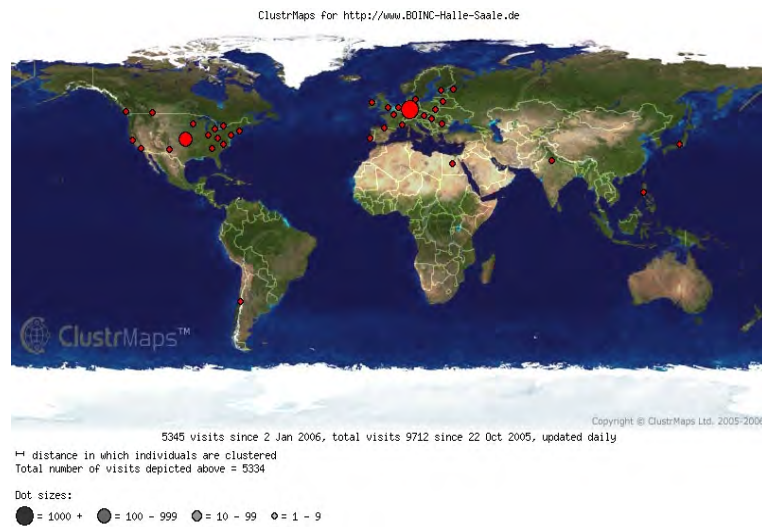


Figura 1.7: Mappa della diffusione dei sistemi BOINC

singolo calcolatore estremamente potente offre la possibilità di ottenere una scalabilità economica maggiore, rendendo possibile l'espandibilità della potenza di calcolo senza dover riammodernare completamente il parco macchine. Naturalmente la rea-



Figura 1.8: IBM Blue Gene

lizzazione di una grid *in house*, cioè con calcolatori di esclusivo possesso dell'ente operante richiede spazi e sistemi di raffreddamento adeguati, che per grid di grandi dimensioni possono rappresentare un problema serio. Inoltre all'aumentare dei dispositivi si riduce il MTBF (*Mean Time Between Failure*), cioè il tempo medio tra

due guasti. Si rende necessario sviluppare il sistema grid in modo da presentare un basso failover. Inoltre un sistema di grid deve implementare delle adeguate politiche per lo scambio dei dati al fine di combinare i risultati parziali ottenuti, problematica ancor più complesse per i sistemi grid che utilizzano la rete internet anzichè reti locali, risultati quindi più soggette a perdita di pacchetti con il conseguente rischio di corruzione dei risultati. In generale i limiti tecnologici spingono sempre di più all'adozione di sistemi di grid in quanto è possibile ottenere potenza di calcolo unificata estramente maggiore di quella ottenibile con sistemi singoli. Basti pensare che il sistema Seti@home è in grado di sviluppare una potenza di calcolo pari a 74.95 Teraflops (reali) mentre il prototipo Blue Gene/L realizzato da IBM impiegando 32768 processori PowerPC a due core personalizzati per l'occasione è in grado di sviluppare una potenza di calcolo (di picco) pari a circa 70 Teraflops. Da questi numeri emerge immediatamente la ragione del crescente interesse relativo ai sistemi grid.



# Capitolo 2

## Graphics Processing Unit

*When any device gets smart enough, someone,  
somewhere ports the classic first-person shooter to it,  
simply because they can.  
Wired about Doom*

### 2.1 Evoluzione delle unità grafiche programmabili

Una GPU (*Graphics Processing Unit*) è un dispositivo hardware dedicato al rendering grafico impiegato in personal computer o in console. Nota anche come *Video Display Processor* ha iniziato a diffondersi alla fine degli anni '80 come chip integrato per evolversi, negli anni recenti, fino a diventare una scheda di espansione su bus preferenziale.

Una GPU fornisce al programmatore la possibilità di svolgere un sottoinsieme più o meno ampio dei principali operatori grafici sfruttando le potenzialità dell'accelerazione hardware.

Nelle istanze moderne le GPU si presentano come dispositivi estremamente performanti in grado di elaborare con picchi fino a 500 Gflops/s (ATI X1900XT) che paragonati ai 12 Gflops/s ottenibili su un Pentium4 3 GHz risultano sicuramente un aspetto che merita di essere approfondito da tutti coloro che si occupano di calcolo ad alte prestazioni.

I punti di forza con cui le GPU riescono a ottenere picchi di tale consistenza possono essere sostanzialmente identificati in:

- frequenza di clock;
- velocità della memoria primaria;
- volume delle informazioni processate in parallelo.

Nel corso degli anni si è assistito a un vertiginoso miglioramento dei tre parametri appena visti, con un drastico aumento delle possibilità offerte da questi dispositivi. Il realismo attualmente raggiunto dagli attuali software per l'intrattenimento è allo stesso tempo la conferma e la concausa di questa crescente capacità computazionale, dalla mole di calcoli necessarie per renderizzare una scena 3D con un alto livello di dettaglio e con la presenza di tutti gli effetti ambientali (luci, ombre, distorsione delle immagini dovuta al calore, ai corpi liquidi, ecc) che rappresentano il requisito minimo per un software videoludico attuale.

## 2.2 Passi per la realizzazione di una vista 3D

Date le ragioni evolutive delle GPU per poterne apprezzare appieno l'architettura è necessario andare a scomporre in passi elementari il processo che porta alla realizzazione di una vista 3D.

### 2.2.1 Primitive

Un oggetto nello spazio tridimensionale viene definito attraverso la forma e il colore; per poter elaborare un oggetto su un calcolatore e poterlo poi eventualmente visualizzare su un dispositivo di output bidimensionale è necessario discretizzarlo andando ad esprimere numericamente le due caratteristiche.

Dato un sistema di assi cartesiani è possibile campionare la forma dell'oggetto andando a selezionare un opportuno sottoinsieme di punti significativi memorizzabile come vertici nello spazio tridimensionale, a partire da questi vertici la forma verrà modellata impiegando figure piane elementari (solitamente triangoli).

### 2.2.2 Frammenti

I triangoli vengono poi suddivisi in quadrati di piccole dimensioni, chiamati frammenti, ognuno dei quali memorizza le informazioni relative al colore, in particolare i valori del rosso, del blu e del verde, oltre all'informazione relativa alla trasparenza del frammento.

Per migliorare il realismo dell'oggetto e per tenere conto degli aspetti di ripetitività propri di un'immagine reale, anziché memorizzare direttamente il valore del colore memorizzato da un frammento si impiegano le texture.

Le texture sono immagini che vengono mappate in modo opportuno sull'oggetto di



Figura 2.1: Un esempio di applicazione di un Vertex Program

interesse: con l'applicazione di una texture il frammento prenderà il valore appropriato a partire dalla texture.

La semplice applicazione di una texture ad un frammento non è sufficiente per ottenere un buon realismo: ciò che l'occhio umano percepisce dipende sia dalla forma dell'oggetto, sia dall'aspetto della superficie sia dal comportamento dinamico della superficie nel suo interagire con le condizioni ambientali presenti nella scena.

### 2.2.3 Shader

Gli shader sono programmi utilizzati per determinare la superficie finale di ogni singolo frammento, includendo, tra le altre cose, le componenti necessarie per il calcolo dell'assorbimento della luce, il texture mapping, la riflettività delle superfici e la loro distorsione.

E' evidente come già dalla definizione di uno shader emerge la possibilità di implementarli in modo parallelo, permettendo di ridurre drasticamente il tempo di calcolo necessario.

Data la loro vocazione grafica gli shader vengono scritti con appositi linguaggi, detti linguaggi di shading, progettati tenendo conto dei punti di forza e di debolezza



Figura 2.2: Un esempio di applicazione di un Fragment Shader

intrinseci nel modello computazione.

Gli shader vengono impiegati principalmente per l'elaborazione di:

- vertici;
- frammenti;

Nel caso venga applicato ad un insieme di vertici lo shader produrrà come output la posizione spaziale finale del vertice, mentre nel caso sia applicato ad un frammento



produrrà in output il suo aspetto finale. Operando sui frammenti sarà possibile esprimere in maniera algoritmica e dinamica le trasformazioni da applicare ad ogni singolo frammento, permettendo di ottenere, tra gli altri, gli effetti di multitexturing[11] e bump mapping[18] che si stanno diffondendo rapidamente nel mondo della grafica 3D data la loro capacità di conferire realismo alla scena.

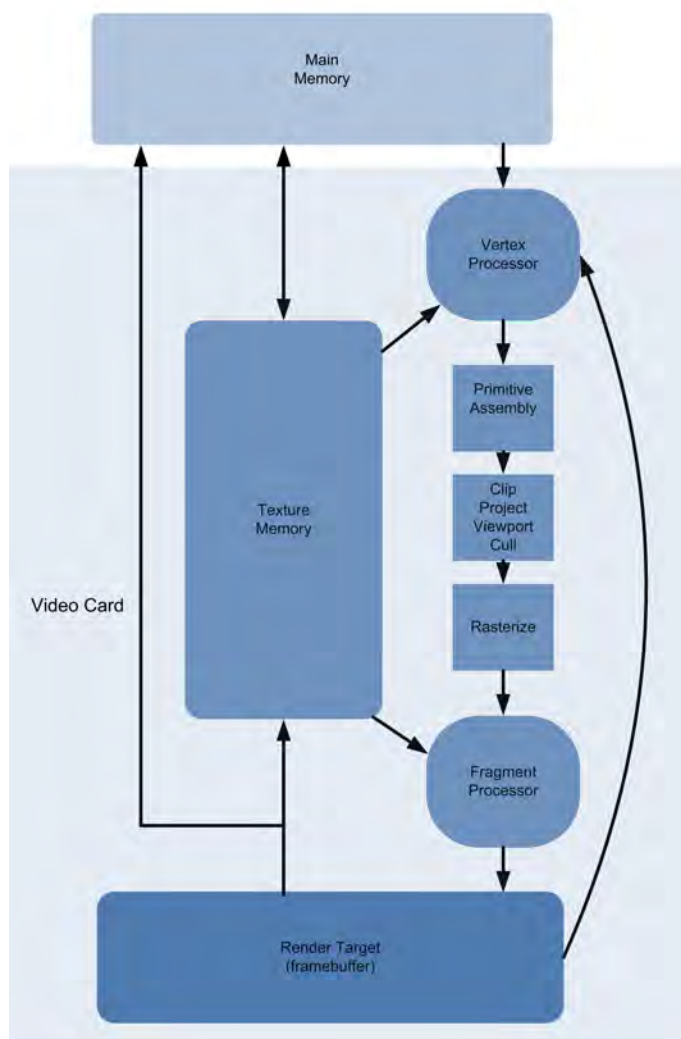


Figura 2.3: Il rendering di una scena 3D.

## 2.3 Architettura di una GPU

Data la natura stessa delle operazioni che abbiamo visto costituire i passi necessari alla visualizzazione di una scena 3D il modello computazionale scelto come paradigma per la realizzazione dell'hardware componente la Graphics Processing Unit si è immediatamente identificato nel modello *stream processor* dove una serie di unità computazionali eterogenee elaborano un flusso di informazioni comune, che rappresenta il canale di comunicazione tra le unità di calcolo.

L'architettura scelta non è tuttavia sufficiente a giustificare il crescente interesse per le Graphics Processing Unit che si sta registrando nel campo del calcolo ad alte prestazioni.

Per esplicitare le reali potenzialità computazionali di questi dispositivi è necessario prendere in considerazione un'implementazione reale che permette di comprendere appieno le potenzialità.

### 2.3.1 NVIDIA GeForce 7800 GTX

Prendiamo come prototipo da presentare la scheda video NVIDIA GeForce 7800 GTX che mostra bene i punti di forza delle GPU.

L'architettura G70 di questa scheda si basa su un redesign dell'architettura NV40 diffusa sui precedenti modelli della serie 6x, il cui modello di punta, la GeForce 6800, ha rappresentato un forte balzo in avanti prestazione con l'introduzione della tecnologia SLI, che offre la possibilità di impiegare due schede video cooperanti in modo parallelo per il rendering di una singola scena.

Vediamo in 2.4 la macrostruttura a blocchi della scheda in esame: L'architettura, che a una prima osservazione può risultare estremamente complessa, presenta in realtà una mappatura uno a uno con i concetti esposti nella sezione precedente, dove abbiamo esaminato da un punto di vista teorico i requisiti hardware di un dispositivo in grado di effettuare il rendering di una scena 3D.

Nella parte alta della figura 2.4 è possibile osservare otto unità parallele che rappresentano la risposta alla problematica principale nel design di un vertex shader, cioè il rischio che questa unità si trasformi nel collo di bottiglia della pipeline grafica. Per evitare questo problema è ormai diffuso il ricorso, come in questo caso, a un

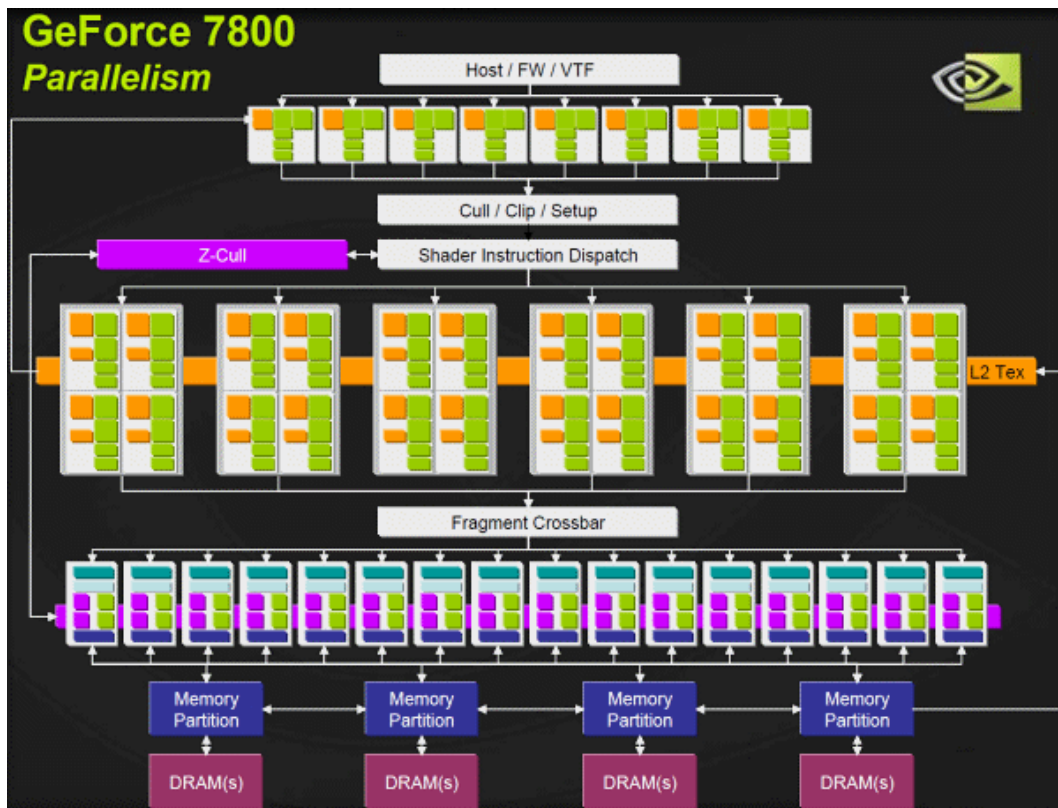


Figura 2.4: L'architettura della scheda 7800 GTX

massiccio parallelismo.

