

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

HIGH PERFORMANCE COMPUTING  
SU UNITA' GRAFICHE PROGRAMMABILI

**Tesi di laurea in**

Fisica Numerica

**Relatore**

Prof. Renato Campanini

**Presentata da**

Mauro De Carolis

**Co-relatore**

Dott. Matteo Roffilli

Sessione III

Anno Accademico 2004/2005

# INDICE

<b>ARCHITETTURE PER L'HIGH PERFORMANCE COMPUTING</b>	<b>1</b>
1.1 LA CPU E IL MODELLO DI VON NEUMANN	1
1.2 IL PROCESSORE VETTORIALE	3
1.3 IL CALCOLO PARALLELO	4
1.4 GLI STREAM PROCESSOR E LO STREAM PROGRAMMING MODEL	5
1.5 SCHEDE GRAFICHE PROGRAMMABILI	6
1.6 LA TECNOLOGIA SLI E CROSSFIRE	7
1.7 CONFRONTO DEL CALCOLO SULLE DIVERSE ARCHITETTURE	8
<b>ANALISI DELL'ARCHITETTURA GPU</b>	<b>11</b>
2.1 LA COMPUTER GRAPHICS	11
2.2 DIFFERENZE ARCHITETTURALI TRA CPU E GPU	13
2.3 LA PIPELINE GRAFICA	16
2.3.1 <i>Vertex transformation</i>	18
2.3.2 <i>Primitive assembly</i>	20
2.3.3 <i>Rasterizzazione, interpolazione del colore e texturing</i>	20
2.3.4 <i>Fragment operations e texturing</i>	21
<b>DISPOSITIVI HARDWARE</b>	<b>25</b>
3.1 NVIDIA GeForce SERIE 6	26
3.2 NVIDIA GeForce SERIE 7	27
3.3 ATI RADEON SERIE X1000	28
<b>LINGUAGGI</b>	<b>33</b>
4.1 I METALINGUAGGI	33
4.2 I LINGUAGGI DI SHADING	34
4.3 IMPLEMENTAZIONE DELLE STRUTTURE DATI SU GPU	35
4.4 IL MODELLO DI MEMORIA DELLA GPU	36
4.4.1 <i>Vertex streams</i>	37
4.4.2 <i>Fragment stream</i>	37
4.4.3 <i>Frame buffer stream</i>	38
4.4.4 <i>Texture stream</i>	38
4.5 PROGRAMMABILITÀ GENERAL PURPOSE	39
4.6 I MODELLI DI PROGRAMMAZIONE	41
4.6.1 <i>Shader Model 1.0</i>	41
4.6.2 <i>Shader Model 2.0</i>	42
4.6.3 <i>Shader Model 3.0</i>	43
4.7 PROFILI	45
4.8 LA SCELTA DEL LINGUAGGIO	46
4.9 L'AMBIENTE GRAFICO	50
4.9.1 <i>OpenGL e Direct3D</i>	51
4.9.2 <i>Il compilatore</i>	52
4.9.3 <i>La compilazione dinamica</i>	52
4.9.4 <i>Il window system</i>	53
<b>APPLICAZIONE</b>	<b>55</b>
5.1 LE MATRICI	55
5.2 IL PRODOTTO MATRICE PER MATRICE	56
5.2.1 <i>Il prodotto matrice per matrice "ijk"</i>	56
5.2.2 <i>Il prodotto matrice per matrice con SSE</i>	57
5.2.3 <i>Il prodotto matrice per matrice con ATLAS</i>	58
5.2.4 <i>Il prodotto matrice per matrice su GPU</i>	60
5.2.5 <i>Il prodotto matrice per matrice a blocchi</i>	62
5.2.5 <i>Il prodotto matrice per matrice su GPU in RGBA</i>	64
5.2.6 <i>Il wrapper e l'unwrapper</i>	66

<b>IMPLEMENTAZIONE DEL PRODOTTO MATRICIALE SU GPU</b>	<b>69</b>
6.1 PROIEZIONE DEI PARADIGMI DI PROGRAMMAZIONE ALLA GPU	69
6.2 IMPLEMENTAZIONE	71
6.3 L'INTERFACCIA	71
6.3.1 <i>WrapCPU ed unwrapCPU</i>	73
6.3.2 <i>GlutCreateWindow e glewinit</i>	73
6.3.3 <i>InitCG</i>	73
6.3.4 <i>InitGL</i>	74
6.3.5 <i>EnableCG e ChkFBO</i>	74
6.3.6 <i>Display</i>	74
6.4 IL PRODOTTO MATRICIALE COME SHADER	76
<b>ANALISI DELLE PRESTAZIONI</b>	<b>79</b>
7.1 PIATTAFORMA DI LAVORO	79
7.2 TEST COMPARATIVI	80
7.3 CONFRONTO NUMERICO DIRETTO	84
7.4 ANALISI INTERNA DELL'APPLICAZIONE GPGPU	85
7.5 CONCLUSIONI	86
<b>BIBLIOGRAFIA</b>	<b>89</b>

## Introduzione

*Il processo attraverso il quale si sviluppa e si amplia la conoscenza scientifica si può vedere come la costruzione di una piramide al contrario. Fin dalle sue origini, l'essere umano ha iniziato a classificare, enumerare e correlare.*

*Risale al 2450AC, in Egitto, il primo metodo di calcolo della circonferenza.*

*Eratostene nel 240AC usa il crivello di Eratostene per isolare i numeri primi dai numeri non primi, dimostrando che anch'essi sono infiniti.*

*Nel 250 DC Diofanto usa dei simboli per definire dei termini sconosciuti e scrive *Arithmetica*, la prima trattazione sistematica dell'algebra.*

*Nel 750 DC Al-Khwarizmi lavora sui dettagli dell'aritmetica e dell'algebra ed alla schematizzazione della teoria delle equazioni lineari e quadratiche.*

*Nel 1629 Pierre de Fermat sviluppa un rudimentale calcolo differenziale.*

*Questi sono solo alcuni dei fondamentali passi in avanti che si sono fatti lungo la storia della scienza, fino ad arrivare alla nascita dell'informatica che si fa risalire al 1642, quando Pascal realizza la cosiddetta "pascalina" macchina metallica in grado di eseguire automaticamente addizioni o sottrazioni.*

*Molto anni dopo nel 1936 il logico e matematico inglese Alan Turing enuncia il modello teorico del calcolatore moderno, la cosiddetta "macchina di Turing", limite tuttora invalicabile che divide i problemi calcolabili da quelli incalcolabili, macchina che potrebbe eseguire qualsiasi software moderno.*

*Questa immaginaria linea evolutiva del calcolo costituisce l'asse portante della moderna ricerca scientifica. L'infinitamente piccolo è esplorato sempre più a fondo, si osserva l'universo a distanze crescenti, si trovano nuove relazioni tra elementi conosciuti, si formalizza sempre più accuratamente il mondo reale attraverso modelli matematici.*

*Al centro di tutto ciò rimane il calcolo ed il propulsore fondamentale, la potenza di calcolo, la quantità di numeri memorizzabili e il volume di operazioni eseguibili in un secondo.*

*Nel 1822 Babbage - a lui si deve la descrizione del primo calcolatore digitale automatico di uso generale, la cosiddetta "macchina analitica", modello per tutti i successivi calcolatori digitali universali - comunica a un collega*

*astronomo l'idea di un calcolo automatico per la compilazione delle tavole astronomiche: "Volesse il cielo che questi calcoli fossero eseguiti a vapore".*

*Il vapore è stato sostituito dall'elettricità e la potenza di questa macchina è in continua crescita.*

*La scienza che si occupa di utilizzarla al meglio è l'High Performance Computing.*

# Capitolo 1

## Architetture per l'High performance Computing

L'High Performance Computing ( HPC ) consiste nell'ottimizzare il codice di un programma e nell'utilizzo efficiente delle risorse hardware disponibili, incrementando la potenza di calcolo di un computer senza ricorrere a costosi supercomputer.

L'HPC partendo dal modello di Von Neumann propone diverse soluzioni del problema.

### 1.1 La CPU e il modello di Von Neumann

Con il termine modello di Von Neumann o macchina di Von Neumann, si indica uno schema a blocchi che descrive il comportamento di un “esecutore sequenziale a programma memorizzato”.

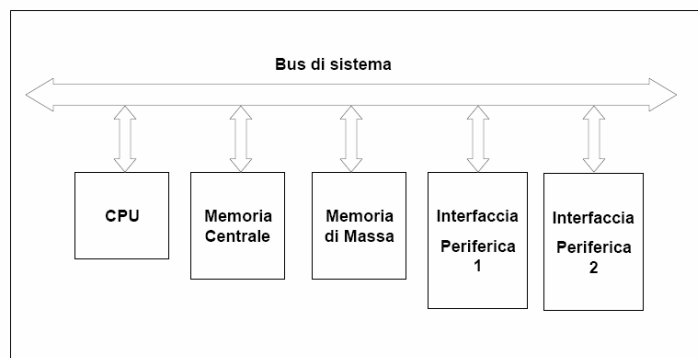


Figura 1 Modello di Von Neumann

Questo modello, ideato dal ricercatore americano di origine tedesca nel corso della seconda guerra mondiale per la realizzazione dei primi elaboratori, è adatto anche per una descrizione elementare del principio di funzionamento delle moderne CPU.

Un generico programma residente in memoria di massa viene caricato in memoria centrale ed eseguito avviando il ciclo fetch-execute ( estrazione ed esecuzione ).