La manipolazione dei vertici è un'attività estremamente parallelizzabile da cui emerge come le otto unità siano in grado, in generale, di operare in simultanea alla massima velocità possibile permettendo di ottenere ottime prestazioni.

Tuttavia la maggior parte degli sforzi realizzativi è concentrata sui pixel shader, che occupano la parte centrale del diagramma. Questa attenzione è giustificata dall'importanza che la manipolazione dei fragment riveste nella realizzazione di scene 3D ad alto realismo.

Nell'architettura G70 i pixel shader sono suddivisi in gruppi di quattro unità, dove ognuna della unità del gruppo condivide alcune risorse con le altre tre unità, nonostante il raggruppamento in un quadrato 2x2 rappresenti in modo grafico un aspetto implementativo, bisogna specificare come non esista l'equivalente del concetto di "unità confinante". Infatti tutte le unità presenti in un gruppo sono in grado di operare parallelamente su quattro frammenti per volta. La presenza di sei gruppi all'interno del diagramma mostra come la NVIDIA GeForce 7800 sia in grado di processare fino a 24 fragment in modo totalmente parallelo.



Figura 2.5: Architettura di un vertex shader.

I pixel shader sono inoltre stati completamente ridisegnati prendendo in considerazione i requisiti dei più comuni algoritmi grafici in modo da fornire il massimo supporto alle operazioni più comuni, basti pensare come l'operazione elementare MADD composta da una moltiplicazione e una somma, rendendo possibile la valutazione di espressioni nella forma  $a * x + b$  è stata drasticamente migliorata, rendendola in grado quindi di effettuare due MADD per ciclo di clock. Teniamo a precisare che ogni singolo parametro dell'operatore MADD può essere composta da quattro valori float, permettendo quindi una prima stima delle potenzialità della scheda.

Considerando le 24 unità in grado svolgere due MADD per ciclo di clock e dato che ogni operatore di MADD può essere composto da 4 float, si ha che la scheda è in grado di operare su 384 valori float per ciclo di clock, prendendo come valore di riferimento la frequenza della scheda si ottiene una potenzialità pari a circa 165 GFlops, superando di gran lunga il picco teorico di un'unità floating point non parallela, che considerando una frequenza di clock ipotetica di 4GHz con la possibilità di operare su due float per ciclo di clock porta a un picco teorico di 8GFlops, che possono essere tranquillamente raggiunti e superati impiegando le tecnologie SIMD, fornendo un fattore moltiplicativo pari a un limite di 4x.

Ogni singolo pixel shader comprende inoltre due cache (condivise con gli altri pixel shader del gruppo) di primo e secondo livello interposte tra le unità di calcolo e la

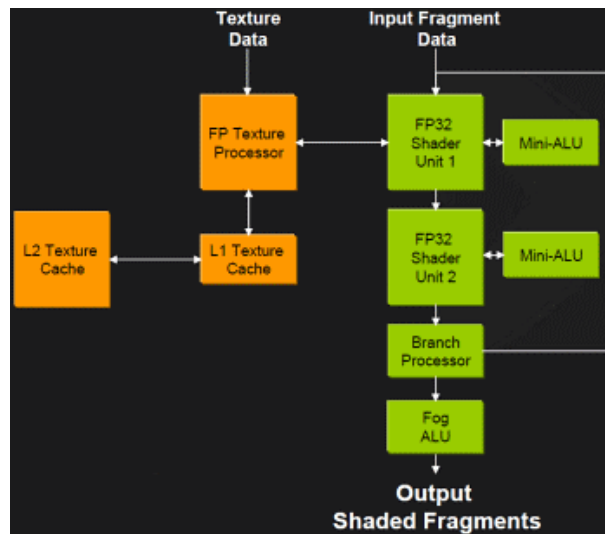


Figura 2.6: Architettura di un pixel shader.

memoria della scheda video dove vengono memorizzate le texture da applicare durante la realizzazione della scena 3D in corso, portando a una drastica riduzione dei tempi di accesso alla memoria on-board che già individualmente presenta delle ottime prestazioni. La massività del parallelismo associato ai pixel shader è ancor più evidente nelle schede realizzate da ATI: tuttavia bisogna notare come l'architettura stessa delle recenti schede video ATI si presenta estremamente efficace nell'applicazione di effetti che richiedono un numero limitato di texture sorgenti, rendendo quindi spesso necessario ricorrere al rendering *multipass* con un conseguente calo delle prestazioni.

La necessità di un alto numero di texture in input, unitamente alla scarsa documentazione interna ATI ha reso di fatto NVIDIA il leader incontrastato per quanto riguarda la scelta di schede video per il calcolo General Purpose.



# Capitolo 3

## C for graphics

*Programmers have a well-deserved reputation for working long hours but are rarely credited with being driven by creative fevers. Programmers talk about software development on weekends, vacations, and over meals not because they lack imagination, but because their imagination reveals worlds that others cannot see.*

*Larry O'Brien and Bruce Eckel*

### 3.1 Caratteristiche del linguaggio

#### 3.1.1 Vantaggi dello shading di alto livello

Storicamente l'hardware grafico è stato programmato ad un livello di astrazione molto basso, dove le pipeline a funzioni predefinite venivano configurate attraverso l'impiego di apposite istruzioni assembler.

Questo approccio è stato ormai del tutto abbandonato per fare posto a una programmabilità diretta dei dispositivi. Nonostante l'uso di un linguaggio di basso livello sia sufficiente a garantire lo sfruttamento adeguato del dispositivo l'impiego di un linguaggio di alto livello presenta numerosi vantaggi, non solo legati alla semplificazione dello sviluppo:

- un linguaggio di alto livello permette di ridurre drasticamente il tempo necessario per giungere a un primo risultato visualizzabile, semplificando la gestione del ciclo codifica-esecuzione-test;

- l'impiego di un compilatore permette di evitare i compiti più sensibili a errori, come ad esempio l'allocazione dei registri;
- uno shader sviluppato con l'impiego di un codice di alto livello è più semplice da comprendere e di conseguenza più semplice anche da modificare;
- inoltre l'impiego di un linguaggio di alto livello permette di ottenere una portabilità maggiore del codice sviluppato.

Il linguaggio Cg (*C for graphics*) è un linguaggio basato sul C, ma con miglioramenti e modifiche sintattiche volte a renderne più semplici sia lo sviluppo sia l'ottimizzazione del codice che si rivela fondamentale per poter competere a livello prestazionale con il codice sviluppato direttamente in codice nativo.

### 3.1.2 Language Profile

Cg permette di sviluppare sia *vertex programs* sia *fragment program*, tuttavia nel farlo deve tenere conto di una differenza sostanziale tra lo sviluppo del software che andrà eseguito su una CPU e il software sviluppato per una GPU.

Sostanzialmente mentre le CPU offrono un insieme di funzionalità di base comune a tutte le generazioni, eventualmente corredate da un insieme di funzionalità specifiche, le GPU offrono un insieme di istruzioni estremamente eterogenee, legate sia alla dualità ATI-NVIDIA presente sul mercato sia alla necessità di sviluppare prodotti per le diverse fasce di mercato e non ultima la necessità di ottenere le massime prestazioni possibili per soddisfare la necessità di realismo degli attuali software per l'intrattenimento. Cg risolve questo problema introducendo il concetto di *profile* inteso come un insieme di funzionalità offerte da un determinato insieme di dispositivi, dando quindi al compilatore tutte le informazioni necessarie per ottenere il miglior codice per il profilo disponibile andando ad impiegare, ad esempio, per ogni funzione, la versione nativa se disponibile nei dispositivi che supportano un determinato profilo o andando ad impiegare un versione software ottimizzata per il dato profile.



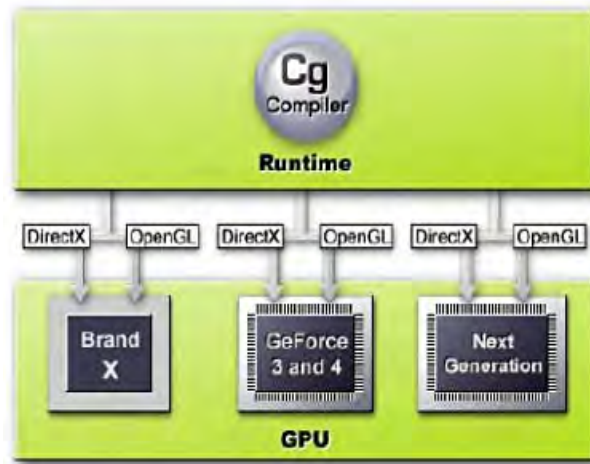


Figura 3.1: Architettura del runtime Cg

### 3.1.3 Input e Output

Le GPU operano su stream di dati, cioè sequenze di vertici da fornire in input al *vertex processor* e flussi di fragment per il *fragment processor*. Questa peculiarità unitamente al fatto che i programmi sviluppati vengono iterati su tutti gli elementi del flusso, presentando quindi un numero elevato di esecuzioni, ha portato il linguaggio Cg a definire una partizione all'interno dello spazio degli input, al fine di rendere accessibili le funzionalità di accelerazione hardware presenti sui dispositivi grafici. In particolare Cg distingue tra:

- input varianti;
- input uniformi.

#### Input varianti

I dati che vengono etichettati come varianti sono specifici per l'elemento a cui viene applicata la funzione codificata nel programma in esecuzione. Ad esempio tra gli input varianti per un programma eseguito su vertex processor avremo i valori specifici dei singoli vertici tra le quali sempre presenti saranno le coordinate spaziali del vertice. Per identificare un input come variante si impiega la sintassi:

```
<type> <name> : <ASSOCIAZIONE>
```

Dove ASSOCIAZIONE è uno delle seguenti semantiche associative:

L'output prodotto dal Vertex Program viene, dopo essere stato sottoposto al pro-

POSITION	BLENDWEIGHT
NORMAL	TANGENT
BINORMAL	PSIZE
BLENDINDICES	TEXCOORD0-TEXCOORD7

Tabella 3.1: Semantiche Associative

<b>float</b>	Floating point a 32 bit	<b>half</b>	Floating point a 16 bit
<b>int</b>	Intero a 32 bit	<b>fixed</b>	Fixed point a 12 bit nel formato (s1.10)
<b>bool</b>	Valore booleano	<b>string</b>	Una sequenza di caratteri

Tabella 3.2: Tipi di dato scalari

cesso di rastering, inviato come input al Fragment Program. Per ottenere la collaborazione tra i due programmi l'output del Vertex Program deve coincidere con l'input del Fragment Program ad esempio utilizzando la medesima struttura per la definizione dell'input e dell'output.

### Input uniformi

Gli input uniformi sono i valori che non variano al variare dell'elemento del flusso su cui il programma viene eseguito.

La dichiarazione di un input uniforme è effettuata attraverso la seguente sintassi:

```
uniform <type> <name>
```

## 3.2 Funzionalità del linguaggio

### 3.2.1 Tipi di dato

#### Tipi scalari

I tipi di dato supportati, esattamente come per la definizione delle tipologie di input, rispecchia direttamente le funzionalità offerte dai dispositivi hardware, in particolare sarà possibile impiegare variabili scalari con i seguenti tipi:

float4	float3	float2	float1
half4	half3	half2	half1
int4	int3	int2	int1
fixed4	fixed3	fixed2	fixed1
bool4	bool3	bool2	bool1

Tabella 3.3: Tipi di dato vettoriali

### Tipi vettoriali

Per ognuno dei tipi di dati presenti in 3.2 Cg mette a disposizione una serie di versioni vettoriali volte a modellare le unità vettoriali messe a disposizione dall'hardware. In particolare avremo:

### Tipi Matriciali

Inoltre vengono offerti dei tipi di dati per la modellazione di matrici di elementi. Per ogni tipo di dato di cui esistono versioni vettoriali esistono anche le versioni matriciali:

`<type>axb`

Dove  $a \in 1, 2, 3, 4$ ,  $b \in 1, 2, 3, 4$ . Un ulteriore tipo di dato che merita un approfondimento particolare sono i cosiddetti **sampler**.

Allo stato attuale né i vertex program né i fragment program posso impiegare un tipo di dato equivalente ai puntatori del linguaggio C. Questo costrutto richiede, come è noto, l'impiego di un tipo di dato appositamente definito e nativamente implementato in grado di fornire l'accesso alle zone di memoria, implementazione non presente sulle moderne GPU. I sampler, per quanto non offrano la flessibilità dell'equivalente C permettono di accedere in maniera diretta e computazionalmente efficace ai dati contenuti nelle texture. In particolare sono definiti sampler specifici per accedere alle diverse tipologie di texture nella forma:

`sampler<TEXTURETYPE>`

Dove TEXTURETYPE è la specifica del tipo di texture, in particolare sono supportate texture di tipo **1D,2D,3D,CUBE,RECT**.

### 3.2.2 Flusso di Controllo e operatori

#### Differenze del flusso di controllo rispetto al C

La gestione del flusso di controllo è ricollegabile direttamente a quella offerta dal linguaggio C con alcune varianti di cui è importante tener conto durante lo sviluppo del codice:

- l'operatore **switch** non è presente
- la ricorsione non è supportata
- alcuni profili non offrono la possibilità di effettuare *branch*.

#### Operatori Vettoriali

Mentre la gestione del flusso è un sottoinsieme delle possibilità offerte dal linguaggio C, gli operatori a disposizione sono arricchite attraverso l'introduzione del versioni vettoriali di tutti gli operatori matematici e logici, in particolare:

```
float4(a,b,c,d) * float4(A,B,C,D) = float4(a*A,b*B,c*C,d*D)
```

```
a * float4(A,B,C,D) = float4(a*A,b*B,c*C,d*D)
```

Tutte le operazioni effettuate su vettori sono eseguite simultaneamente in hardware su tutti gli elementi del vettore grazie all'impiego di tecnologie SIMD. L'accelerazione hardware è garantita anche per le operazioni sui tipi matriciali, benchè per questi sia necessario ricorrere all'operatore **mul**.

#### Operatore .

Un'operatore la cui semantica è stata espansa rispetto a quella C è l'operatore **.** che mantenendo la semantica propria del C quando applicata ad una struttura acquista anche la semantica di selezione tra i campi di un tipo vettoriale. In particolare

```
float3(a,b,c).xxzx = float4(a,a,c,a)
```

```
float4(a,b,c,d).wzyx = float4(d,c,b,a)
```

Inoltre la selezione di un elemento del vettore attraverso l'operatore permette anche di modificare un singolo valore del vettore, in particolare:

```
float3 vettore = float3(1,1,1);
```

```
vettore.x = 2; // vettore == float3(2,1,1);
```

Esecuzione di programmi GPU su CPU

## 3.3 Uso del linguaggio

### 3.3.1 Cg Runtime

I programmi Cg sono linee di codice che descrivono le operazioni da applicare agli stream di dati e necessitano del supporto da parte di un'applicazione per poter essere messi in esecuzione. Nello specifico per poter mandare in esecuzione un programma Cg è necessario svolgere due operazioni:

- compilare il programma per il profilo adatto per l'hardware disponibile a runtime;
- creare il collegamento tra l'applicazione e il programma Cg specificando i parametri necessari all'esecuzione.

Queste due operazioni possono essere svolte in fase di compilazione o in fase di esecuzione (*runtime*) attraverso l'impiego delle API offerte dalle librerie Cg.

L'operare in fase di esecuzione presenta tali vantaggi in termini di performance computazionali da giustificare la predominanza di questo approccio rispetto allo svolgere le operazioni in fase di compilazione, fase generalmente preferita in ambito di High Performance Computing, in particolare:

- la possibilità di effettuare la compilazione runtime permette di non dovere memorizzare una versione compilata per ogni profilo che si intende supportare;
- compilando runtime è possibile sfruttare le ottimizzazioni future relative al profilo scelto;
- la possibilità di sfruttare profile non ancora creati permette di garantire il supporto anche per ottimizzazioni hardware future;

### 3.3.2 Inizializzazione dell'ambiente

Ad ogni applicazione che impiega Cg viene garantita la possibilità di costruire uno o più contesti Cg che vengono impiegati come contenitori per i programmi che si intende eseguire.

Attraverso la chiamata:

```
CGcontext context = cgCreateContext();
```

vengono allocate le risorse necessarie per la creazione di programmi Cg, che possono essere compilati attraverso l'impiego della funzione

```
CGprogram program = cgCreateProgram(context,  
                                     CG_SOURCE,  
                                     codicesorgente,  
                                     profile,  
                                     ''nomeprogramma'',  
                                     argomenti);
```

Attraverso il secondo parametro è possibile alterare la semantica del terzo parametro indicando ad esempio che esso contiene il codice dello shader come stringa di testo (CG\_SOURCE) o come codice precompilato (CG\_OBJECT).

Il programma, sia esso sotto forma di codice sorgente o binario già compilato può essere fornito al runtime anche sotto forma di file esterno attraverso la funzione

```
CGprogram program = cgCreateProgramFromFile(context,  
                                             CG_SOURCE,  
                                             codicesorgente,  
                                             profile,  
                                             ''nomeprogramma'',  
                                             argomenti);
```

Dopo aver ottenuto un programma valido è possibile procedere al suo caricamento sulla GPU

### 3.3.3 Caricamento del programma

Il caricamento sull'unità grafica risulta estremamente semplice grazie alla scelta di operare con il linguaggio Cg: in particolare dato un programma valido è possibile caricarlo attraverso l'uso della chiamata

```
cgGLLoadProgram (program);
```

In generale in fase di caricamento del programma vengono anche preparati gli *handle* per la gestione dei parametri del codice dello shader, ad esempio dato il seguente codice sorgente Cg:

```
float3 cos (
    float2 coords : TEXCOORD0,
    uniform samplerRECT textureY
) : COLOR
{
    float3 y = texRECT(textureY,coords);
    return cos(y);
}
```

sarà possibile ottenere un riferimento per la manipolazione del secondo argomento attraverso l'invocazione della funzione:

```
CGparameter yParam = cgGetNamedParameter (fragmentProgram,"textureY");
```

Vedremo come l'impiego di questo handle in fase di esecuzione permette di controllare i parametri con cui viene invocato il programma Cg. Dopo aver preparato il programma è necessario impostare il driver OpenGL affinché le operazioni di rendering abbiano come destinazione delle zone di memoria (*texture*) anziché direttamente il framebuffer video[4]. Ciò avviene mediante la creazione di un *FrameBuffer Object*, che fornito come estensione OpenGL dalla maggior parte dei dispositivi grafici è attivabile con il seguente blocco di codice:

```
GLuint fb;
glGenFramebuffersEXT(1, &fb);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
```

### 3.3.4 Preparazione dell'input e dell'output

L'input viene passato ad un programma Cg sotto forma di texture, la cui creazione può essere effettuata impiegando il seguente blocco di codice:

```
glEnable (GL_TEXTURE_RECTANGLE_ARB);
glGenTextures (1, &input);
glBindTexture(GL_TEXTURE_RECTANGLE_ARB,input);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
    GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,
```

```
        GL_TEXTURE_MAG_FILTER, GL_NEAREST);  
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,  
        GL_TEXTURE_WRAP_S, GL_CLAMP);  
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,  
        GL_TEXTURE_WRAP_T, GL_CLAMP);  
glTexImage2D(GL_TEXTURE_RECTANGLE_ARB, 0,  
        GL_RGB32F_ARB, w, h, 0, GL_RGB,  
        GL_FLOAT, 0);
```

Questo codice è direttamente mappabile con il codice impiegato normalmente per la creazione di texture da parte delle applicazioni che usano le librerie OpenGL, in particolare l'uso dei parametri *GL\_NEAREST* e *GL\_CLAMP* garantisce che l'accesso alla texture non venga alterato da politiche particolari di wrapping o di scalatura. Un differenza rispetto al classico codice OpenGL è data dalla presenza del target *GL\_TEXTURE\_RECTANGLE\_ARB* che permette di impiegare anche texture non quadrate.

Attraverso il medesimo blocco di codice presentato andremo a costruire una texture che impiegheremo per memorizzare l'output del programma.

### 3.3.5 Upload dell'input

L'upload dell'input avviene in maniera analoga a quanto avviene per l'upload di texture, attraverso la funzione

```
glTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, 0, 0, 0,  
        src_w, src_h, GL_RGB, GL_FLOAT, data);
```

### 3.3.6 Esecuzione

L'impiego di proiezioni visive, da definire necessariamente per una corretta esecuzione di programmi, deve essere tale da garantire che ogni pixel della texture in input venga preso in considerazione; per fare ciò è sufficiente richiedere al driver opengl di impiegare la seguente proiezione:

```
glMatrixMode(GL_PROJECTION);  
glLoadIdentity();
```



```
gluOrtho2D(0.0, w, 0.0, h);  
glMatrixMode(GL_MODELVIEW);  
glLoadIdentity();  
glViewport(0, 0, w, h);
```

La fase di esecuzione prevede una serie di semplici passaggi:

1. associazione tra FrameBuffer Object e texture di output;
2. specifica dell'FBO di destinazione;
3. specifica della texture da impiegare (input);
4. attivazione del programma;
5. specifica degli eventuali parametri;
6. attivazione dei parametri;
7. disegno di un poligono che garantisca la presa in considerazione dell'input nella sua interezza.

Questi passi possono essere svolti impiegando il seguente blocco di codice:

```
glFramebufferTexture2D(GL_FRAMEBUFFER_EXT,  
                        GL_COLOR_ATTACHMENT0_EXT,  
                        GL_TEXTURE_RECTANGLE_ARB,  
                        output, 0);           // Step 1  
glDrawBuffer (GL_COLOR_ATTACHMENT0_EXT);    // Step 2  
glBindTexture(GL_TEXTURE_RECTANGLE_ARB, input); // Step 3  
cgGLBindProgram(program);                   // Step 4  
cgGLSetTextureParameter(yParam, input);    // Step 5  
cgGLEnableTextureParameter(yParam);        // Step 6  
glBegin(GL_QUADS);                          // Step 7  
glTexCoord2f(0.0, 0.0);  
glVertex2f(0,0);  
glTexCoord2f(src_w, 0.0);  
glVertex2f(dst_w, 0);
```

```
glTexCoord2f(src_w, src_h);  
glVertex2f(dst_w, dst_h);  
glTexCoord2f(0.0, src_h);  
glVertex2f(0, dst_h);  
glEnd();
```

### 3.3.7 Download dell'output

Al termine del rendering della scena è possibile recuperare i risultati ottenuti specificando il FrameBuffer Object da cui si intende leggere ed effettuando la lettura vera e propria. Il codice impiegato per svolgere questa operazione, che rappresenta un forte collo di bottiglia per l'esecuzione di un programma GPU è il seguente:

```
glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);  
glReadPixels(0, 0, w, h, GL_RGB, GL_FLOAT, output);
```

dove output è un array opportuno.

# Capitolo 4

## Prestazioni Elementari

*Anyone can build a fast CPU. The trick is to build a fast system.  
Seymour Cray (on the importance of memory, bandwidth and throughput.)*

### 4.1 Costi di caricamento e fattori di convenienza

Il modello computazionale alla base delle Graphics Processing Unit è, come abbiamo visto, sostanzialmente differente dal modello tradizionalmente implementato nelle CPU. E' necessario quindi andare ad analizzare con un approccio quantitativo i costi computazionali legati alle operazioni elementari.

Questo approccio, tipico del calcolo ad alte prestazioni, è stato applicato solo in minima parte ai lavori svolti sulle GPU, dove si è preferita una valutazione strettamente collegata agli algoritmi implementanti, basandosi su osservazioni, che per quanto valide non possono prevaricare i risultati ottenuti attraverso l'esperienza diretta. Ogni algoritmo che si voglia implementare al fine di sfruttare le potenzialità offerte dalle GPU deve essere suddiviso in quattro macro-fasi:

1. caricamento del programma;
2. upload degli input;
3. esecuzione del programma;
4. download degli output.

In generale si deve risolvere un problema attraverso l'applicazione di un algoritmo fissato a più istanze in input ammortizzando quindi i costi di caricamento del pro-



Figura 4.1: Ciclo di Esecuzione

gramma, già di per sè ridotti.

Il collo di bottiglia evidente, e presente anche in letteratura [15][9], è costituito dall'upload e dal download dei dati, analogamente per quanto accade per il calcolo su CPU dove il collo di bottiglia è rappresentato dai costi di accesso alla memoria.

Nel caso delle GPU questo rallentamento è evidenziato dalla non-integrità strutturale della scheda video, che quindi necessita di un BUS di I/O esterno. Negli ultimi anni si è operato molto a livello hardware per ridurre questo collo di bottiglia, introducendo il bus PCI-Express.

#### 4.1.1 PCI Express e caratteristiche hardware

Il PCI Express, indicato anche con PCIe, è un'implementazione del bus PCI, che basandosi sui concetti elementari alla base del bus precedente, introduce un diverso livello di comunicazione non più basato sul concetto di bus, ma su un'architettura a stella dove le singole componenti comunicano in maniera seriale. Quest'architettura oltre a offrire una maggiore velocità di comunicazione richiede minimi cambiamenti alle periferiche che possono essere quindi facilmente adattate per poter operare sul PCI Express.

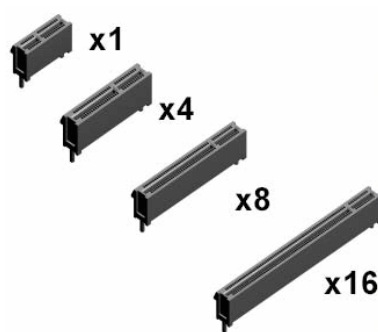


Figura 4.2: Dettaglio dei connettori PCI Express

Diverso è il caso delle schede video, che per offrire pieno supporto al bus PCI Express hanno richiesto drastici cambiamenti in termini di logica hardware per gestire il diverso protocollo di comunicazione; per completezza ricordiamo infatti che prima del PCIe il bus privilegiato per le schede video era il bus AGP, sostanzialmente differenze dal bus PCI delle altre periferiche. Basandosi su un trasferimento dei dati seriale, a differenza di quello parallelo del bus PCI, si semplifica il layout del PCB delle schede madri. Ciò consente una notevole modularità, in quanto possono essere aggregati più canali per aumentare la banda passante disponibile o per supportare particolari configurazioni come l'utilizzo di due o più schede video. Inoltre la bandwidth di ciascun canale è indipendente da quella degli altri. Un singolo canale PCI-ex, chiamato x1, offre una bandwidth full duplex di 266 MBytes/sec. I connettori PCI-ex per schede video sono generalmente costituiti da 16 canali, dunque 16x, offrendo una velocità doppia rispetto allo standard AGP 8x.

I primi chipset in commercio offrivano un massimo di 20 canali, pertanto nel caso di un utilizzo simultaneo di due schede grafiche, come è possibile attraverso l'impiego della tecnologia sviluppata da Nvidia nota con l'acronimo SLI (Scalable Link Interface), i canali si riducono a 8x per ciascuna scheda, senza tuttavia evidenziare decadimenti prestazionali.

Nelle ultime incarnazioni del chipset Nvidia SLI-16X i canali sono stati praticamente raddoppiati, in attesa che i futuri applicativi CAD o ludici possano trarne giovamento.

PCI Express è infine progettato per sostenere il sempre maggior fabbisogno energetico delle schede video di ultima generazione. Infatti, a differenza dello slot AGP, in grado di erogare un massimo di 50 Watt, l'attuale revisione di PCI-ex supporta

carichi fino a 75W, permettendo così di eliminare il connettore MOLEX di alimentazione da molte schede.