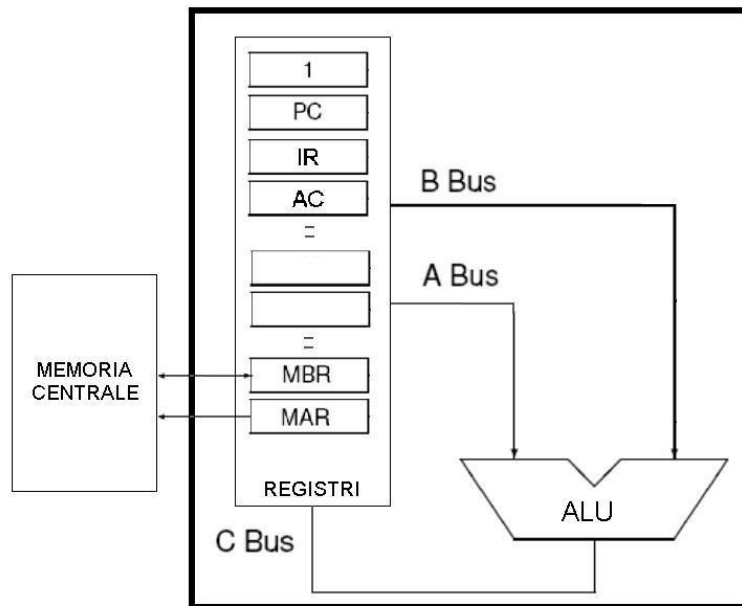
Il processore esegue le singole istruzioni nella sequenza stabilita dal programma iterando il ciclo.

- Estrazione dalla memoria centrale dell'istruzione.
- Esecuzione dell'istruzione contenuta in IR.

Nella prima fase, l'estrazione dell'istruzione dalla memoria centrale avviene all'indirizzo indicato dal contatore di programma PC, inoltre, la stessa viene caricata nel registro interno di istruzione IR e si incrementa il PC, che punterà all'istruzione successiva.

Nella fase di esecuzione si possono incontrare diversi tipi di istruzioni:

- Istruzione di calcolo, il valore di uno o due registri convergono nell'Arithmetic Logic Unit ( ALU ) attraverso i bus A e B il risultato viene scritto su un registro di destinazione attraverso il bus C.
- Istruzione di lettura di un valore dalla memoria centrale all'indirizzo indicato dal memory address register MAR che sarà caricato nel registro memory buffer register MBR oppure di scrittura, allora il valore contenuto in MBR verrà scritto in memoria centrale all'indirizzo specificato dal MAR.
- Istruzione di salto o di altra istruzione che altera la sequenza di esecuzione del programma cambiando il valore di PC.



**Figura 2 Architettura di una CPU**

## **1.2 Il processore vettoriale**

Questo tipo di processore, pur rispettando la struttura descritta da Von Neumann esegue la stessa istruzione su più dati contemporaneamente. Oltre ai normali registri e istruzioni scalari, contiene degli speciali tipi di registri (registri vettoriali) che possono contenere N valori contemporaneamente, ed ogni operazione che coinvolge uno di questi registri viene eseguita su tutti i valori in esso memorizzati.

Secondo la classificazione di Flynn il processore vettoriale rientra tra i sistemi di calcolo Single Instruction Multiple Data.

I processori vettoriali sono comuni nelle applicazioni scientifiche e sono spesso alla base dei supercomputer come il CRAY-1 fin dagli anni '80. Con la fine degli anni 90 i microprocessori sono cresciuti di prestazioni e molti processori per applicazioni generiche si sono dotati di unità vettoriali o sono diventati vettoriali al loro interno.

E' il caso del "3DNow!" nome di un'estensione multimediale creata da AMD per i propri processori, a partire dal K6-2 del 1998; del MMX ( Multi Media

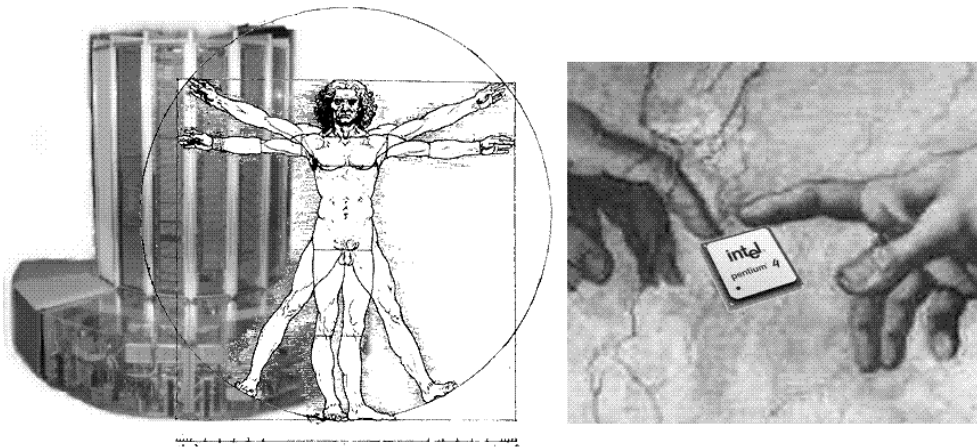


eXtension ) e del SSE (Streaming SIMD Extensions) presentate dalla Intel rispettivamente nel 1997 e nel 1999.

Quest'ultima venne introdotta con l'uscita del processore Pentium III e successivamente fu adottata dalla stessa AMD.

L'estensione SSE prevede registri vettoriali di 4 floating-point a singola precisione ( 32 bit ) che consentono il quadruplo delle prestazioni.

Il CRAY-1 misurava due metri di altezza e quasi tre di diametro, al costo di 700.000 dollari nel 1976 sviluppava 80 Mflops. Un computer con un pentium 4 al suo interno costa poche centinaia di euro e sviluppa più di 1000 Mflops.



**Figura 3 Un CRAY-1 ed un Pentium4 a confronto**

### **1.3 Il calcolo parallelo**

Il calcolo parallelo è l'esecuzione simultanea dello stesso codice, diviso e specificamente adattato, su più microprocessori allo scopo di aumentare le prestazioni del sistema.

Questa ottimizzazione esula dall'architettura interna della singola CPU e riguarda la comunicazione, i flussi dei dati e la ripartizione del calcolo e della memoria tra i vari processori.

Il Blue Gene, il più potente computer del pianeta ha 32.768 processori installati, una variante dei processori PowerPC appositamente sviluppata per questa architettura, ha una potenza di calcolo di 280 Tflops.

Soluzioni meno potenti ma molto più economiche sono i cluster, costituiti da un numero variabile di PC interconnessi tramite rete ethernet, ampiamente descritti da O.Schiaratura [SCH03] e C.Zoffoli [ZOF05]. In questo caso la potenza di calcolo dipende dal numero di PC che compongono il cluster. Si può raggiungere una potenza di centinaia di Gflops con l'evidente limite dello spazio necessario per installare un sistema del genere.

## 1.4 Gli stream processor e lo stream programming model

Lo stream programming model è un paradigma relativamente nuovo che permette di raggiungere livelli di efficienza di calcolo ottimi.

Comparato con le architetture esistenti, ha prestazioni venti volte maggiore a parità di consumo energetico e dimensioni.

Dato un insieme di valori in input ( input stream ), il paradigma è essenzialmente basato nel definire una serie programmi indipendenti ( kernel ) che saranno applicati ad ogni elemento del flusso di input.

Fino ad ora si erano visti solo esempi di *uniform streaming paradigm* dove c'è un unico kernel che elabora i dati, le sezioni che lo precedono o lo seguono si occupano di gestire il flusso di dati.

L' uniform streaming, essenzialmente SIMD, ha comunque il vantaggio di semplificare il flusso dei dati portando ad un incremento prestazionale.

Lo stream programming model è alla base di molti dei nuovi chip presentati sul mercato e di molti in uscita.

Tra gli ispiratori di questo modello troviamo "Imagine" processore sviluppato dalla Stanford University, con un'architettura veramente flessibile, veloce ed energeticamente efficiente. Costruita con tecnologia 0.15 micron, è composta da otto cluster di sei ALU ognuna.

Di prossima uscita invece, il microprocessore "Cell" sviluppato da Sony, Toshiba e IBM la cui prima applicazione commerciale sarà la PlayStation 3.

Questo processore è composto da 8 Synergistic Processing Elements (SPE) ognuna composta da una Synergistic Processing Unit ( SPU ) e da un'unità per la gestione della memoria.

Un altro esempio di processori stream, ma con un ridotto livello di genericità, ma dai costi contenuti e dallo sviluppo e diffusione certi sono le schede grafiche programmabili.

## **1.5 Schede grafiche programmabili**

L'affacciarsi sul mercato di schede grafiche programmabili, sta aprendo una nuova strada nell'HPC, portando ad una crescita d'interesse nell'applicare alle Graphic Processing Unit ( GPU ) algoritmi general purpose.

La combinazione di memorie a banda larga ed hardware dotato di aritmetica floating-point, con prestazioni maggiori rispetto ad una CPU convenzionale, rendono i processori grafici strumenti potenti per l'implementazione di algoritmi di calcolo.

Gli strumenti utilizzati nel campo del GPGPU ( General-Purpose Computation using Graphics Processing Units ) sono quelli tipici della programmazione in ambiente grafico. Strumenti molto potenti, portabili e con un livello di sviluppo e stabilità ormai assodati. Quello che cambia è il modo di utilizzarli, ma soprattutto, un significato alternativo che si attribuisce ai dati ed alla loro elaborazione. Occorre fondere concetti di programmazione parallela, computer graphic, algoritmi numerici classici ed osservare, concepire, descrivere ogni cosa seguendo parallelamente due prospettive, quella grafica, imposta dalla natura dell'ambiente di lavoro e quella del calcolo che rappresenta l'obiettivo che ci si propone.

L'architettura di una GPU è totalmente diversa dal modello di Von Neumann, essendo progettata esclusivamente per elaborare elementi grafici.

Essa è composta da più classi di unità di calcolo collocate serialmente lungo una pipeline. Le unità all'interno di ogni classe sono disposte parallelamente ed eseguono tutte la stessa operazione su dati diversi, secondo la logica SIMD,

mentre diverse classi eseguono diverse operazioni. Si può quindi affermare che le GPU nel complesso sono architetture Multi Instruction Multi Data (MIMD). Per fare un primo confronto, si pensi che una NVIDIA 7800 GTX 512 ha una potenza teorica di 200 Gflops.