Confrontando le prestazioni nella tabella 4.1 di alcuni dei più noti bus impiegati nel corso dell'evoluzione dei personal computer emerge immediatamente come il PCI Express compete direttamente con il bus HyperTransport che richiede tuttavia un enorme sforzo in fase di progettazione delle schede madri in quanto essendo un bus parallelo richiede un numero molto maggiore di piste.

Attualmente HyperTransport e PCI Express non vengono considerate tecnologie concorrenti ma, data la diversa natura dei dispositivi che connettono, tecnologie complementari.

## 4.2 Valutazione degli operatori elementari

Nella recente letteratura scientifica [15][9][14][3] riguardante l'uso delle Graphics Processor Unit per la risoluzione di calcoli non direttamente correlati alla visualizzazione di scene 3D, si parte da un'assioma secondo il quale per ottenere dei benefici dall'applicazione di procedure scritte per GPU è necessario che l'algoritmo sia aritmeticamente intenso, cioè deve essere tale per cui ogni word caricata dalla memoria principale sarà impiegata in un alto numero di operazioni.

In generale si preferisce [12] implementare su GPU algoritmi con una forte intensità aritmetica, cioè con un alto rapporto  $\frac{flops}{word \text{ in input}}$ , al fine di ottenere i migliori benefici computazionali.

Benché a livello generale questa assunzione risulti vera, cioè le applicazioni che implementano algoritmi che soddisfano la proprietà di intensità aritmetica si prestano bene a essere implementate su GPU, la superiorità tecnologia con cui sono implementate le micro-operazioni è tale da compensare, oltre una certa quantità di dati, i costi di caricamento e scaricamento.

Per questo abbiamo deciso di implementare e valutare le prestazioni di alcune funzioni elementari di grandissimo utilizzo, in particolare dato il vettore:

$$a = (a_1, a_2, a_3, \dots, a_n)$$

definendo

$$.op(a) = (op(a_1), op(a_2), op(a_3), \dots, op(a_n))$$

Bus	Bit/s	Byte/s
ISA 08-Bit/4.77Mhz	38.66 Mbit/s	4.83 MB/s
ISA 16-Bit/8.33Mhz	134.66 Mbit/s	16.85 MB/s
PCI 32-bit/33Mhz	1066.66 Mbit/s	133.33 MB/s
PCI Express (x1 link)	2500 Mbit/s	250 MB/s
PCI 64-bit/33MHz	2133.33 Mbit/s	266.66 MB/s
PCI 32-bit/66MHz	2133.33 Mbit/s	266.66 MB/s
AGP 1x	2133.33 Mbit/s	266.66 MB/s
AGP 2x	4266.66 Mbit/s	533.33 MB/s
PCI 64-bit/66MHz	4266.66 Mbit/s	533.33 MB/s
PCI-X DDR 16-bit	4266.66Mbit/s	533.33 MB/s
PCI Express (x4 link)	10000 Mbit/s	1000 MB/s
AGP 4x	8533.33 Mbit/s	1066.66 MB/s
PCI-X 133	8533.33 Mbit/s	1066.66 MB/s
PCI-X QDR 16-bit	8533.33 Mbit/s	1066.66 MB/s
InfiniBand	10.00 Gbit/s	1.25 GB/s
PCI Express (x8 link)	20.00 Gbit/s	2.0 GB/s
AGP 8x	17.066 Gbit/s	2.133 GB/s
PCI-X DDR	17.066 Gbit/s	2.133 GB/s
PCI Express (x16 link)	40.0 Gbit/s	4.0 GB/s
PCI-X QDR	34.133 Gbit/s	4.266 GB/s
HyperTransport (800MHz, 16-pair)	51.2 Gbit/s	6.4 GB/s
HyperTransport (1GHz, 16-pair)	64.0 Gbit/s	8.0 GB/s

Tabella 4.1: Prestazioni dei più noti BUS

valuteremo quindi le prestazioni di:

`. = (a)`

`.cos(a)`

`.sin(a)`

### 4.2.1 Shader

Nel capitolo precedente abbiamo presentato il linguaggio Cg che è stato scelto per la realizzazione del codice da impiegare, adesso mostriamo il codice impiegato per il calcolo delle funzioni scelte come metro di valutazione:

```
float3 copy (  
    float2 coords : TEXCOORD0,  
    uniform samplerRECT textureY  
    ) : COLOR  
{  
float3 y = texRECT(textureY,coords);  
    return y;  
}
```

```
float3 cos (  
    float2 coords : TEXCOORD0,  
    uniform samplerRECT textureY  
    ) : COLOR  
{  
float3 y = texRECT(textureY,coords);  
    return cos(y);  
}
```

```
float3 sin (  
    float2 coords : TEXCOORD0,  
    uniform samplerRECT textureY  
    ) : COLOR  
{
```



```
float3 y = texRECT(textureY,coords);  
    return sin(y);  
}
```

Il codice impiegato per il caricamento del programma Cg è il seguente:

```
void initCG(void) {  
    // set up Cg  
    cgSetErrorCallback(cgErrorCallback);  
    // Create context  
    cgContext = cgCreateContext();  
    // Best profile  
    fragmentProfile = cgGLGetLatestProfile(CG_GL_FRAGMENT);  
    cgGLSetOptimalOptions(fragmentProfile);  
  
    // create fragment program  
    fragmentProgram = cgCreateProgramFromFile(  
        cgContext, CG_SOURCE, "copy.cg",  
        fragmentProfile, "copy", 0);  
    // load program  
    cgGLLoadProgram (fragmentProgram);  
    // and get parameter handles by name  
    yParam = cgGetNamedParameter (fragmentProgram,"textureY");  
}
```

Mentre per l'esecuzione dei calcoli si è impiegato il seguente segmento di codice:

```
void drawScene(int src_w, int src_h, int dst_w,int dst_h)  
{  
    glFramebufferTexture2D(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,  
        GL_TEXTURE_RECTANGLE_ARB, output, 0);  
    //glFramebufferTexture2D(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT1_EXT,  
        GL_TEXTURE_RECTANGLE_ARB, input, 0);  
    glDrawBuffer (GL_COLOR_ATTACHMENT0_EXT);  
    glBindTexture(GL_TEXTURE_RECTANGLE_ARB, input);  
}
```

```
    glPolygonMode(GL_FRONT, GL_FILL);

// enable fragment profile
    cgGLEnableProfile(fragmentProfile);
// bind saxpy program
    cgGLBindProgram(fragmentProgram);
    cgGLSetTextureParameter(yParam, input);
    cgGLEnableTextureParameter(yParam);

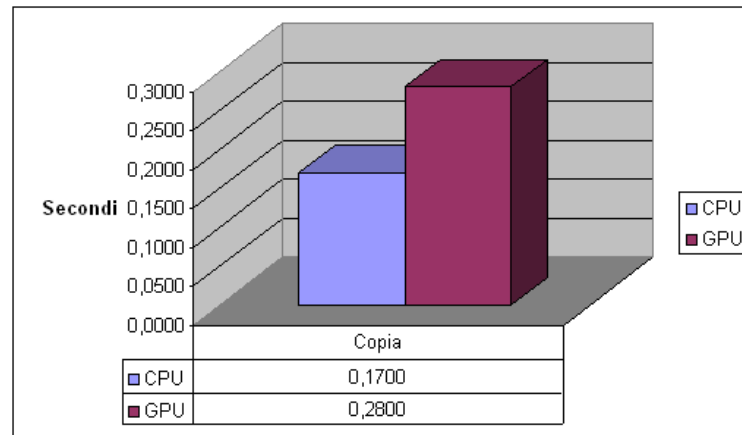
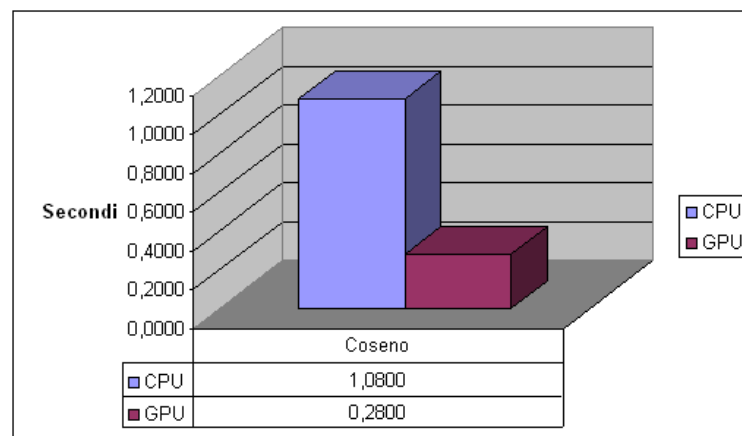
    glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0);
    glVertex2f(0,0);
    glTexCoord2f(src_w, 0.0);
    glVertex2f(dst_w, 0);
    glTexCoord2f(src_w, src_h);
    glVertex2f(dst_w, dst_h);
    glTexCoord2f(0.0, src_h);
    glVertex2f(0, dst_h);
    glEnd();

}
```

### 4.2.2 Risultati

Andando ad analizzare i risultati evidenziati in tabella emerge immediatamente come, nel caso i flops per word siano pari a 0 (come nel caso dell'operatore  $.$  =) l'osservazione sulla necessità di un'intensità aritmetica sembra confermata, tuttavia non appena si applica una semplice operazione (implementata hardware su entrambi i dispositivi) l'osservazione viene smentita, o meglio confermata solo per piccole istanze di problemi, eliminando quindi il vincolo sulla tipologia dell'algoritmo (intensità aritmetica) per essere sostituita da una considerazione che tenga conto anche della dimensionalità del problema.

I test sono stati compiuti operando su texture di dimensioni pari a 2048x2048 pixel con una 1 float per ogni componente RGBA del pixel, per un totale di 16777216


Figura 4.3: Confronto per `.=`

Figura 4.4: Confronto per `.cos`

floats per un totale di 64 MB.

Osservando la figura 4.3 si può notare come il costo necessario per round trip dei dati sia superiore, come atteso, nel caso si operi su GPU. Questo risultato, facilmente ottenibile da un'attenta analisi dell'architettura, può rappresentare una metrica di confronto interessante per l'analisi di figura 4.4 e 4.5.

Nel caso venga applicata una semplice operazione, implementata in hardware in entrambi i dispositivi, la superiorità computazione della GPU risalta immediatamente: il lavoro parallelo dello 24 unità di calcolo unitamente ad un approccio moderno al design di quest'ultime copre abbondantemente i costi necessari da sopportare per la movimentazione dei dati.

Con questo risultato abbiamo dimostrato come la superiorità tecnologia delle unità di calcolo parallele e il massivo parallelismo offerto possono ridurre i requisiti struttu-

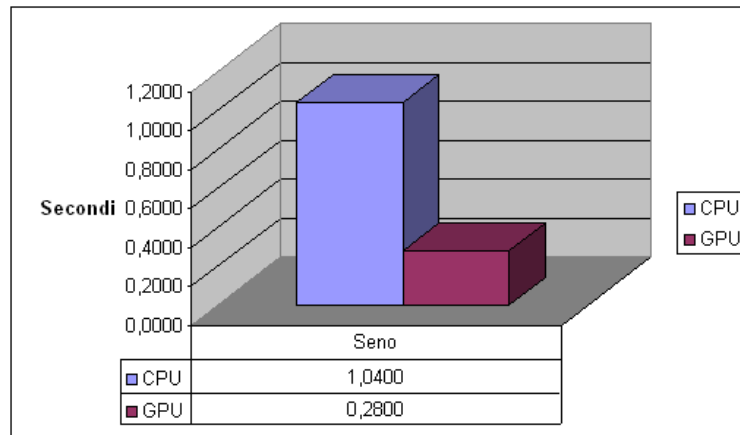


Figura 4.5: Confronto per .sin

rali necessari per ottenere incrementi prestazionali, riducendo l'intensità aritmetica strutturale richiesta.

Il risultato ottenuto attraverso la valutazione che ha portato alla realizzazione del grafico in 4.3 permette di scorporare dai risultati nelle figure 4.4,4.5 il tempo impiegato per la movimentazione dei dati, ottenendo un confronto diretto sulle potenzialità di calcolo.

Questa operazione permette di avere una linea guida per la valutazione dell'opportunità di implementare un determinato algoritmo su architettura GPU. Infatti il costo legato alla movimentazione dei dati rappresenta un costo sicuramente da considerare ma legato esclusivamente alla mole delle informazioni su cui viene applicato l'algoritmo, ma assolutamente non correlato alla complessità computazionale dell'algoritmo implementato. Si ha quindi che il costo computazionale di una procedura implementata su scheda video può essere decomposto in:

$$\text{Costo Computazione} = \text{Costi di Gestione} + \text{Esecuzione}$$

Naturalmente a seconda della tipologia di algoritmo i costi di gestione possono essere analizzata con un livello maggiore di dettaglio, in particolare:

$$\text{Costi di Gestione} = \text{Inizializzazione} + \text{Upload} + \text{Download}$$

Nei grafici 4.6,4.7 emerge direttamente la potenza computazionale del dispositivo. Il costo computazione passa, nel caso del coseno da 0.92 secondi a 0.01 secondi con un incremento delle prestazione di 92 volte. Per completezza ricordiamo che in tutti

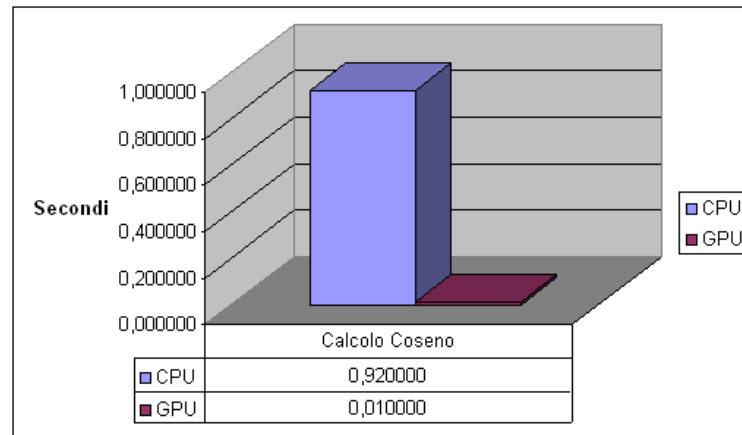


Figura 4.6: Costo di esecuzione per .cos

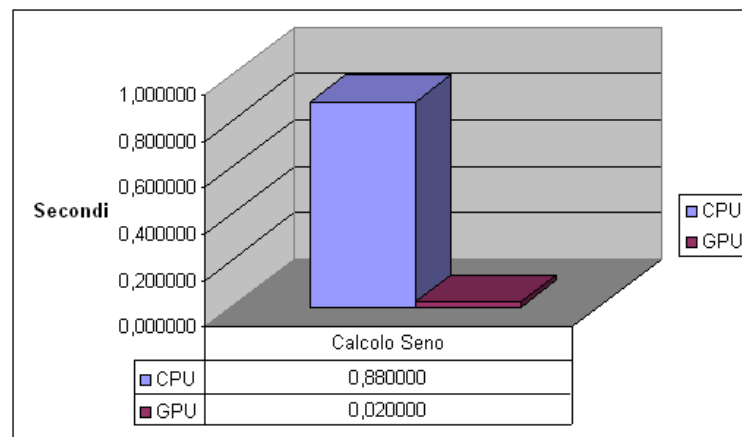


Figura 4.7: Costo di esecuzione per .sin

i dispositivi grafici, le funzioni seno e coseno vengono fornite anche attraverso l'istruzione sincos che calcola in parallelo i due valori senza alcun overhead.

Ciò offrirebbe la possibilità di modificare il codice presentato per ottenere un codice che con un tempo di esecuzione ipotizzato su gpu di 0.02 ( $\max[0.01; 0.02]$ ) secondi sia in grado di ottenere i medesimi risultati di un algoritmo seriale su cpu con un tempo di calcolo pari 1.8 secondi ( $0.92 + 0.88$ ) ottenendo un incremento superiore a 90 volte.

Tuttavia una simile considerazione sarebbe fuorviante in quanto una simile funzionalità è offerta solo per gli operatori presi in esame, senza quindi poter essere utilizzate come strumento di valutazione attendibile.



# Capitolo 5

## Scalatura di Immagini

*All exact science is dominated  
by the idea of approximation  
Bertrand Russel*

### 5.1 Aspetti teorici

#### 5.1.1 Campionamento e Quantizzazione

Nella creazione di un'immagine digitale ci si trova, in generale, di fronte alla necessità di convertire i dati raccolti in forma continua dal sensore in forma digitale, al fine di poter procedere all'elaborazione degli stessi mediante un calcolatore digitale.

Durante il processo di creazione di un'immagine digitale a partire da un'immagine analogica ci si trova a dover affrontare due problemi di discretizzazione distinti:

- Il Campionamento, cioè la discretizzazione dei valori delle coordinate continue;
- La Discretizzazione, cioè la discretizzazione dell'intensità percepita dal sensore nelle coordinate in esame;

Il passaggio dal continuo al discreto introduce inevitabilmente degli errori legati alle diverse potenzialità informative dei due insiemi, nonché alla metodologia seguita per la fase di acquisizione dell'immagine e alla granularità dello spazio di destinazione, problemi affrontati ampiamente in letteratura.[5]

### 5.1.2 Scalatura di immagini digitali

Il processo di ridimensionamento di un'immagine è strettamente legato ai concetti di campionamento e quantizzazione non appena si identifica il nesso tra l'operazione di ingrandimento di un'immagine con l'operazione di sovracampionamento e l'operazione di riduzione di un'immagine con il sottocampionamento dell'immagine stessa. La differenza caratteristica tra le due famiglie di operazioni è la sorgente dell'informazione: nel caso delle operazioni di campionamento ci troveremo a acquisire le informazioni da una sorgente analogica, mentre nel caso delle operazioni di ridimensionamento di un'immagine la sorgente dell'informazione è memorizzata in formato digitale. Il processo di ingrandimento può essere suddiviso in due fasi distinte: una fase di allocazione dei pixel di destinazione in una zona di memoria opportunamente dimensionata e la successiva determinazione dei valori dei pixel allocati. Supponendo di voler ridimensionare con un fattore pari a 1.5x un'immagine quadrata di 500 pixel di lato dovremmo allocare una zona di memoria in grado di ospitare  $750^2$  pixel. La seconda fase può essere concettualmente visualizzata come la sovrapposizione di una griglia di dimensioni pari all'immagine originale, ma con un'opportuna (750x750) griglia di pixel. Nel griglia sussisteranno dei pixel non completamente contenuti in un pixel della griglia originaria, in questo caso bisogna scegliere come identificare il valore da assegnare al pixel.

Vediamo ora i principali metodi per la scelta dell'intensità luminosa.

## 5.2 Interpolazione Nearest Neighbour

L'interpolazione Nearest Neighbour determina l'intensità luminosa delle componenti del pixel scegliendo i valori del pixel noto più vicino al pixel in esame. Dopo l'allocazione della zona di memoria che conterrà l'immagine di destinazione è possibile procedere all'applicazione del metodo Nearest Neighbour che prevede una scansione di tutti i pixel dell'immagine di output attraverso l'iterazione sulle coordinate  $(x, y)$  dei pixel di cui si vuole calcolare l'intensità luminosa. Naturalmente nel caso RGB si procede considerando l'immagine a colori come la sovrapposizione 3 di immagini monocromatiche, operando quindi singolarmente su ogni componente.

Disponendo del fattore di scala  $s$  applicato all'immagine originaria per aumentarne ( $s > 1$ ) o diminuirne ( $s < 1$ ) le dimensioni è possibile, date le coordinate  $(x_i, y_i)$ , sta-



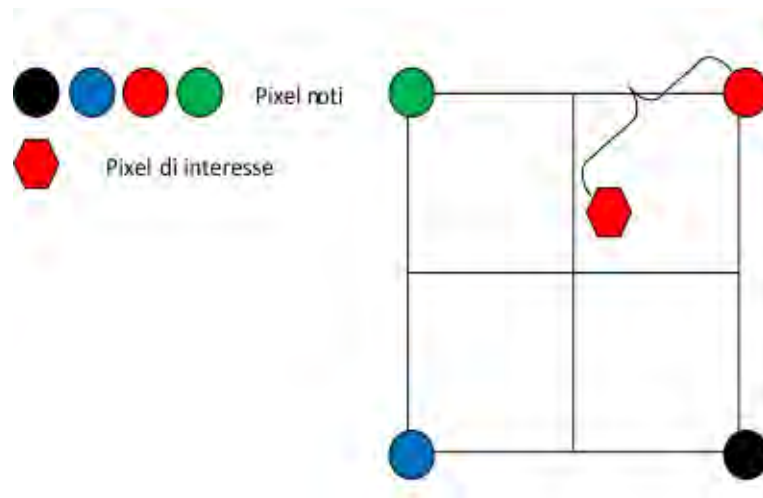


Figura 5.1: Interpolazione Nearest Neighbour

bilire le coordinate corrispondenti nell'immagine di partenza  $(X_i, Y_i)$  dove  $X_i = x_i \frac{1}{s}$  e  $Y_i = y_i \frac{1}{s}$ , come risulta evidente dall'ispezione delle formule precedenti la mappatura delle coordinate calcola coordinate reali, la cui parte non intera viene impiegata, dall'algoritmo Nearest Neighbour, per identifica i 3 pixel la cui distanza dal pixel preso considerazione è minima.

Il valore di tale pixel verrà poi utilizzato come valore del pixel nell'immagine che si sta creando. Nearest Neighbour è universalmente riconosciuto come uno dei metodi di scalatura più performanti: basti considerare l'esiguo numero di operazioni necessarie per stabilire il valore di un pixel. Oltre alle prestazioni computazioni bisogna notare come benchè l'immagine scalata possa presentare evidenti imprecisioni visuali l'algoritmo preserva i colori originari dell'immagine, in quanto la mappatura è solo spaziale e non varia l'insieme delle intensità dei pixel presenti nell'immagine.

### 5.3 Interpolazione Bilineare

L'interpolazione bilineare è un'estensione dell'interpolazione Nearest Neighbour, che interpola il valore di una funzione a una variabile come una media pesata rispetto alla distanza dei valori noti, adattata per operare su funzioni a due variabili e per analogia sulle immagini digitali. Intuitivamente il metodo bilineare si basa sull'applicazione del metodo lineare rispetto alle due direzione. Supponiamo di essere interessati al valore della funzione  $f$  nel punto  $P = (x, y)$  assumendo di conoscere il valore di  $f$

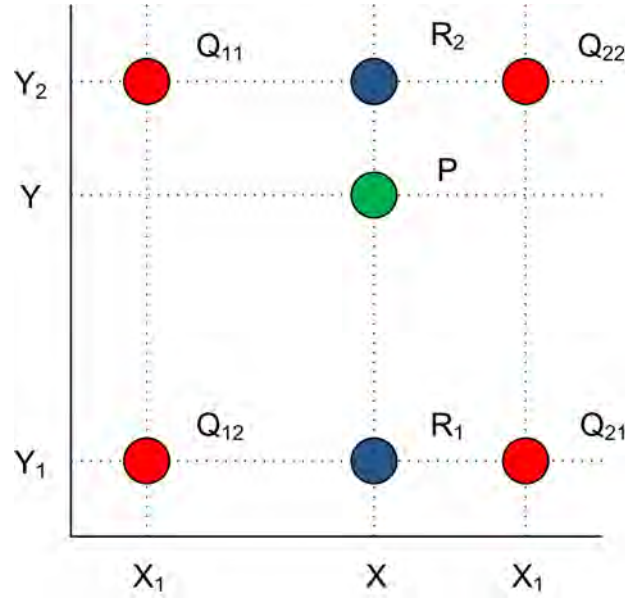


Figura 5.2: Interpolazione Bilineare

nei quattro punti  $Q_{11} = (x_1, y_1)$ ,  $Q_{12} = (x_1, y_2)$ ,  $Q_{21} = (x_2, y_1)$ ,  $Q_{22} = (x_2, y_2)$ . Il primo passo che compiamo è l'interpolazione nella direzione x, ottenendo:

$$f(R_1) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21})$$

$$f(R_2) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22})$$

Dove  $R_1 = (x, y_1)$ ,  $R_2 = (x, y_2)$ . Permettendoci di ottenere:

$$\begin{aligned} f(x, y) \approx & \frac{f(Q_{11})}{(x_2 - x_1)(y_2 - y_1)} (x_2 - x)(y_2 - y) + \\ & \frac{f(Q_{21})}{(x_2 - x_1)(y_2 - y_1)} (x - x_1)(y_2 - y) + \\ & \frac{f(Q_{12})}{(x_2 - x_1)(y_2 - y_1)} (x_2 - x)(y - y_1) + \\ & \frac{f(Q_{22})}{(x_2 - x_1)(y_2 - y_1)} (x - x_1)(y - y_1) \end{aligned}$$

Il metodo bilineare presenta un comportamento migliore a livello grafico se confrontato con il metodo Nearest Neighbour, risultando tuttavia computazionalmente più costoso.



Figura 5.3: Interpolazione Bilineare con fattori di scala 16, 8, 4 e l'immagine originaria, successivamente riscalate con il filtro di Lanczos, per ottenere le medesime dimensioni.

## 5.4 Aspetti implementativi

Una prima implementazione dei metodi di interpolazione appena presentati è, nel caso di codice sviluppato per un'architettura classica, ottenibile attraverso la trasposizione della formulazione matematica in nel linguaggio di programmazione C. A questa implementazione è possibile affiancare tutte le ottimizzazioni applicabili nella realizzazione di un software per un processore moderno come ad esempio l'uso di istruzioni SIMD, l'impiego di politiche avanzate per la gestione della memoria cache e anche la distribuzione del carico di lavoro su più unità di calcolo attraverso l'uso della programmazione multithreading e distribuita. In generale per un confronto corretto sarebbe necessario implementare tutte le possibili ottimizzazioni sul codice CPU per poterlo confrontare con il codice GPU, tuttavia è obiettivo di questo lavoro di tesi dimostrare come sia possibile considerare l'esecuzione su Graphics Processing Unit, non come l'impiego di un'unità computazionale interna aggiuntiva, ma come un'unità di calcolo indipendente. Come nel caso del calcolo distribuito non ha senso prendere come metro di paragone una versione ottimizzata con le tecnologie SIMD in quanto queste ottimizzazioni possono essere applicate anche nel calcolo distribuito. Impiegheremo quindi l'implementazione presente in letteratura prestando attenzione esclusivamente alle ottimizzazioni a livello di codice sorgente senza andare ad alterare la struttura del codice.

### 5.4.1 Filtri per il ridimensionamento texture

Abbiamo visto come implementeremo gli algoritmi per la riduzione delle immagini su architettura classica, nel caso dell'impiego di una Graphics Processing Unit è possibile seguire sostanzialmente due strade. La prima strada prevede la realizzazione di uno shader program che applicato a ogni pixel dell'immagine di destinazione stabilisce se il pixel deve essere attiva e nel qual caso il valore da attribuire al pixel. Un'altro metodo, che viene scelto in questo lavoro data la totale assenza di riferimenti a esso in letteratura, attraverso l'impiego esclusivo delle librerie opengl, presentando quindi la massima portabilità tra schede hardware e sistemi operativi differenti. Sostanzialmente questo approccio è applicabile conoscendo approfonditamente le tecniche impiegate nei videogiochi per l'adattamento delle texture e le opportunità computazionali offerte dalle schede sotto forma di un'opportuna accele-

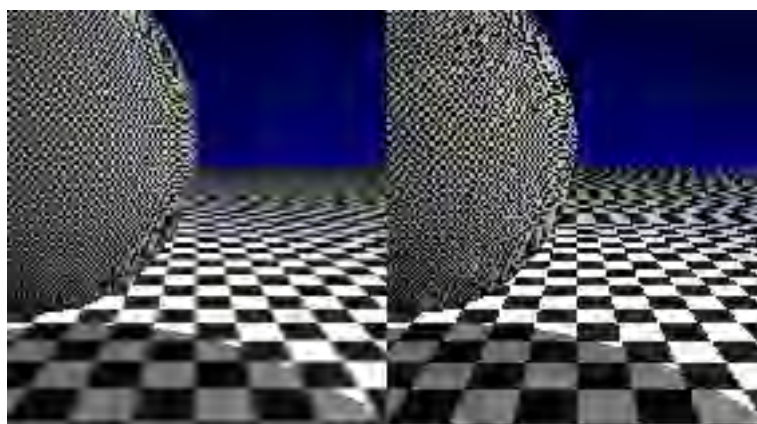


Figura 5.4: MipMapping (sinistra) e Nearest Neighbour (destra)

razione hardware. Vediamo nel dettaglio di quali operazioni hardware disponiamo su un dispositivo andando a presentare le operazioni grafiche legate alle texture.