## **1.6 La tecnologia SLI e Crossfire**

Un ulteriore scelta architetturale è la SLI di NVIDIA che sta per “Scalable Link Interface”, Crossfire per ATI, sebbene in questo caso il supporto è allo stadio di sviluppo. Questa tecnologia permette a più GPU di lavorare insieme, consentendo un miglioramento di prestazioni. Ovviamente si rende necessaria una scheda madre con due porte grafiche PCI Express 16x e che abbia il supporto SLI. Quando due schede grafiche sono collegate in questa maniera il driver grafico riconosce tale configurazione con il nome SLI Multi-GPU mode. In questa configurazione il sistema riconosce le schede come un unico dispositivo logico che lavora fino a 1.9 volte più veloce che una singola scheda, dato che il driver divide il carico di lavoro tra i due dispositivi fisici. Si noti comunque che con la tecnologia SLI non si raddoppia la memoria video disponibile. Per esempio utilizzando due schede grafiche con 256MB il dispositivo logico risulterà avere 256MB di memoria video. La ragione è che, in generale, il driver replica tutta la memoria video in entrambe le GPU. Cioè, in un dato momento la memoria video della GPU 0 contiene gli stessi dati della memoria video della GPU 1.

Quando si fa girare un applicazione in un sistema SLI, è possibile utilizzare diverse modalità:

- Compatibility Mode
- Alternate Frame Rendering (AFR)
- Split Frame Rendering (SFR)

Il compatibility mode semplicemente utilizza una singola GPU per renderizzare tutto (cioè la seconda GPU rimane inattiva per tutto il tempo). Questa modalità non porta a nessun miglioramento di prestazioni, ma serve solo ad assicurare la compatibilità di un'applicazione con la tecnologia SLI.

In modalità AFR il driver renderizza tutti i frame pari nella GPU 0 ed i frame dispari nella GPU 1. Quindi i diversi frame non sono condivisi e si raggiunge la massima efficienza. Tutto il lavoro di rendering come le operazioni pre-vertex, la rasterizzazione e le operazioni pre-pixel sono divise tra le GPU.

Se qualche dato deve essere condiviso tra i frame, ad esempio il risultato della renderizzazione precedente, i dati devono essere trasferiti tramite Bus.

Il trasferimento costituisce un overhead di comunicazione riducendo la potenza di calcolo teorica che è doppia. Nel caso della programmazione general purpose si utilizza spesso il feedback del rendering, quindi tale modalità è sconsigliabile.

In modalità SFR il driver assegna la porzione superiore del frame alla GPU 0 e la porzione inferiore del frame alla GPU 1. La suddivisione del frame è bilanciata: se la GPU 0 è sottoutilizzata in un frame, perché la porzione superiore non deve essere elaborata a pieno, il driver ne aumenta la dimensione, in modo da distribuire equamente il carico di lavoro. Dividere la scena in due porzioni, impone che tutti i vertici di un frame siano elaborati da entrambe le GPU, mentre l'elaborazione dei fragment è ripartita.

La modalità SFR richiede la condivisione dei dati ed ha un maggiore overhead di comunicazione, ma rimane comunque preferibile nella programmazione general purpose, anche se la potenza di calcolo è notevolmente ridotta.

## **1.7 Confronto del calcolo sulle diverse architetture**

Per capire meglio le differenze tra le architetture presentate ipotizziamo di dover calcolare la sommatoria di 1.000.000 floating-point a 32 bit.

Un Pentium 4 senza l'utilizzo del supporto SSE, quindi che rispetta lo schema di Von Neumann, dovrebbe leggere dalla memoria un valore alla volta e poi

accumulare la loro somma in un registro interno. Semplificando, diremo che sarebbero necessari 1.000.000 letture e 1.000.000 somme.

Lo stesso pentium 4 con il supporto SSE leggerebbe da memoria quattro valori alla volta, ne farebbe la somma ed accumulerebbe il valore in un registro interno.

Mantenendo sempre un certo grado di genericità si potrebbe ipotizzare che sarebbero necessari 250.000 letture e 250.000 somme.

Il Blue Gene suddividerebbe i 1.000.000 valori tra i suoi 32.768 processori, ognuno dovrebbe leggere 30 valori e sommarli ed infine sommare i risultati parziali.

Utilizzando una NVIDIA 7800 GTX 512 i 1.000.000 floating-point verrebbero trasferiti dalla memoria alla GPU sotto forma di immagine per gruppi di quattro valori che rappresentano il colore di un pixel nel formato RGBA, ognuna delle 24 unità di calcolo interne, specializzate per l'elaborazione dei pixel, eseguirebbe in un solo ciclo di clock sia la lettura che la somma dei quattro valori, completando la somma con circa 10.000 cicli di clock.

Ovviamente tra le diverse architetture si dovrà valutare la frequenza del clock e la velocità di trasferimento dei dati, ma da quanto detto si può intuire il significato delle valutazioni in flops/secondo citate.

Tipo di Architettura	Cicli necessari per eseguire la sommatoria di 1.000.000 floating-point
Von Neumann	1.000.000 read + 1.000.000 write
Processor ventricle ( SSE )	250.000 read + 250000 write
Cluster ( Big Blue - 32.768 processors )	30 read + 30 write + Gathering
GPGPU NVIDIA 7800 GTX ( 24 pixel unit )	10.000



## Capitolo 2

### Analisi dell'architettura GPU

Il calcolo scientifico da decenni ha abbandonato il semplice modello architetturale descritto da Von Neumann, affidandosi ad architetture parallele che comunque utilizzano processori fondamentalmente ispirati dal matematico tedesco. La Graphic Processing Units ( GPU ), nata per altri scopi, ha un'architettura totalmente diversa, concepita seguendo il concetto di stream processing.

Per capire al meglio la scelta di utilizzare le GPU per il calcolo scientifico ed il loro funzionamento in questo capitolo introdurremo i concetti fondamentali della computer graphics, vedremo in cosa si differenziano dalle CPU, per poi analizzare in dettaglio la loro architettura interna.

#### 2.1 La computer graphics

La computer graphics è utilizzata ormai in una grande varietà di applicazioni professionali e di consumo. Le principali applicazioni di consumo sono: videogiochi, ritocco fotografico, montaggio di filmati amatoriali; mentre quelle professionali includono: industria cinematografica, tipografia, progettazione grafica (CAD) nelle industrie metalmeccanica, elettronica, impiantistica e edile, visualizzazione di dati tecnico/scientifici (CAE), sistemi informativi territoriali (SIT o GIS).

Inizialmente la gestione della grafica era a totale appannaggio della CPU, mentre le schede grafiche come la VGA standard erano equiparabili all'attuale framebuffer, la memoria che contiene i valori del colore di ogni singolo pixel visibile sul monitor.

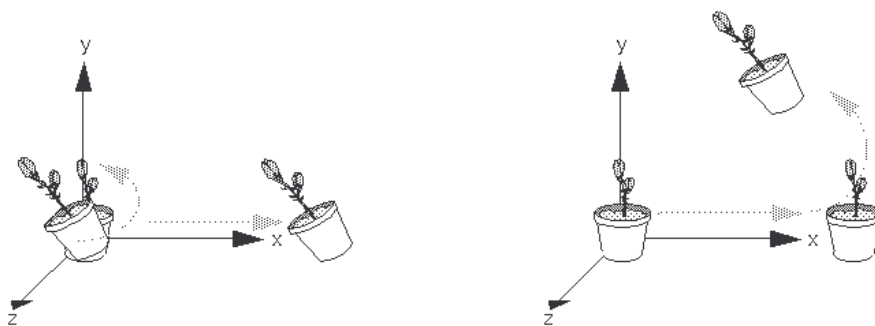


Una scheda grafica di ultima generazione svolge direttamente tutto il lavoro e alla CPU non rimane che controllare l'esecuzione.

Un oggetto in uno spazio tridimensionale è descrivibile attraverso forma e colore. Per riportare queste informazioni nello spazio bidimensionale finito del monitor e poterlo memorizzare ed elaborare, si dovrà discretizzare ed esprimere numericamente i due parametri.

Dato un sistema di assi cartesiani la forma è riproducibile campionando un sottoinsieme di punti significativi della descrizione dell'oggetto, memorizzabili come vertici  $(x, y, z)$ , la superficie sarà descritta raggruppando i vertici in triangoli.

Definita staticamente la forma, nella maggior parte delle applicazioni sopra descritte troviamo il concetto di movimento, che può essere una rotazione o una traslazione. Dato l'insieme dei vertici, queste trasformazioni avvengono moltiplicando o sommando i valori per opportune matrici.

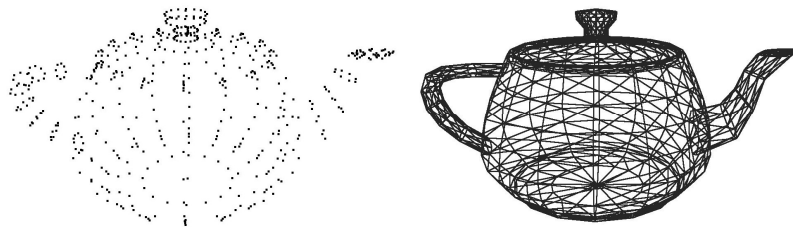


**Figura 4** Rotazione e traslazione ( sinistra ) traslazione e rotazione ( destra )

Rimanendo ancora nello spazio tridimensionale, a seconda della dimensione dell'oggetto rispetto all'area di rappresentazione, ad esempio la dimensione del monitor, ogni triangolo è suddiviso in quadratini detti fragment ognuno dei quali memorizza il valore colore relativo ed il livello di trasparenza discretizzandolo nella forma RGBA ( red, green, blue, alfa ).