### 5.4.2 Texture Mapping

Ogni primitiva geometrica viene disegnata, senza considerare l'applicazione di texture, o con un colore tinta unita o con un colore sfumato tra i vertice. Questo approccio è fortemente limitativo. Supponiamo ad esempio di voler realizzare un muro di mattoni senza utilizzare le texture; in questo caso saremmo costretti a disegnare ogni singolo mattone come un poligono distinto. Nel caso di un muro di grandi dimensioni questo approccio può richiedere migliaia di mattoni, che oltre a rappresentare un costo computazionale enorme, risulteranno troppo regolari per risultare realistici. Il texture mapping permette di applicare l'immagine di un muro mattoni, realizzata digitalmente o acquisita da un muro reale, su un singolo poligono, andando a ottenere un'aspetto più realistico. Sempre attraverso l'impiego del texture mapping si ottiene la coerenza della scena alle diverse posizioni assumibili dall'osservatore realizzando tutti gli effetti propri della visione in prospettiva. Le texture sono di forma quadrata o rettangolare, ma dopo essere state applicate a un poligono o a una generica superficie e trasformate per aderire al punto di vista di cui si effettua il rendering, le componenti della texture sono raramente mappabili uno a uno con i pixel dell'immagine di destinazione. Le texture sono semplicemente array rettangolari che memorizzano dati riguardanti la luminosità o il colore sia con l'esplicitazione del canale alpha che in forma tradizionale. Gli elementi singoli che

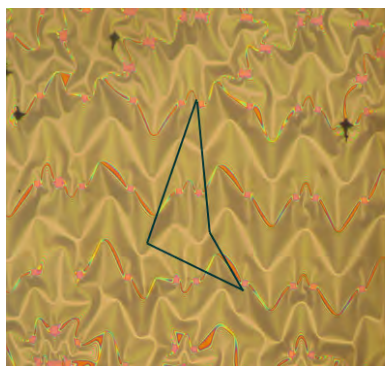


Figura 5.5: Processo del texture mapping.

costituiscono la texture sono noti con il nome *texels*.

Osservando la figura è possibile identificare i due passaggi logici che portano al texture mapping. La parte sinistra dell'immagine rappresenta la texture, mentre il poligono disegnato con il bordo nero rappresenta come i vertici del quadrilatero vengono mappati con specifici punti della texture. Quando il quadrilatero viene visualizzato sullo schermo esso può venire distorto dall'applicazione delle trasformazioni di rotazione, traslazione, scalatura e proiezione. La parte destra della figura mostra come il quadrilatero a cui è stata applicata la texture può apparire sullo schermo dopo questo processo di trasformazione. Si può notare come la texture viene distorta per corrispondere alla distorsione del quadrilatero. In questo caso viene dilatata nella direzione x e ridotta nella direzione y; è possibile anche notare, prestando attenzione al motivo della texture, come essa è stata ruotata per aderire alla leggera rotazione a cui è stato sottoposto il poligono. A seconda delle trasformazioni applicate un pixel della texture può corrispondere a più di un pixel sullo schermo o più pixel della texture possono corrispondere a un singolo pixel sullo schermo, si necessita quindi di politiche di ridimensionamento dell'immagini del tutto analoghe a quelle viste in precedenza. Per soddisfare la necessità di un trade-off configurabile che tenga conto da un lato dei requisiti computazionali dell'applicazione e dell'altro la necessità di un rendering realistico, OpenGL offre la possibilità di definire le modalità di manipolazione della texture per adattare il processo di texture mapping alla singola applicazione. La specifica dei parametri avviene attraverso la funzione OpenGL

```
void glTexParameterf(GLenum target,  
                     GLenum pname,
```

GLfloat param)

dove **target** indica la famiglia di texture su cui si vuole operare. Condideriamo **GL\_TEXTURE\_2D** la famiglia delle texture bidimensionali, **pname** il nome del parametro su cui si vuole operare, nel nostro caso **GL\_TEXTURE\_MIN\_FILTER** e il filtro da applicare chiamato *Minification filter*. Il Minification filter viene utilizzato quando la texture è più grande dell'oggetto su cui viene incollata o quando l'oggetto è lontano, quindi significa che una texel rappresenta più pixel reali. Per il Minification filter si possono però utilizzare opzioni specifiche per la metodologia di interpolazione, alcune delle quali specifiche del processo di riduzione tra le quali la tecnica del mipmapping che vedremo più tardi. Una volta creata la texture deve essere attivata e devono essere indicate le sue coordinate rispetto al poligono. Per attivarla è sufficiente utilizzare la funzione **glBindTexture** passando a essa l'identificativo della texture su cui si sta operando. Per la specifica delle coordinate vengono utilizzate le coordinate  $(s, t)$  dove  $s$  indica il valore in  $x$  e  $t$  il valore in  $y$  rispetto alle coordinate nello spazio delle texture. Quindi a ogni vertice del poligono che vogliamo texturizzare dobbiamo associare una coppia di coordinate  $s, t$ . Per esempio in un quadrato definito dalle coordinate:

-1.0, 1.0	1.0, 1.0
-1.0, -1.0	1.0, -1.0

Le coordinate delle texture saranno:

0.0, 1.0	1.0, 1.0
0.0, 0.0	1.0, 0.0

## Mipmap

Gli oggetti a cui vengono applicate le texture possono essere visti, come ogni altro oggetto nella scena, a distanze differenti rispetto al punto di vista. In una scena dinamica, in contemporanea all'allontanarsi dell'oggetto texturizzato dal punto di vista, la mappatura della texture deve diminuire in dimensioni per rispecchiare il diminuire delle dimensioni dell'oggetto a cui viene applicata. Per permettere ciò, OpenGL deve applicare un opportuno filtro di scalatura alla texture, senza introdurre degli *artefatti* visivi che possono ridurre drasticamente la fruibilità della scena. Per tornare all'esempio visto sopra del muro, nel caso esso venga a trovarsi in prossimità del punto di vista può essere conveniente utilizzare una texture di grandi

dimensioni, nel caso l'oggetto inizi ad allontanarsi la riduzione della texture deve avvenire in modo continuo per evitare che siano percepibili dei punti di transizione specifici.

Questi punti di transizione possono essere evitati specificando una serie di texture map a risoluzione e dimensioni decrescenti chiamate *mipmaps*. Il termine mipmap, introdotto da Lance Williams, è esplicativo della struttura piramidale. La radice *mip* è un acronimo della locuzione latina *multum in parvo*, letteralmente “Molte cose in un piccolo spazio”. Quando viene impiegato il mipmapping, OpenGL determina automaticamente quale texture utilizzare in base alle dimensioni in pixel dell'oggetto da renderizzare, attraverso questo approccio il livello di dettaglio impiegato nella texture è adeguato per le dimensioni dell'immagine da visualizzare come output del rendering. Naturalmente il mipmapping richiede un overhead computazionale e un spazio di memorizzazione aggiuntivo, tuttavia senza l'impiego del mipmapping la scalatura di oggetti sia per ragioni dimensionali sia per proiezioni andrà a presentare evidenti artefatti visivi. Per l'uso del mipmapping è necessario specificare tutte le dimensioni delle texture in potenze di 2 a partire dalla dimensioni più grande fino ad arrivare alle dimensioni 1x1. Se vogliamo impiegare il mipmapping in congiunzione con una texture di dimensioni 64x16 sarà necessario specificare texture di dimensioni 32x8, 16x4, 8x2, 4x1, 2x1 e 1x1; tuttavia si può omettere questa fase andando a impiegare i meccanismi per la generazione automatica dei livelli di mipmapping presenti in OpenGL.

### **Analogie tra filtraggio delle texture e scalatura dell'immagine**

Nelle sezioni precedenti abbiamo visto come OpenGL offre varie modalità per la scalatura di immagini. Il metodo più semplice attivabile attraverso l'impiego del flag `GL_NEAREST` è del tutto equivalente all'applicazione dell'interpolazione Nearest Neighbour, basandosi sulla copia del pixel prescelto dopo la scalatura delle immagini. D'altro canto il flag `GL_LINEAR` utilizza la combinazione con coefficienti pari alle distanze dei pixel dalla coordinata riscalata, rendendosi quindi direttamente ricollegabile all'impiego dell'interpolazione bilineare. L'impiego delle tecnologie di mipmapping portano al metodo di interpolazione noto come trilineare, che abbiamo deciso di non presentare dato che il suo impiego è generalmente riservato all'ambito videoludico.



### 5.4.3 Codice

Trascurando, data la loro notorietà, le implementazioni dei metodi di interpolazione per l'architettura CPU, presentiamo rapidamente le implementazioni dei due algoritmi per GPU, che si riducono alla semplice specifica degli opportuni parametri delle texture. Per l'interpolazione Nearest Neighbour avremo:

```
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,  
                GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

mentre per l'interpolazione bilineare avremo:

```
glTexParameteri(GL_TEXTURE_RECTANGLE_ARB,  
                GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

Il codice impiegato non si limiterà a effettuare una scalatura singola dell'immagine in quanto un simile problema non rappresenta direttamente una forte esigenza computazionale, ci concentremo su scalature multiple, che partendo da immagini di dimensioni sempre maggiori verranno riscalata progressivamente fino a ottenere immagini di dimensioni ridotte. Questo approccio è impiegato ampiamente in ambito scientifico data l'estrema efficacia dell'analisi multirisoluzione.

### 5.4.4 Risultati

Vediamo i risultati computazionali ottenuti attraverso l'applicazione del codice che, a partire da un'immagine di dimensioni 2048x2048 RGB, effettua scalature discendenti partendo da 2047x2047 fino alle dimensioni di 2x2 con passo 1.

Analizziamo i risultati presentati nelle figure 5.6, 5.7. Come si nota immediatamente la GPU è in grado di ridurre il tempo di calcolo impiegato per l'esecuzione del procedimento implementata. Interessante è il caso dell'interpolazione Nearest Neighbour: come è stato visto in precedenza l'algoritmo richiede un basso numero di operazioni per dato in ingresso, presentandosi quindi come un'algoritmo difficilmente implementabile su architettura GPU, tuttavia il parallelismo massivo presente nell'algoritmo unitamente a circuiteria ottimizzata per l'accesso 2D alla memoria permette alla scheda video di registrare un incremento prestazionale di circa 10 volte.

I risultati relativi all'interpolazione bilineare, come presentati in figura 5.7 mettono in luce come un incremento, per quanto ridotto, del numero di operazioni per byte

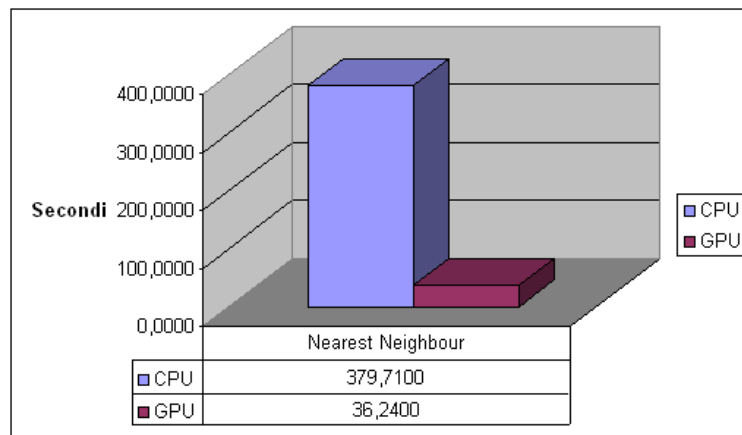


Figura 5.6: Nearest Neighbour

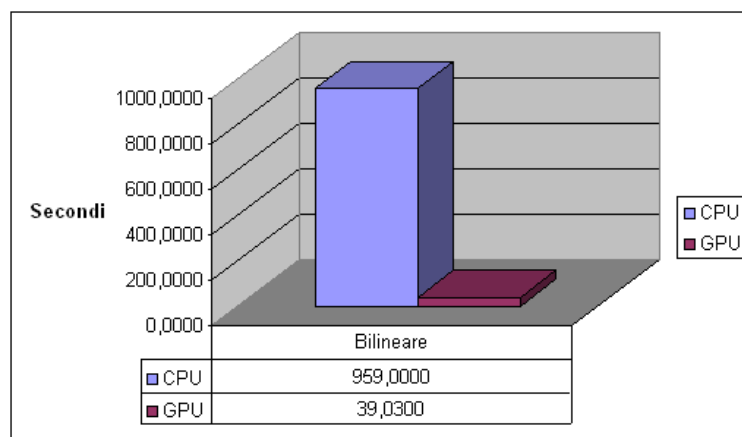


Figura 5.7: Interpolazione Bilineare

in ingresso / uscita forniscono lo spunto strutturale per garantire alla GPU un costo computazionale pari a  $\frac{1}{24}$  del tempo impiegato dalla CPU per svolgere lo stesso compito.

# Capitolo 6

## Calcolo Ibrido

*I not only use all of the brains I have, but all I can borrow.*

Woodrow Wilson

### 6.1 Razioni di un'architettura ibrida

Lo sviluppo di una tecnologia per il calcolo ad alte prestazioni richiede un'attenta analisi della compatibilità dell'approccio scelto con le presenti tecnologie per ridurre, ove possibile, le situazioni di mutua esclusione.

La reimplementazione del nucleo computazionale di una procedura in linguaggio assembler tende, ad esempio, a rendere vani gli sforzi del compilatore di effettuare ottimizzazioni relative alla generazione globale del codice assembler che possono portare a ottenere drastici incrementi prestazionali.

Naturalmente questi aspetti possono essere attenuati, se non completamente soppressi, da un'approccio sistematico e integrato allo sviluppo per le prestazioni. Nel caso del linguaggio assembler i principali compilatori C/C++ offrono la possibilità di impiegare apposite estensioni al linguaggio C direttamente riconducibili a singole istruzioni assembler, ma implementate all'interno del compilatore in modo da poter essere successivamente impiegate dai moduli di ottimizzazione degli stessi. Un esempio lampante di questo approccio è l'uso dei registri *shadow* possibili all'interno dei blocchi *extended* in GCC: richiedendo al programmatore di impiegare una nomenclatura apposita durante la specifica dei registri rende possibile un'efficace allocazione degli stessi lungo tutto il flusso di esecuzione.

Nel caso dell'architettura GPU-CPU il nostro obiettivo è stato quello di rendere possibile l'utilizzo simultaneo dei due dispositivi, cercando, per quanto possibile, di ridurre sia i vincoli entro cui è possibile applicare questo approccio sia l'overhead computazionale legato alle operazioni di sincronizzazione e inizializzazione delle sezioni con esecuzione parallela.

## 6.2 Applicazioni Multithreading

Un qualsiasi programma è costituito dal codice oggetto generato dalla compilazione del codice sorgente. Esso è un'entità statica, che rimane immutata durante l'esecuzione. Un processo, invece, è un'entità dinamica, che dipende dai dati che vengono elaborati, e dalle operazioni eseguite su di essi. Il processo è quindi caratterizzato, oltre che dal codice eseguibile, dall'insieme di tutte le informazioni che ne definiscono lo stato, come il contenuto della memoria indirizzata, i thread, i descrittori dei file e delle periferiche in uso.

Il concetto di processo è associato a quello di thread, con cui si intende una posizione nell'esecuzione del programma. Un processo ha sempre almeno un thread, ma in alcuni casi un processo può avere più thread che avanzano in parallelo. Un'altra differenza fra thread e processi consiste nel modo con cui essi condividono le risorse. Mentre i processi sono di solito fra loro indipendenti, utilizzando diverse aree di memoria ed interagendo soltanto mediante appositi meccanismi di comunicazione messi a disposizione dal sistema, al contrario i thread tipicamente condividono le medesime informazioni di stato, la memoria ed altre risorse di sistema. Quando si vuol far riferimento indistintamente a un processo o un thread, si usa la parola *task*. In un sistema che non supporta i thread, se si vuole eseguire contemporaneamente più volte lo stesso programma, è necessario creare più processi basati sullo stesso programma. Tale tecnica funziona, ma è dispendiosa di risorse, sia perché ogni processo deve allocare le proprie risorse, sia perché per comunicare tra i vari processi è necessario eseguire delle relativamente lente chiamate di sistema, sia perché la commutazione di contesto tra thread dello stesso processo è più veloce che tra thread di processi distinti. Avendo più thread nello stesso processo, si può ottenere lo stesso risultato allocando una sola volta le risorse necessarie, e scambiando i dati tra i thread tramite la memoria del processo, che è accessibile a tutti i suoi thread.

I sistemi operativi si classificano nel seguente modo in base al supporto che offrono a processi e thread:

- Monotasking: non sono supportati né processi né thread; si può lanciare un solo programma per volta.
- Multitasking cooperativo: sono supportati i processi, ma non i thread, e ogni processo mantiene la CPU finché non la rilascia spontaneamente.
- Multitasking preventivo: sono supportati i processi, ma non i thread, e ogni processo mantiene la CPU finché non la rilascia spontaneamente o finché il sistema operativo sospende il processo per passare la CPU a un altro processo.
- Multithreaded: sono supportati sia i processi, che i thread.
- Multitasking embedded: sono supportati i thread, ma non processi, nel senso che si può eseguire un solo programma per volta, ma questo programma può avere più thread (solitamente detti task).

Si noti inoltre che la mancanza di supporto ai thread da parte del sistema operativo non impedisce la programmazione parallela. Infatti il parallelismo tra thread può essere simulato da librerie di programmazione o anche dal supporto run-time del linguaggio di programmazione. In tal senso si parla di thread del kernel per indicare un thread gestito dal sistema operativo, e di thread utente per indicare un thread gestito da una libreria applicativa. Per esempio, alcune versioni di Unix non supportano i thread, per cui si ricorre ai thread utente, altri (per esempio Linux) supportano direttamente i thread a livello del kernel.

## 6.3 Thread Library

La maggior parte del codice multithread sviluppato su piattaforma Linux impiega le librerie pthread (*POSIX thread (pthread) libraries*) la cui API, che si rifà direttamente allo standard POSIX relativo, rappresentano ormai lo standard de facto per il codice sviluppato su sistemi \*nix che faccia uso dei thread.

L'implementazione standard dei thread su linux, nota con il nome di LinuxThread ha rappresentato la prima e tutt'ora la più diffusa implementazione di un supporto

ai POSIX thread.

In seguito alla crescente richiesta di un migliore supporto alla programmazione multithread e di migliori prestazioni dell'intera infrastruttura responsabile della gestione dei thread IBM ha iniziato un processo di sviluppo che ha portato all'implementazione NGPT (*Next Generation Posix Thread*). Questa implementazione, al prezzo di un compatibilità a livello di sorgente, ma non binaria, implementava i più avanzati risultati teorici nel campo della progettazione di sistemi operativi: sopra tutti il supporto al modello di thread M:N in cui il rapporto tra thread a livello kernel e thread a livello utente varia dinamicamente, al contrario del precedente modello 1:1 il cui maggior vincolo è rappresentante dal tetto massimo di thread che possono essere in stato di running contemporaneamente.

In contemporanea allo sforzo sostenuto IBM, nella migliore rappresentazione della filosofia Open Source, Ulrich Drepper e Ingo Molnar realizzano un'ulteriore implementazione i cui primi risultati presentati nel 2003 convincono IBM ad abbandonare il suo progetto. La Native Posix Thread Library (*NPTL*) impiega, al contrario di NGPT, un approccio direttamente riconducibile alla precedente implementazione: privilegiando il vecchio modello 1:1 ne viene fornita una totale reimplementazione volta a sfruttare totalmente il supporto hardware, generalmente ignorato, per la programmazione multithread. Inoltre viene drasticamente ridotto il costo legato alla creazione e distruzione di un nuovo thread, rimuovendo quindi un forte overhead che ha sempre scoraggiato l'impiego massivo della programmazione multithread. In [7] viene presentata la struttura della libreria e ne vengono valutate le prestazioni in base al tempo impiegato da un numero crescente di *creators* per inizializzare e terminare 50 thread ottenendo risultati 4 volte migliori di quelli ottenuti da NGPT e 8 volte migliori rispetto a quelli ottenuti da LinuxThread.

## 6.4 Aspetti Implementativi

Al fine di assicurare la totale indipendenza delle due unità di calcolo abbiamo deciso di impiegare un codice in grado di sfruttare appieno la potenza del processore in modo da assicurare che la potenza residua non vada a coprire overhead nascosti.

A livello di algoritmo è stato scelto il prodotto matriciale implementato per CPU in

[2], mentre per il codice GPU abbiamo scelto quello presentato in [4]. Il problema principale che ci si pone nel momento dell'integrazione della tecnologia multithreading con il calcolo su GPU è legato alla natura stessa di OpenGL che non risulta essere thread safe.

Risulta quindi fondamentale che l'inizializzazione del contesto OpenGL avvenga nello stesso thread che effettuerà poi la computazione GPU, per ridurre il costo legato alla reinizializzazione abbiamo deciso di implementare il codice GPU all'interno del thread principale, mantenendo quindi valido il principio di una sola inizializzazione e una sola terminazione del supporto OpenGL. A prima vista l'assenza di supporto thread safe nello standard OpenGL può far pensare a una seria limitazione legata alla possibilità di impiegare unicamente una sola GPU, tuttavia la tecnologia Scalable Link Interface (*SLI*), sviluppata da NVIDIA, permette di collegare due o più schede video per produrre un unico segnale video in uscita. Il nome SLI è stato inizialmente utilizzato da 3dfx con il suo Scan-Line Interleave, che è stato introdotto nel 1998 e utilizzato nelle linee di acceleratori grafici Voodoo 2 e successive. Con SLI è possibile quasi duplicare la complessità grafica che un personal computer può gestire attraverso l'inserimento di una seconda scheda video identica (altrimenti non è assicurata la compatibilità).

Nel gennaio 2006, NVIDIA e Dell hanno annunciato di aver realizzato un nuovo sistema, quattro volte più potente, il nuovo Quad-SLI che avrebbe permesso a quattro schede video di lavorare in modalità SLI. La compagnia OEM Dell ha rivelato al Consumer Electronics Show del 2006 un computer Dell XPS che utilizza quattro schede video GeForce nella modalità Quad SLI.

A livello tecnico le schede in SLI possono essere configurate per assumere tre modalità totalmente trasparenti all'applicazione:

- Split Frame Rendering (*SFR*) in cui il frame viene suddiviso in  $n$  sottoframe renderizzati indipendentemente dalle  $n$  schede;
- Alternate Frame Rendering (*AFR*) in cui i frame vengono interamente elaborati su una scheda ma a turni alternati;
- SLI Antialiasing in cui la potenza offerta viene impiegata non per un incremento delle prestazioni ma per un miglioramento della qualità visiva della scena renderizzata.

Questi profili possono essere modificati a runtime interagendo con il driver risultando quindi configurabili da parte dell'utente, ma totalmente trasparenti all'applicazione, che può trarre beneficio della potenza di calcolo extra che questi sistemi mettono a disposizione.

## 6.5 Risultati

Dopo aver isolato le componenti di codice implementanti l'algoritmo del prodotto matriciale abbia provveduto ad allocare tre matrici di dimensioni  $n \times n$  con  $n \in \{512, 1024, 2048\}$ , dove due ( $A, B$ ) di esse saranno impiegate come matrici in input e la restante ( $C$ ) come output.

I valori delle matrici in input sono stati scelti attraverso la generazione di numeri casuali, al fine di impedire al compilatore ottimizzazioni che non sarebbero state significative al di fuori del contesto in cui si sono svolti i test.

Il codice è stato inizialmente eseguito sequenzialmente, verificando che l'ordine delle chiamate (GPU, CPU vs CPU, GPU) non alteri le prestazioni del codice in esame. Questa verifica è stata necessaria in quanto come presentato in [2] il ruolo giocato dalla cache nell'esecuzione di un algoritmo può alterare pesantemente le prestazioni del software. Un simile vantaggio, per quanto utile in generale, sarebbe stato controproducente se applicato a questo contesto, in quanto l'accelerazione del secondo prodotto  $A \times B$  dovuta al fatto di aver già compiuto un simile prodotto avrebbe inficiato la valutazione dell'overhead dovuto all'esecuzione parallela. Come si nota in

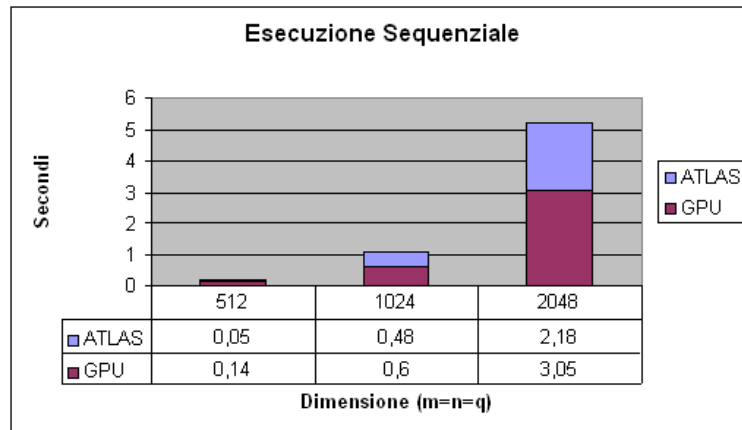


Figura 6.1: Tempi per l'esecuzione sequenziale



figura 6.1 vengono confermati i risultati presentanti in [4]: il codice ATLAS presentato in [2] risulta più performante rispetto all'esecuzione dell'algoritmo su GPU.

Questa superiorità non deve far pensare che l'uso della GPU sia controproducente: il prodotto matriciale contenuto in ATLAS rappresenta il risultato degli sforzi congiunti dei migliori programmatori assembler e progettisti di CPU ottenuto in circa 5 anni di lavoro; inoltre anche dal punto di vista teorico la moltiplicazione matriciale *cache orientend* è stata studiata approfonditamente, al contrario dell'esecuzione *stream oriented*.

L'esecuzione parallela, i cui risultati sono presentati in figura 6.2, mostra come sia

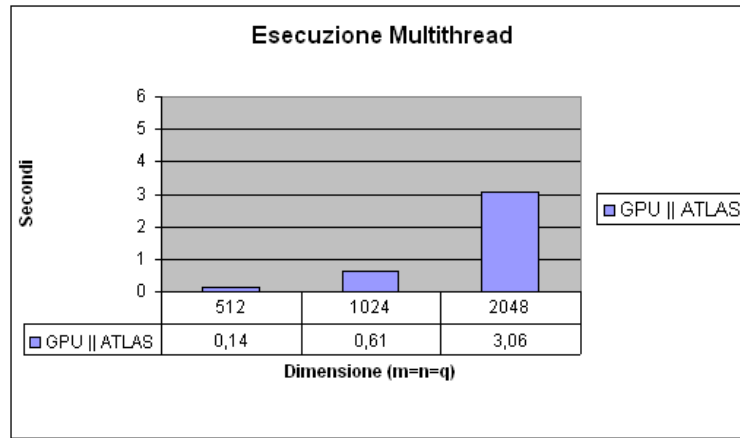


Figura 6.2: Tempi per l'esecuzione parallela

stato raggiunto appieno il risultato ottenuto. Siano  $C_{cpu}$ ,  $C_{gpu}$  i costi computazionali delle esecuzioni su CPU e GPU, il codice implementato con l'impiego dell'esecuzione parallela presenta un costo tale per cui  $C_{gpu||cpu} \approx \max[C_{gpu}; C_{cpu}]$  con un overhead totalmente trascurabile.



# Capitolo 7

## Conclusioni

*Facts are more mundane than fantasies, but a better basis for conclusion.*

*Amory Lovins*

In questo lavoro di tesi abbiamo mostrato come sia possibile impiegare una famiglia di dispositivi, sviluppati esclusivamente in ambito industriale, per compiere un ulteriore passo in avanti nell'ambito del calcolo ad alte prestazioni.

Inoltre abbiamo mostrato come uno sviluppo con linguaggi compilati a runtime per GPU possa portare a implementazioni software in grado di cavalcare quell'onda prestazionale che, muovendosi con un ritmo pari al cubo della legge di Moore, ha portato i dispositivi per l'accelerazione 3D dei software per il video intrattenimento a diventare i sistemi di calcolo personale più potenti mai realizzati. Se, da un lato, un adattamento dell'hardware per una mappatura diretta con un'architettura classica di Von Neumann potrebbe portare a una semplificazione del software da sviluppare, dall'altro lato farebbe perdere la spirale positiva [10] che ha caratterizzato il successo di questi sistemi.