Spesso gli oggetti hanno un tema cromatico ripetitivo, si pensi ad esempio ad un prato. A tale scopo si è introdotto il concetto di texture. La texture non è altro che un'immagine bidimensionale di dimensioni ridotte che riproduce un tema di colorazione che viene poi applicato ripetutamente fino a coprire tutta la

forma dell'oggetto interessato, una sorta di carta da parati. In questo caso il fragment assume il colore della porzione di texture coincidente.



**Figura 5** Vertex Image e Primitive Image



**Figura 6** Texture e Textured Image

Una volta definita la scena attraverso i fragment, vi è la proiezione sul piano bidimensionale dello schermo, alcuni saranno esclusi perché non visibili, altri fusi tra loro perché coincidenti, i risultanti saranno memorizzati sotto forma di pixel.

## **2.2 Differenze architetturali tra CPU e GPU**

Il progresso tecnologico ha permesso ai progettisti di processori di incorporare enormi risorse computazionali nei loro ultimi chip.

Oggi i processori sono costruiti da milioni di transistor connessi tra loro, collocati in spazi sempre più ridotti.

L'architettura di un processore indica il ruolo di ogni singolo transistor classificabile in tre categorie di utilizzo:

- controllo
- datapath
- storage

La sezione di controllo gestisce l'esecuzione delle istruzioni, l'aggiornamento del program counter e quindi i salti condizionali, del registro delle istruzioni e degli altri registri

Il datapath è quella preposta al calcolo, gestisce le operazioni logiche AND OR NOT e lo shift dei bit.

Lo storage è composto dai vari registri e dalla cache interna.

L'obiettivo è quindi di tradurre l'incremento di capacità in un incremento di prestazioni, suddividendo i transistor a disposizione, comunque limitati, tra le tre categorie, in base alle funzioni che il chip dovrà svolgere.

Nonostante CPU e GPU si basino sulla stessa tecnologia costruttiva, sono diverse nelle funzioni svolte, quindi nell'architettura e di conseguenza nella distribuzione dei transistor tra le tre categorie.

Un generico programma che gira su CPU necessita raramente di parallelismo; il sottoutilizzo del recente supporto al SIMD (Single Instruction, Multiple Data) dato da SSE (Streaming SIMD Extensions), ne è la dimostrazione.

Al contrario, spesso la CPU si trova a dover gestire complesse funzionalità di controllo come branch prediction ed Out-of-order execution.

Inoltre data la genericità dei programmi da eseguire le CPU non dispongono di dispositivi particolarmente specializzati su singole funzioni.

Infine è da sottolineare la fondamentale importanza nel ridurre la latenza nell'accesso alla memoria utilizzando parte dei transistor come memoria di cache. Infatti tipicamente un generico programma richiede numerosi accessi ad una ridotta quantità di memoria non contigua.

La GPU nasce dall'esigenza di alleggerire il carico di lavoro svolto dalla CPU nelle applicazioni 3D rendendole più fluide e realistiche.

Per quanto possa essere complessa, una scena in 3D si può ridurre ad una serie di triangoli in uno spazio tridimensionale, proiettati sul piano bidimensionale dello schermo, la cui superficie assume colorazioni nello spazio RGBA. Il movimento di un oggetto si realizza applicando ai vertici che lo rappresentano matrici di traslazione, scala, rotazione.

Ciò che rende realistica una scena sono:

- Il numero di triangoli che si possono gestire, di conseguenza la granularità degli oggetti.
- La quantità di parametri utilizzabili per il calcolo della colorazione finale di ogni pixel.
- La fluidità dei movimenti, quindi la velocità di calcolo ed ovviamente la risoluzione grafica.

Le funzioni svolte da una GPU sono minori, particolarmente specializzate, gestiscono un flusso lineare di una grande mole di dati contigui, su cui operare la stessa singola istruzione.

Ne risulta che mentre nell'architettura di una CPU si dia largo spazio ai transistor di controllo e di storage a dispetto del calcolo, nell'architettura di una GPU le scelte sono diametralmente opposte.

Avendo un flusso voluminoso di dati contigui cui spesso si accede un'unica volta, non vi sono grossi problemi di latenza e caching; il controllo del flusso dei dati è minimo, quindi gran parte delle risorse sono dedicate al calcolo. Valutando poi il tipo di operazioni che una GPU andrà a svolgere, si intuisce che la sua architettura avrà una forte connotazione parallela.

La forte richiesta su larga scala di tali prodotti, con prestazioni sempre maggiori, da parte del mercato dell'intrattenimento, ha portato all'introduzione della programmabilità che consente di aggiungere effetti grafici più complessi e molto apprezzati.

Inoltre il notevole successo di tali dispositivi ha aumentato esponenzialmente l'attività di sviluppo e ricerca, immettendo sul mercato GPU sempre più potenti ad un prezzo relativamente contenuto.

## 2.3 La pipeline grafica

La GPU rientra nella categoria degli stream processor, le risorse di calcolo sono disposte lungo una pipeline, una sequenza di fasi SIMD operanti contemporaneamente ed in ordine fisso. Ogni fase riceve in input ciò che la fase precedente genera come output.

Le architetture proposte dai produttori per le varie generazioni di GPU sono differenti, qui di seguito proporremo l'architettura NV40 proposta dalla NVIDIA per le schede della serie 6 e 7, le ultime in ordine di tempo.

Le architetture delle versioni precedenti sono solo delle versioni ridotte e quelle di produttori differenti seguono la stessa logica.

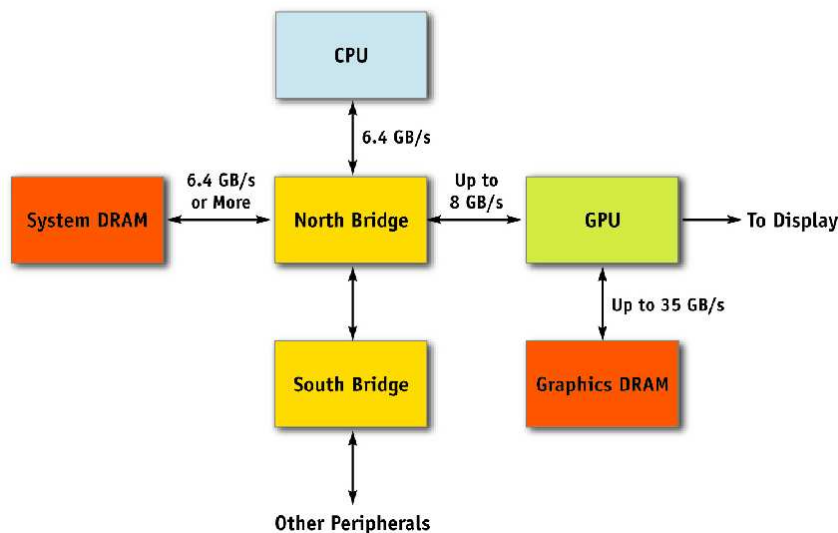


Figura 7 Sistema di comunicazione in un sistema dotato di GPU

La CPU comunica con la GPU tramite un connettore grafico, gli standard sono principalmente due, PCI Express e AGP. Dato che il connettore grafico è responsabile del trasferimento dei comandi, texture e dati relativi ai vertici dalla CPU alla GPU, la tecnologia del bus grafico è evoluta parallelamente alle GPU negli ultimi anni. L'originario slot AGP aveva una frequenza di 66 Mhz ed una banda di 32 bit, con un transfer rate asimmetrico di 264 MB/sec in input e molto inferiore in output, dato che l'output principale viene trasferito alla memoria video visualizzata su display. I successivi AGP 2x, 4x e 8x hanno raddoppiato la banda disponibile, fino a quando è stato introdotto lo standard

PCI Express nel 2004, con una banda massima teorica di 4 GB/sec in ingresso e 4 GB/sec in uscita contemporaneamente.

Le attuali schede madri supportano connettori grafici che in realtà non superano i 3.2 GB/sec.

E' importante notare la grande differenza tra la larghezza di banda dell'interfaccia di memoria della GPU e quella in altre parti del sistema, come descritto in tabella 1.

<i>Componente</i>	<i>Banda passante</i>
<i>GPU memory Interface</i>	<i>35 GB/sec</i>
<i>PCI Express Bus ( 16x )</i>	<i>8 GB/sec</i>
<i>CPU Memory Interface (800 Mhz Front-Side Bus)</i>	<i>6.4 GB/sec</i>

**Tabella 1 Transfer rate in varie zone del sistema**

Già questa tabella ci fa intuire che un algoritmo che gira su una GPU può trarre grande vantaggio dalla velocità di banda e ottenere un notevole incremento di prestazioni.

Descriviamo ora l'architettura di una GPU attraversando tutta la pipeline, partendo dall'input fornito dalla CPU fino all'output dei pixel memorizzati sul frame buffer.

Le fasi che la costituiscono la pipeline grafica sono:

- VERTEX TRASFORMATION
- PRIMITIVE ASSEMBLY
- RASTERIZZAZIONE
- INTERPOLAZIONE DEL COLORE E TEXTURING
- FRAGMENT OPERATIONS E TEXTURING
- Z-COMPARE, BLENDING
- FRAME BUFFER

I comandi, le texture ed i dati relativi ai vertici sono inviati dalla CPU attraverso un buffer condiviso in memoria o un frame buffer locale.

Le istruzioni inviate dalla CPU inizializzano e modificano i registri di stato ed attraverso comandi di rendering, passano le texture e i vertici. I comandi vengono decodificati ed una vertex fetch unit legge i valori relativi ai vertici. I comandi, i vertici ed i cambi di stato fluiscono verso le successivi stadi della pipeline.

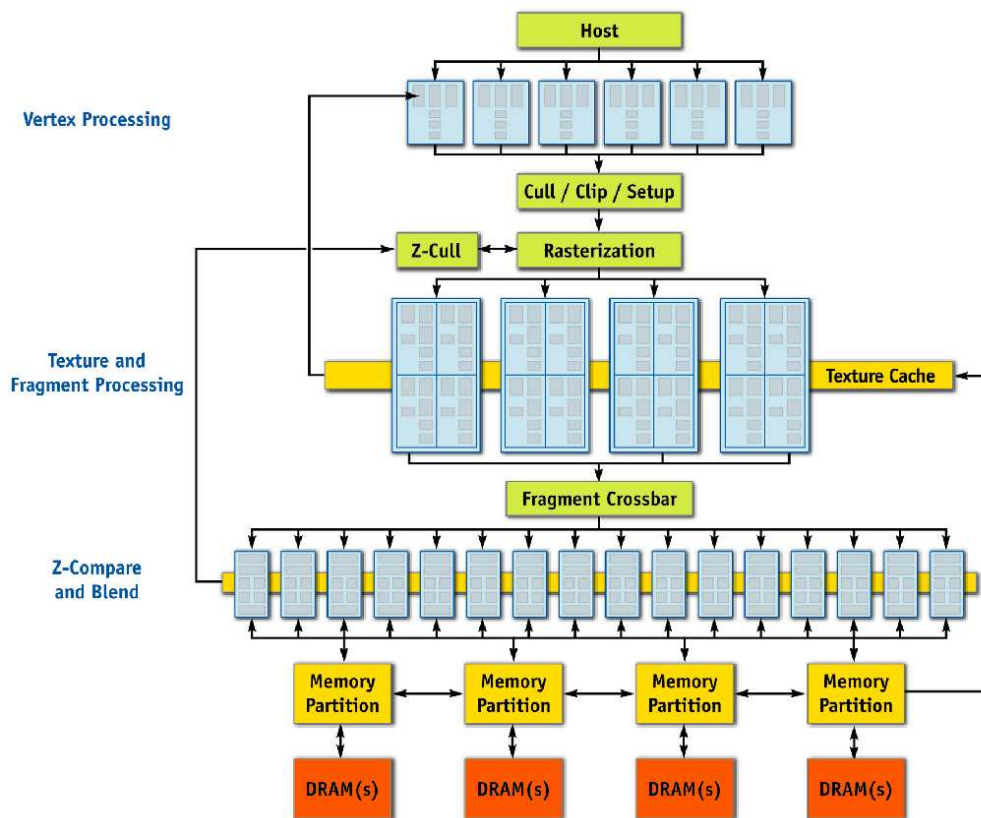


Figura 8 Pipeline grafica NV40 di NVIDIA

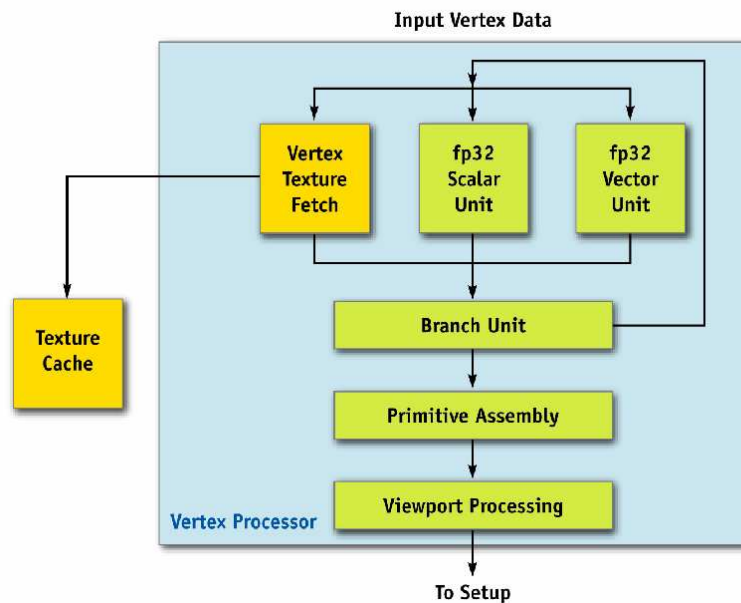
### 2.3.1 Vertex transformation

I vertex processor, chiamati anche vertex shader, sono la prima unità programmabile.

Il programma definito shader è eseguito indipendentemente da più unità di calcolo su ogni vertice in parallelo, applicando tutte le trasformazioni nello spazio tridimensionale che l'utente ha definito.

L'input è costituito dall'insieme della posizione dei vertici con relativo colore, vettore normale, necessario per calcolare e simulare l'illuminazione, le coordinate di adiacenza di eventuali texture da applicare. A questi parametri fondamentali che se non vengono passati saranno inizializzati a zero, si possono affiancare parametri aggiuntivi, come l'angolo di rotazione, o vari parametri di deformazione.

L'output prodotto ovviamente sarà della stessa natura dell'input.



**Figura 9** Vertex processor

Nelle più recenti schede il vertex program può accedere anche alle texture, cosa non prevista precedentemente, perché i programmatori avevano iniziato ad utilizzare le texture come struttura dati per passare parametri di calcolo e non più solo mappe grafiche al fragment program. Le nuove schede hanno assorbito questa tecnica alternativa e l'hanno resa possibile anche al vertex program che non sarebbe preposto a gestire mappe grafiche quali le texture. Tutte le operazioni sono in virgola mobile a 32 bit. L'architettura è fortemente



scalabile per aderire alle varie esigenze di potenza e di prezzo sul mercato. A seconda dei modelli si possono avere da una a otto vertex processor in parallelo e la tendenza è in crescita.

Dato che il vertex processor può accedere alle texture, condivide la texture cache con il fragment processor. In aggiunta vi è una vertex cache che memorizza i vertici sia prima che dopo le trasformazioni riducendo i tempi di latenza. Questo significa che se un vertice occorre più volte in una trasformazione, ad esempio nella definizione di una forma attraverso triangoli in cui un vertice è comune a più triangoli, come in un tetraedro, la lettura dei dati è velocizzata.

### **2.3.2 Primitive assembly**

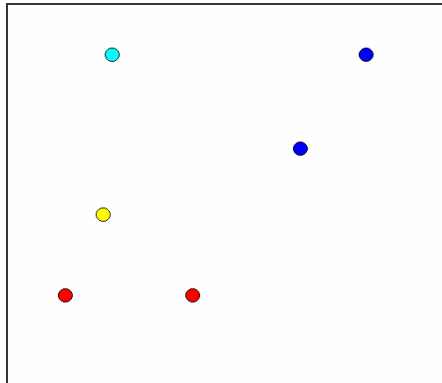
I vertici sono poi raggruppati in primitive: punti, linee e triangoli. La fase di Cull/Clip/Setup esegue le operazioni pre-primitiva, rimuovendo le primitive che non sono visibili perché dietro la visuale e ritagliando le primitive che intersecano il riquadro visualizzabile.

### **2.3.3 Rasterizzazione, interpolazione del colore e texturing**

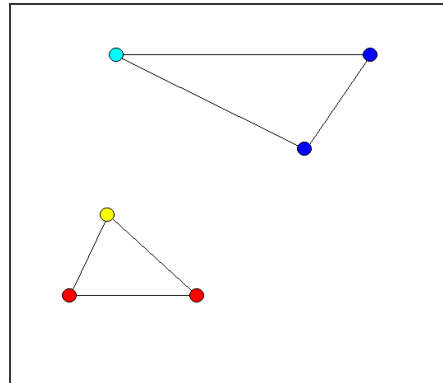
Si è ora pronti alla fase di rasterizzazione. Il blocco di rasterizzazione calcola quanti fragment compongono ogni primitiva, ed usa l'unità z-cull per scartare i pixel che sono occlusi da oggetti meno in profondità. I fragment sono colorati interpolando il colore dei vertici e si calcola la deformazione della texture solitamente regolare, in base alla primitiva.

Si pensi ad un fragment come ad un “possibile” pixel: il fragment passa attraverso il fragment processor che eseguirà diversi test, se saranno tutti

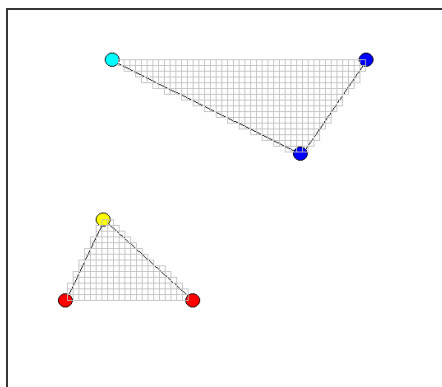
positivi, il suo colore le texture coincidenti e la profondità definiranno un pixel nel frame buffer o nel render target.



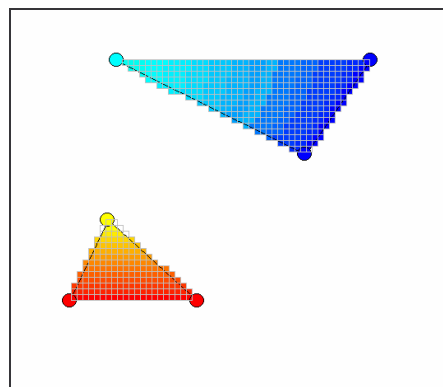
**Figura 10 Vertici con parametro colore**



**Figura 11 Primitive assembly**



**Figura 12 Rasterizzazione**



**Figura 2-13 Interpolazione**

### **2.3.4 Fragment operations e texturing**

Una volta che le primitive sono rasterizzate in un insieme di fragment, questa unità ne interpola i parametri come necessario tramite operazioni matematiche e di texturing che determineranno il colore finale per ognuno di essi.

Inoltre tale unità può determinare un nuovo valore di profondità o evitare che il valore precedente del colore del fragment venga aggiornato.