Inoltre la valutazione delle prestazioni elementari ha permesso di gettare un po' di luce sulle proprietà richieste a un algoritmo per poter ottenere un miglioramento delle sue prestazioni.

Il lavoro svolto sugli algoritmi di scalatura ha mostrato i passi necessari per sviluppare un software in grado di sfruttare con successo il potere computazionale del dispositivo.

L'integrazione del codice sviluppato per GPU con codice operante su CPU ha mostrato come sia possibile sfruttare una possibilità ottimizzativa come affiancamento

e non in sostituzione di precedenti approcci aprendo quindi la strada a un suo vasto impiego.

# Appendice A

## Sistema di Testing

*What you see is what you have.*

*Brian Kernighan*

Il sistema impiegato per la valutazione delle prestazioni è un Pentium 4 con HT (*HyperThread Technology*) e una frequenza di clock pari 3.03 Ghz, 1 GByte di memoria distribuita su due banchi DDR 2. La scheda video impiegata per i test è una scheda NVIDIA 7800 GT con un bus di memoria a 256 bit, una frequenza di core pari 400 MHz, con 20 pixel shader e 7 unità per la computazione sui vertici e una banda passante per il trasferimento dati pari a un tetto teorico di 32GB.



Figura A.1: Nvidia 7800 GT

Benchè il codice sia stato sviluppato in modo da risultare totalmente multiplatforma su tutti i sistemi operativi supportati dal linguaggio Gc (Windows, Linux, FreeBSD) i test prestazionali sono stati condotti su una distribuzione Gentoo Linux che offre un controllo assoluto sul processo di costruzione del sistema di base, offrendo la possibilità di effettuare un tuning quantitativo su tutte le componenti del sistema.

Il sistema sfrutta un kernel versione 2.6.18 mentre il compilatore C del sistema è GCC 4.1. Il compilatore Cg impiegato è la versione 1.4.1, mentre i driver impiegati sono i nuovi 1.0-8774. Il supporto alla programmazione multithread è garantito dalle glibc versione 2.3.5-r2 opportunamente compilate per offrire supporto NPTL (*Native Posix Thread Library*). Il sistema impiegato per la verifica della totale portabilità è stato un Turion64 ML-32 con una frequenza di clock pari a 1.6 GHz, 1 GByte di memoria distribuita su due banchi DDR a 333 MHz. La scheda video impiegata è stata una NVIDIA 7300 con supporto alla tecnologia TurboCache. Il software è

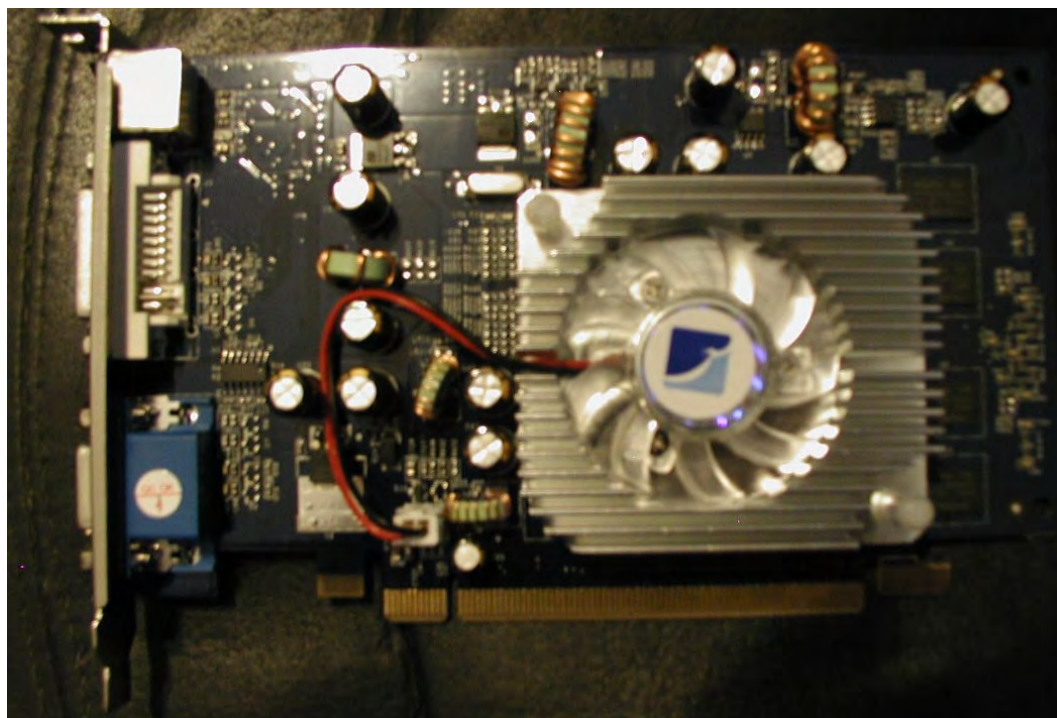


Figura A.2: Nvidia 7300

stato compilato con Microsoft Visual Studio 2005 ed eseguito su Windows XP Professional Service Pack 2 con gli ultimi aggiornamenti disponibili.

Inoltre è stato verificato il funzionamento del compilatore Cg anche su scheda video

---

di vecchia generazione per garantirne il corretto funzionamento anche in assenza di un profilo adeguato. La scheda impiegata è stata una NVIDIA 5200. Come sistema

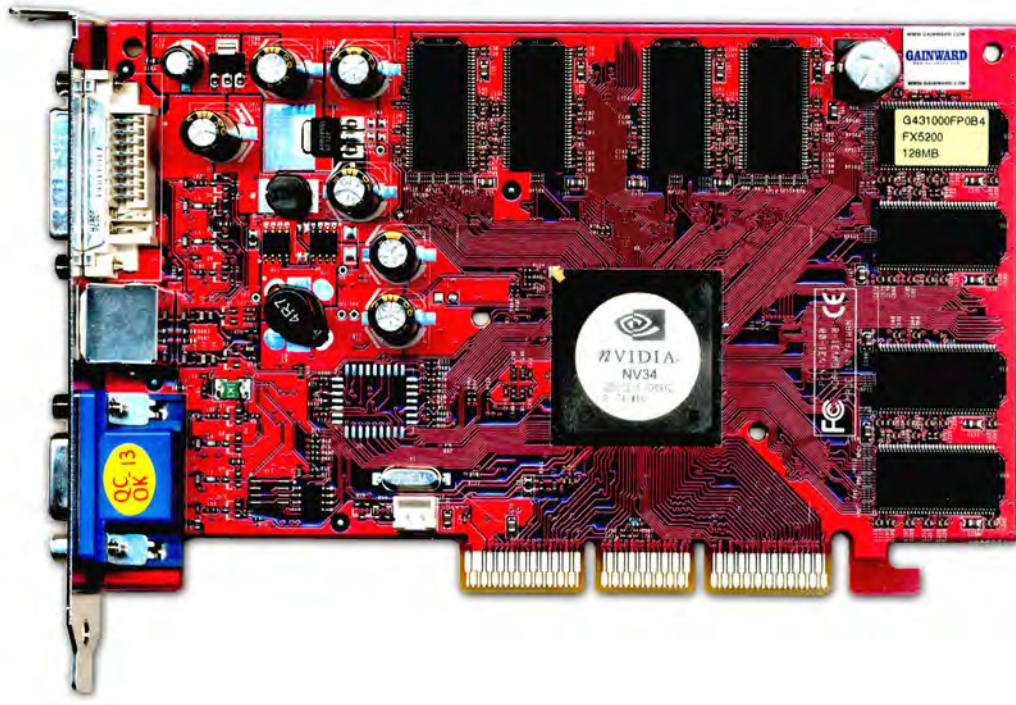


Figura A.3: Nvidia 5200

di build è stato scelto in entrambi in casi il software SCONS, in grado di fornire un supporto superiore allo sviluppo del software.





# Appendice B

## YAW - Yet Another Wrapper

*Real programmer can write assembly code in any language.*

*Larry Wall*

Durante lo sviluppo legato allo svolgimento di questo lavoro di tesi sono emersi dei punti in comune tra le diverse implementazioni sia del medesimo algoritmo sia di algoritmi diversi. Queste osservazioni unitamente alla volontà di poter riutilizzare le ottimizzazioni individuate hanno portato alla decisione di implementare un framework per semplificare lo sviluppo di applicazioni facenti uso dell'architettura GPU.

Il primo passo è stata la creazione di una classe che rappresenti il punto di interazione diretta con il sottosistema GPU, fornendo l'accesso alle procedure di inizializzazione e di query delle funzionalità presenti.

```
class LBGPGPU
{
    static CGcontext  context;
    static bool glewInited;
    static LBGPGPU* _hInstance;
    LBGPGPU();
    ~LBGPGPU();
    void initGLUT();
    void initCG();

public:
```

```
static LBGPGPU* getInstance();
CGprofile getFragmentProfile();
CGprofile getVertexProfile();
CGcontext getContext();
bool glewActivated() {return glewInited;}
void doProjection(int w,int h);
void drawTexturedQuad(int qw, int qh,int tw=-1, int th=-1);
};
```

Questa classe sfrutta il pattern *Singleton* per modellare al meglio l'unicità del contesto OpenGL all'interno dell'applicazione; vengono inoltre fornite le funzionalità base per ogni programma GPGPU: la specifica delle dimensioni della proiezione e per l'operazione di disegno del quadrato equivalente, come visto in precedenza, all'invocazione di una procedura per CPU. La classe si occupa, se necessario, di inizializzare e terminare automaticamente tutte le risorse necessarie (OpenGL, GLUT, GLEW, Cg).

La seconda classe implementata modella un singolo programma Cg fornendo, attraverso il meccanismo di overloading di funzioni presente in C++, gli strumenti necessari per un rapido caricamento e utilizzo dei programmi Cg.

```
class LBGLProgram
{

    bool fragment;
public:
    CGprogram program_id;
    LBGLProgram(const char *programname,
        const char *progfile,
        bool fragment=true
    );
    void Activate();
    void SetParameter(const char*paramname, GLuint texture);
    void SetParameter(const char*paramname, float scalar);
    void SetParameter(const char*paramname, double scalar)
    {
```

---

```
    SetParameter(paramname,(float)  scalar);
}
};
```

La classe più importante del framework sviluppato è indubbiamente la LBFBO che fornisce una valida astrazione dei FrameBuffer Object:

```
class LBFBO
{
static GLuint framebuffer_id;
static int counter;
GLuint texture_id;
GLenum latest_buffer;
int w,h;
bool rgba;
void initFramebufferObject();
void checkFramebufferStatus();
public:
~LBFBO();
LBFBO();
GLuint getId(){return texture_id;}
void Init(int w, int h,bool rgba=true);
void AsOutput(GLenum colorAttach=GL_COLOR_ATTACHMENT0_EXT);
void AsInput();
void Upload(float *data);
float* Download(GLenum colorAttach=GL_COLOR_ATTACHMENT0_EXT);
void SaveAsRawImage(const char*filename);
void LoadFromRawImage(const char*filename);
};
```

Attraverso l'impiego di questo framework è possibile riscrivere il codice impiegato per la valutazione delle prestazioni elementari nel caso del seno attraverso le seguenti linee di codice:

```
#include "LBGPGPU.h"
#define W 2048
```

```
#define H 2048
LBFB0 source,output;
LBGLProgram cgp("seno","seno.cg");
LBGPGPU* gpu = LBGPGPU::getInstance();
float* doComputation_GPU(float *memory)
{
source.Init(H,W,true);
output.Init(H,W,true);
source.Upload(memory);
output.AsOutput();
cgp.SetParameter("textureX",source.getId());
cgp.Activate();
gpu->drawTexturedQuad(W,H);
return output.Download();
}
```

Il framework, pubblicato sul sito di riferimento per lo sviluppo di applicazioni GPU <http://www.gpgpu.org>, ha riscosso un discreto interesse per l'approccio scelto per la modellazione delle componenti, nonostante le ovvie limitazioni di un framework sviluppato con un target applicativo di dimensioni ridotte.

# Bibliografia

- [1] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home an experiment in public-resource computing. *Communications of the ACM*, 45:56 – 61, 2002.
- [2] L. Benini. Ottimizzazioni microarchitetturali per high performance computing. Master's thesis, Scienze dell'Informazione - Università di Bologna - Sede di Cesena, 2003-2004.
- [3] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. pages 917–924.
- [4] M. D. Carolis. High performance computing su unita' grafiche programmabili. Master's thesis, Scienze dell'Informazione - Università di Bologna - Sede di Cesena, 2004-2005.
- [5] K. R. Castleman. *Digital Image Processing*. Prentice Hall, 1996.
- [6] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345 – 363, 1936.
- [7] U. Drepper and I. Molnar. The native posix thread library for linux. <http://people.redhat.com/drepper/nptl-design.pdf>, 2003.
- [8] M. L. et al. Experiences with the glite grid middleware. *Proc. Computing for High Energy Physics (CHEP)*, 2004.
- [9] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. Gpu cluster for high performance computing. *sc*, 00:47, 2004.
- [10] B. Gates. *La strada che porta al domani*. Arnoldo Mondadori Editore, 1995.

- [11] M. I. Gold. The arb multitexture extension. <http://berkelium.com/OpenGL/GDC99/multitexture.html>, 1999.
- [12] M. Harris. Gpgpu lessons learned. <http://download.nvidia.com/developer/presentations/2006/gdc/2006-GDC-OpenGL-tutorial-GPGPU-2.pdf>, 2006.
- [13] D. Hilbert and W. Ackermann. *Grundzuge der theoretischen Logik*. Springer, 1928.
- [14] J. Krüger and R. Westermann. Linear algebra operators for gpu implementation of numerical algorithms. pages 908–916.
- [15] K. Moreland and E. Angel. The fft on a gpu. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.
- [16] A. Origin. Beingrid: Business experiments in grid. <http://www.beingrid.com/>, 2006.
- [17] M. Roffilli. *Advanced Machine Learning Techniques for Digital Mammography*. PhD thesis, University of Bologna, Department of Computer Science, 2006.
- [18] R. Surdulescu. Cg bumpmapping. <http://www.gamedev.net/reference/articles/article1903.asp>, 2003.
- [19] A. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1937.