Il fragment processor è composto da un'unità di texturing e dalla fragment unit, che cooperano nell'eseguire il fragment program, per ogni fragment

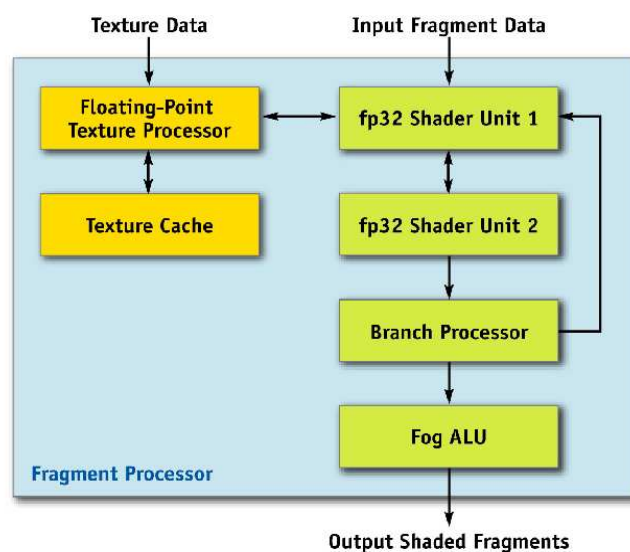
indipendentemente. L'architettura anche in questo caso è scalabile e si possono avere da una a ventiquattro fragment processor in parallelo. Comunemente si dice che la GPU ha un numero variabile di fragment pipeline.

Come per il vertex processor, le texture sono memorizzate su una cache nel chip per ridurre le comunicazioni con la memoria centrale ed aumentare le prestazioni.

La texture unit ed il fragment unit operano su matrici di quattro pixel alla volta, chiamati quads. Quindi l'insieme dei fragment processor possono gestire fino a cento pixel alla volta secondo il paradigma SIMD.

Il fragment processor usa la texture unit per leggere i dati dalla memoria. La texture unit supporta vari formati grafici che vengono passati alla fragment unit in floating-point a 16 bit o 32 bit.

Una texture può essere vista come un array bidimensionale o tridimensionale, che può essere letto dalla texture unit in una locazione arbitraria.



**Figura 14** Fragment processor

Ogni fragment processor ha due unità di calcolo floating-point a 32 bit. I fragment attraversano entrambe poi passano al branch processor prima di ciclare di nuovo la fragment pipeline che eseguirà le successive istruzioni del fragment program. L'intera pipeline è attraversata in un ciclo di clock, quindi in

generale è possibile eseguire otto o più operazioni matematiche per ciclo di clock, quattro se vi è una lettura da texture, che sarà eseguita dalla prima unità.

### 2.3.5 Z-compare e blending

I fragment lasciano i fragment processor nell'ordine in cui sono stati rasterizzati e fluiscono all'unità z-compare e all'unità blend, che eseguono vari test, quindi il colore finale viene scritto sul render target o sul framebuffer.

Lo scissor test elimina i fragment che cadono fuori dal rettangolo specificato dal framebuffer. Raramente questo test è utile per applicazioni general purpose, dato che tipicamente queste utilizzano rettangoli specificatamente dimensionati per l'elaborazione richiesta.

Il test successivo è il confronto della profondità del fragment con il corrispondente nel framebuffer.

Dopo di che, facoltativamente si può testare e modificare lo stencil buffer, che memorizza un valore intero per ogni pixel. Questo buffer era originariamente previsto per permettere ai programmatori di mascherare certi pixel, per esempio nei videogame, per restringere il disegno al parabrezza di un'auto, ha poi trovato altri utilizzi.

Infine vi è la funzione di blending o più comunemente detta "effetto nebbia".

La fog unit è utilizzata per aggiungere l'effetto nebbia alla scena, senza perdita di performance. L'effetto nebbia si usa spesso in molte applicazioni grafiche specie nei video giochi. La funzione è fissa e non supporta il floating-point, ma in alcuni casi potrebbe essere di qualche utilità nel calcolo general purpose.

$$\text{Out} = \text{FogColor} * \text{FogFraction} + \text{SrcColor} * (1 - \text{FogFraction})$$

La memoria è divisa in quattro partizioni indipendenti, ognuna costituita da un modulo DRAM. Le GPU usano i moduli DRAM piuttosto che tecnologie dedicate per trarre vantaggio dai prezzi di mercato e ridurre i costi.

Avere partizioni piccole ed indipendenti permette al sistema di operare efficientemente sia nel caso di grandi o piccoli blocchi di dati da trasferire. Tutte le immagini renderizzate sono memorizzate sulla DRAM, mentre le texture e i vertici di input possono essere memorizzati sia su DRAM che sulla memoria centrale, quest'ultimo caso è previsto per le GPU più economiche. Le quattro partizioni di memoria indipendenti costituiscono un flessibilissimo sistema interno di memoria che permette un accesso vicino al limite fisico di 35GB/sec.

## Capitolo 3

### Dispositivi hardware

I produttori che hanno investito di più sulle architetture grafiche programmabili sono ATI e NVIDIA. Ognuna con pregi e difetti, la NVIDIA ha prestazioni migliori se si utilizzano le OpenGL, mentre l'ATI ha un supporto migliore per le DirectX, ma in definitiva sono molto simili. Per ovvi motivi, i loro prodotti sono classificati in base allo Shader Model supportato, che corrisponde alle versioni di DirectX a partire dalla 8.0, cui si può far riferimento per comodità. Di seguito si farà una panoramica dell'ultima generazione.

Shader Model	DirectX	NVIDIA GeForce		ATI Radeon	
		Nome commerciale	Architettura	Nome commerciale	Architettura
Shader Model 1.0	DirectX 8.0	Serie 3 e 4	NV20	8500-9200	R200
Shader Model 2.0	DirectX 9.0	Serie FX	NV30	R95xx-R98xx X3xx-X8xx	R300-R400
Shader Model 3.0	DirectX 9.0c	Serie 6 e 7	NV40	X1300-X1800	R500

**Tabella 2** Classificazione delle GPU per Shader Model

### 3.1 NVIDIA GeForce serie 6

La GeForce Serie 6 è la sesta generazione di processori grafici NVIDIA. Tutte le schede di questa generazione supportano lo Shader Model 3.0.

Lanciata nell'aprile del 2004, con uscite scaglionate per modello e con tecnologia costruttiva a 130 o 110 nanometri, la GeForce 6 introduce importanti funzionalità alla linea GeForce, tra le quali proprio lo Shader Model 3.0 e la tecnologia SLI non supportata in alcuni modelli.

La famiglia delle GeForce 6 ha eliminato le imperfezioni della precedente GeForce FX con scarse prestazioni nel eseguire le funzionalità dello Shader Model 2.0. Quindi, sia dal punto di vista commerciale che tecnico la GeForce 6 ha permesso a NVIDIA di recuperare una posizione di competitività contro i concorrenti diretti della ATI.

Le schede in varie versioni sono classificate nei gruppi:

- GeForce 6800
- GeForce 6600
- GeForce 6500
- GeForce 6200
- GeForce 6100

Il gruppo delle GeForce 6800 è stato il primo della serie, prodotto destinato al mercato dei videogiochi che richiedono più potenza. Il primissimo modello fu la GeForce 6800 Ultra, più veloce del 50-80% rispetto al top di gamma precedente la GeForce 5950 Ultra.

La GeForce 6600 è stata realizzata nell'agosto 2004, con la metà di pipeline ed un bus di memoria ridotto a 128 bit, l'architettura rimane invariata in generale e si è mantenuto il supporto SLI, si è inoltre passati ad una tecnologia costruttiva a 110 nanometri che ha ridotto ulteriormente i costi. Il bus è nativo PCI Express, con la possibilità di bridge per le versioni AGP. Le performance sono leggermente migliori della GeForce FX 5950 Ultra.

La GeForce 6500 è stata realizzata nell'ottobre 2005 basata sull'architettura della GeForce 6200 con 3 vertex processor e 4 fragment processor, ma con un core clock maggiore e con più memoria video ed il supporto SLI.

La GeForce 6100/6150 è stata realizzata alla fine del 2005. Questa è una GPU integrata nella scheda madre nForce4 in competizione con ATI RS480 ed Intel GMA 900. Ovviamente come nel caso della GeForce 6500 non è possibile il supporto SLI, è dotata di 1 vertex processor e 2 fragment processor.

### **3.2 NVIDIA GeForce serie 7**

La GeForce Serie 7 è la settima e più recente generazione di schede grafiche NVIDIA GeForce. Tutte le schede di questa generazione supportano lo Shader Model 3.0 e offrono il supporto alla tecnologia SLI.

Le schede in varie versioni sono classificate nei gruppi:

- GeForce 7800
- GeForce 7200
- GeForce 7600

La GeForce 7800 GTX è la prima GPU della serie 7, lanciata nel giugno 2005. Il chip ha il supporto nativo PCI Express, ma può usare un bridge chip per la versione AGP.

La maggiore differenza rispetto alla GeForce serie 6 è l'aumento del numero di fragment processor passato da 16 a 24. Tuttavia il numero di Raster Operation Pipeline ( ROP ) rimane 16, ciò indica che NVIDIA pensa che incrementare il numero di fragment generati dal rasterizer al secondo, sia meno importante che supportare più operazioni per fragment. Ogni pipeline è stata migliorata per aumentare le prestazioni nel caso di fragment program molto complessi. Il numero di vertex processor è stato anch'esso aumentato a 8 dai precedenti 6.



Di questa scheda esiste anche una versione con 512MB di memoria video, anche la frequenza di clock è stata aumentata e si è passato a moduli di memoria GDDR3 con un clock di 1.7GHz contro i 1.2GHz della GDDR1.

Si ritiene che questo processore sia il più complesso e potente attualmente in commercio, con 300 milioni di transistor ( un AMD Athlon 64 2800+ ha 105.9 milioni di transistor ).

La GeForce 7800 GT è la seconda GPU della serie, presentata nell'agosto 2005. Ha 20 fragment processor e 7 vertex processor, 16 raster operation pipeline con 400Mhz di core clock e 500Mhz di memory clock che può essere raddoppiato utilizzando moduli GDDR3.

Questa GPU è stata sviluppata come alternativa più economica alla precedente con un rapporto costo/prestazioni notevole.

Intanto si prevede per il 2006 l'utilizzo del processo produttivo a 90 nanometri con l'uscita della GeForce 7900 con miglioramenti architetturali ed un prezzo rivisto.

La GeForce 7200 e la GeForce 7600 di imminente uscita sono versioni con architettura identica alla capostipite, ma con un ridotto numero di unità di calcolo si pensa che saranno 12 fragment processor per la prima e 8 per la seconda, entrambe con un bus meno veloce. Andranno ad occupare le fasce più basse del mercato.

### **3.3 ATI Radeon serie X1000**

Le schede ATI Radeon sono state le prime a presentare il supporto allo Shader Model 2.0, ma hanno tardato ad inserire il supporto alla versione 3.0, disponibile solo dall'uscita della serie X1000. La Radeon serie X1000 è stata presentata nell'ottobre 2005 in tutte le sue versioni e sostituisce la ATI Radeon X850. Tutta la serie è stata realizzata con processo produttivo a 90 nanometri.

Il supporto Crossfire, equivalente del SLI, al momento attuale, è una tecnologia limitata nell'implementazione, non solo dal punto di vista prestazionale quando confrontata con le soluzioni NVIDIA GeForce 7800 in modalità SLI.

Le schede in varie versioni sono classificate nei gruppi:

- X1300
- X1600
- X1800

Radeon X1800 è per le soluzioni top di gamma; Radeon X1600 per quelle di fascia media e Radeon X1300 per le proposte entry level.

La famiglia Radeon X1800 ha un architettura con 16 pipeline di rendering e bus di memoria a 256bit; la scheda X1800XT viene fornita in due distinte versioni, con 256 oppure 512 Megabyte di memoria video.

La famiglia di schede Radeon X1600 vede l'utilizzo di un chip con 12 pipeline di rendering, in abbinamento a bus memoria a 128bit di ampiezza. Due sono i modelli disponibili, ciascuno configurabile con 128 oppure 256 Megabyte di memoria video.

La terza famiglia di schede video è quella Radeon X1300 che però non supporta lo Shader Model 3.0. Il chip ha in questo caso 4 pipeline di rendering, in abbinamento ad un bus memoria da 128bit di ampiezza. La scheda PRO è disponibile solo con 256 Megabyte di memoria video, mentre la scheda Radeon X1300 è disponibile in versioni con 128 oppure 256 Megabyte di memoria video.

Per tutte le schede il connettore grafico è PCI Express, anche se dovrebbero essere in uscita le versioni AGP con bridge.

Le tabelle seguenti sintetizzano quali sono le caratteristiche tecniche delle nuove soluzioni ATI, a confronto con le soluzioni concorrenti di NVIDIA e con le schede video ATI sino ad ora commercializzate.

	ATI	NVIDIA	ATI	NVIDIA	NVIDIA	ATI
	X1800XT	7800 GTX	X1800XL	7800 GT	6800 Ultra	X850XT PE
Bus di memoria	256 bit	256 bit	256 bit	256 bit	256 bit	256 bit
Processo produttivo	0.09 micron	0.11 micron	0.09 micron	0.11 micron	0.13 micron	0.13 micron
Frequenza chip	625 Mhz	430 Mhz	500 Mhz	400 Mhz	400 Mhz	540 Mhz
Frequenza memoria	1500 Mhz	1200 Mhz	1000 Mhz	1000 Mhz	1100 Mhz	1180 Mhz
Unità di Vertex Shading	8	8	8	7	6	6
Unità di Pixel Shading	16	24	16	20	16	16
Numero di pipeline	16	24	16	20	16	16
Texture per ciclo di clock	1	1	1	1	1	1
Fill Rate	10000 MPixel	10320 MPixel	8000 pixel	8000 MPixel	6400 MPixel	8640 MPixel
Banda Passante	48 GB	38,4 GB	32 GB	32 GB	35,2 GB	37,7 GB
Versione Vertex Shader	3.0	3.0	3.0	3.0	3.0	2.0
Versione Pixel Shader	3.0	3.0	3.0	3.0	3.0	2.0b

	ATI	ATI	ATI	NVIDIA	NVIDIA
	X1600XT	X1600XL	X700PRO	6800	6600 GT
<b>Bus di memoria</b>	128 bit	128 bit	128 bit	256 bit	128 bit
<b>Processo produttivo</b>	0.09 micron	0.09 micron	0.11 micron	0.13 micron	0.11 micron
<b>Frequenza chip</b>	590 Mhz	500 Mhz	425 Mhz	325 Mhz	500 Mhz
<b>Frequenza memoria</b>	1380 Mhz	780 Mhz	864 Mhz	700 Mhz	1000 Mhz
<b>Unità di Vertex Shading</b>	5	5	6	6	3
<b>Unità di Pixel Shading</b>	12	12	8	12	8
<b>Numero di pipeline</b>	12	12	8	12	8
<b>Texture per ciclo di clock</b>	1	1	1	1	1
<b>Fill Rate</b>	7080 MPixel	6000 MPixel	3400 MPixel	3900 MPixel	4000 MPixel
<b>Banda Passante</b>	22,08 GB	12,5 GB	13,8 GB	22,4 GB	16 GB
<b>Versione Vertex Shader</b>	3.0	3.0	2.0	3.0	3.0
<b>Versione Pixel Shader</b>	3.0	3.0	2.0b	3.0	3.0

	ATI	ATI	ATI	ATI	ATI	NVIDIA
	X1300PRO	X1300	X600 PRO	X550	X300 SE HM	GeForce 6200
<b>Bus di memoria</b>	128 bit	128 bit	128 bit	128 bit	64 bit	128 bit
<b>Processo produttivo</b>	0.09 micron	0.09 micron	0.13 micron	0.11 micron	0.11 micron	0.11 micron
<b>Frequenza chip</b>	600 Mhz	450 Mhz	400 Mhz	400 Mhz	325 Mhz	300 Mhz
<b>Frequenza memoria</b>	800 Mhz	500 Mhz	600 Mhz	500 Mhz	600 Mhz	550 Mhz
<b>Unità di Vertex Shading</b>	2	2	2	2	2	3
<b>Unità di Pixel Shading</b>	4	4	4	4	4	4
<b>Numero di pipeline</b>	4	4	4	4	4	4
<b>Texture per ciclo di clock</b>	1	1	1	1	1	1
<b>Fill Rate</b>	2400 MPixel	1800 MPixel	1600 MPixel	1600 MPixel	1300 MPixel	1200 MPixel
<b>Banda Passante</b>	12,8 GB	8 GB	9,6 GB	8 GB	4,8 GB + 8 GB	8,8 GB
<b>Versione Vertex Shader</b>	2.0	2.0	2.0	2.0	2.0	3.0
<b>Versione Pixel Shader</b>	2.0	2.0	2.0	2.0	2.0	3.0

## Capitolo 4

### Linguaggi

I principali linguaggi di shading, quelli cioè preposti alla stesura di vertex program e fragment program sono principalmente quattro:

- Cg
- HLSL
- OpenGL Shading Language
- Renderman

Ad essi si aggiungano Sh e Brook che non sono linguaggi veri e propri ma dei metalinguaggi.

#### 4.1 I metalinguaggi

I metalinguaggi non sono linguaggi veri e propri ma delle librerie che incluse in un linguaggio di programmazione aggiungono nuovi comandi di facile lettura e scrittura che facilitano il lavoro del programmatore. Sia Sh che Brook sono metalinguaggi che si appoggiano al C++, e permettono l'utilizzo della GPU come coprocessore.

La differenza principale tra Sh e Brook è che il primo si propone per un utilizzo grafico, ma facilita la stesura anche di applicazioni general purpose mentre Brook è specializzato per la general purpose computation.

Inoltre Brook ha un suo compilatore che traduce il codice in Cg che poi deve essere ricompilato, mentre Sh viene compilato direttamente insieme al codice C++, saltando questa fase intermedia. Possono essere molto utili se si vuole utilizzare la GPU come coprocessore matematico senza approfondirne i dettagli e senza pretendere prestazioni ottimali.

## 4.2 I linguaggi di shading

Esiste uno strettissimo legame tra i linguaggi di shading e i dispositivi su cui dovranno girare, dato dalla relativa rigidità delle funzionalità offerte da questo tipo di hardware e lo scopo che i linguaggi si propongono cioè la stesura di programmi che elaborino trasformazioni geometriche più o meno complesse e effetti grafici dovuti al calcolo del colore finale di ogni pixel.

Le differenze spesso sono puramente sintattiche mentre l'insieme delle funzioni disponibili praticamente identico. Come si è già accennato il set di istruzioni ed i paradigmi di programmazione utilizzabili sono dipendenti direttamente dall'hardware e quindi dallo Shading Model supportato.

Sono quindi linguaggi di programmazione dedicati. Questo li rende più semplici, rispetto ad un moderno linguaggio general purpose come il C++.

Essendo specializzati nella trasformazione di vertici e fragment, non prevedono molte delle complesse funzionalità necessarie per sviluppare software come word editor, DBMS o sistemi operativi. Diversamente dal C++ e dal Java, non supportano le classi o la programmazione ad oggetti, i puntatori, l'allocazione di memoria o la gestione di file, ma in alcuni è già previsto che parte di queste restrizioni andranno eliminate in accordo con la definizione di nuove architetture che lo rendano possibile.

Gestiscono array ed è possibile definire strutture, prevedono controlli di flusso quali le condizioni, i cicli e infine le chiamate di funzioni.

Data la natura dei dati che devono gestire, il supporto a vettori e matrici e relativi operatori matematici sono fortemente ottimizzati ed è prevista tutta una serie di funzioni tipiche dell'ambiente grafico.

I programmi sono eseguiti in relativo isolamento. Questo significa che l'elaborazione di un dato vertice o fragment non influisce sugli altri.

Questo apparente limite rende i programmi fortemente parallelizzabili.

Da questa linea si discosta esclusivamente Renderman, il primo linguaggio di shading mai concepito che però è statico, cioè il suo scopo è quello di renderizzare immagini statiche, il che limita notevolmente le possibilità nel caso di programmazione general purpose.

### **4.3 Implementazione delle strutture dati su GPU**

Gli streaming processor come le GPU sono programmati in maniera diversa da un processore seriale come le CPU. Molti programmatori sono abituati ad un modello di programmazione in cui possono scrivere in qualsiasi locazione di memoria in qualsiasi punto del loro programma. Quando si programma uno streaming processor, invece, accediamo alla memoria in maniera molto più strutturata. Nello stream model, i programmi sono espressi come serie di operazioni su flussi di dati. Gli elementi di uno stream o flusso, ovvero un vettore ordinato di dati, sono processati dalle istruzioni del kernel ( nel nostro caso il vertex program o il fragment program ). Il kernel opera su ogni elemento del flusso indipendentemente.

Le restrizioni dello stream programming model permettono alla GPU di eseguire il kernel in parallelo e quindi processare molti dati simultaneamente. Questo parallelismo dei dati è reso possibile dalla certezza che il calcolo su di un elemento non può influire l'elaborazione degli altri elementi dello stesso stream. Conseguentemente, i soli valori che vengono usati nell'elaborazione di un kernel sono gli input del kernel e le letture da memoria. Inoltre, la GPU richiede che anche l'output del kernel sia indipendente: non sono possibili scritture random in memoria, possono scrivere solamente in una specifica singola posizione nel flusso di output corrispondente alla posizione nel flusso in input da dove abbiamo letto i dati. Il parallelismo dei dati di questo modello è fondamentale e principale motivo della differenza di prestazioni rispetto alle altre architetture.



## 4.4 Il modello di memoria della GPU

I processori grafici hanno la loro gerarchia di memoria analoga a quella usata dai microprocessori seriali:

- Memoria centrale
- Cache
- Registri

Questa gerarchia, tuttavia, è progettata per accelerare le operazioni grafiche tipiche dello streaming programming model. Inoltre le interfacce grafiche come OpenGL e Direct3D limitano ulteriormente l'uso della memoria a specifiche primitive come vertici, texture e framebuffer.

La GPU come la CPU ha la propria cache ed i propri registri per accelerare l'accesso ai dati durante l'elaborazione, ma ha anche la propria memoria principale con il suo spazio di indirizzi. Questo permette al programmatore di copiare nella memoria della GPU i dati prima che cominci l'esecuzione. Questo trasferimento è stato il collo di bottiglia per molte applicazioni, ma con l'avvento del nuovo bus PCI Express la trasmissione di dati tra CPU e GPU e viceversa si è molto velocizzata, con prospettive future, nel breve periodo, ancora migliori.

Diversamente dalla memoria di una CPU, la memoria video è accessibile solo tramite astrazioni grafiche. Ognuna di queste astrazioni può essere pensata come un diverso tipo di flusso con le proprie regole di accesso.

I quattro tipi di stream sono:

- Vertex stream
- Texture stream
- Fragment stream
- Framebuffer stream

Il Framebuffer stream comincia e termina all'interno della GPU.

#### 4.4.1 Vertex streams

Il flusso di vertici è passato come un vertex buffer attraverso l'API grafica. Il flusso consiste di posizione ed una varietà di attributi pre-vertex. Questi attributi sono normalmente usati per le coordinate delle texture, il colore, il vettore normale e così via, ma essi possono essere usati per input arbitrari per il vertex program. Il vertex program non può indicizzare in modalità random i vertici in input. Prima delle ultime versioni dell'architettura grafica, il flusso di vertici non poteva che essere aggiornato dal trasferimento di dati dalla CPU alla GPU. Quest'ultima non poteva generare flussi di vertici. Le recenti API grafiche lo hanno reso possibile, attraverso le funzioni "copy-to-vertex-buffer" o "render-to-vertex-buffer" dove il risultato del rendering viene copiato dal framebuffer al vertex buffer oppure il risultato viene scritto direttamente nel vertex buffer. Queste recenti aggiunte permettono alla GPU, per la prima volta, di ciclare il flusso dalla fine all'inizio della pipeline.

#### 4.4.2 Fragment stream

Il flusso di fragment è generato dal rasterizer e costituisce l'input del fragment program, ma non è direttamente accessibile al programmatore perché creato e consumato interamente dentro la pipeline grafica. Il flusso di fragment include l'insieme dei valori interpolati generati dal vertex processor: posizione, colore, coordinate delle texture e così via. Come per gli attributi pre-vertex anche questi possono essere utilizzati per input arbitrari.

Il flusso di dati non può essere letto in posizioni casuali dal fragment program. Permetterlo causerebbe una dipendenza tra diversi elementi del flusso, contraddicendo le condizioni necessarie per il parallelismo del modello di programmazione. Se dovesse essere necessario un accesso random nell'algoritmo, lo stream dovrebbe essere prima salvato in memoria e poi convertito in un texture stream.

### **4.4.3 Frame buffer stream**

Il frame buffer stream è scritto dal fragment processor. E' normalmente costituito dai pixel che saranno visualizzati sullo schermo. L'elaborazione nella GPU usa il frame buffer anche per memorizzare i risultati intermedi dell'algoritmo. Le moderne GPU possono scrivere contemporaneamente su più frame buffer. Attualmente ogni singola unità può scrivere fino a 16 floating-point per passo di renderizzazione, con prospettive di incremento per il futuro.

Il frame buffer non può essere letto in posizioni casuali dal vertex o fragment processor, comunque può essere letto o scritto direttamente dalla CPU attraverso le API grafiche. Recentemente si è iniziato a non fare grossa distinzione tra frame buffer, vertex buffer e texture, consentendo di renderizzare direttamente su uno qualsiasi di essi.

### **4.4.4 Texture stream**

Le texture sono l'unico supporto di memoria accessibile in qualsiasi locazione dal fragment program e dal vertex program a partire dal vertex shader model 3.0. Se il programmatore ha bisogno di accedere in questa maniera ai precedenti stream può trasformarli in texture. Le texture possono essere lette e scritte sia dalla GPU che dalla CPU. La GPU scrive su di essa direttamente utilizzandola come destinazione del rendering o copiandovi il contenuto del frame buffer. Le texture possono essere monodimensionali, bidimensionali e tridimensionali e possono essere indirizzate direttamente con indici rispettivamente singoli, doppi o tripli. Una texture può essere dichiarata anche come cubo, gestita come un insieme di sei texture bidimensionali.

Vertex e fragment program costituiscono lo spazio di lavoro per i programmatori delle moderne GPU. Le capacità di questi programmi sono definite dalle operazioni aritmetiche che si possono eseguire e dalla memoria a

cui possono accedere. La varietà di operazioni permesse è sostanzialmente simile a quelle eseguibili su CPU, ma come abbiamo visto vi sono delle restrizioni nell'accesso alla memoria, necessarie a preservare il parallelismo e la conseguente maggiore velocità di elaborazione. Altre restrizioni sono, comunque, artefatti dell'evoluzione delle GPU e saranno certamente rilassate nelle future generazioni. Riassumiamo una lista delle restrizioni all'accesso alla memoria.

- Accesso diretto alla memoria centrale della CPU vietato
- Accesso a disco vietato
- Lettura random della texture memory consentito
- Lettura dei registri di costanti consentito
- Lettura e scrittura dei registri temporanei consentiti
- Lettura del vertex stream consentito al vertex program
- Lettura del fragment stream consentito al fragment program
- Scrittura nello stream consentita in posizione fissa
- Il vertex program scrive sull'output del vertex stream
- Il fragment program scrive sull'output del fragment stream

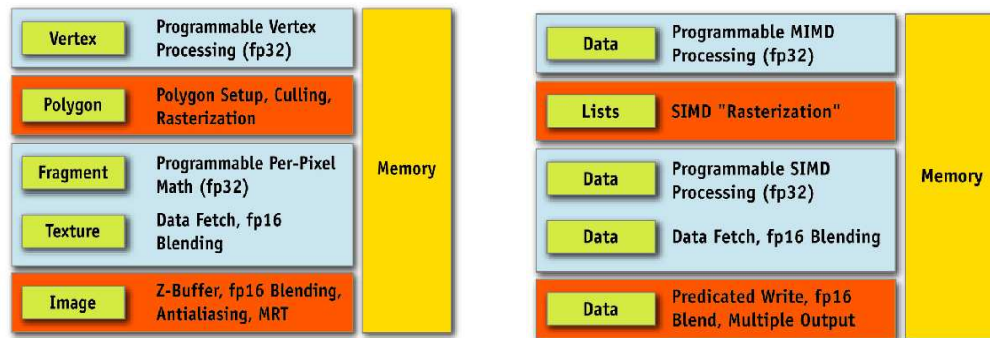
## **4.5 Programmabilità general purpose**

Le schede grafiche stanno diventando sempre più programmabili e stanno aumentando le possibilità di eseguire applicazioni non grafiche sulle GPU.

L'architettura di una GPU è organizzata per gestire oggetti grafici. E' composta da una unità programmabile per i vertici ed una per i fragment, un'unità di lettura delle texture e di una unità di scrittura.

In modo alternativo, una GPU può essere vista come un potente processore programmabile con diverse unità di calcolo floating-point ed una banda di memoria molto veloce adatta ad applicazioni non grafiche di calcolo intensivo. Quando usiamo la GPU per applicazioni non grafiche, può essere vista come due blocchi programmabili connessi serialmente: il vertex processor ed il

fragment processor, entrambi con supporto per operazioni floating-point a 32 bit; la texture unit come una unità di accesso alla memoria con una velocità di 35GB/sec.



**Figura 15** Blocchi programmabili di una GPU come strumento grafico e come strumento general purpose

Anche gli altri stadi non programmabili della pipeline grafica possono essere di utilità.

Il vertex processor elabora i dati, e li passa al rasterizer che espande i dati e li passa al fragment processor. Ogni triangolo viene trasformato in uno o più fragment

Prima che i fragment raggiungano il fragment processor, l'unità z-cull controlla la profondità, se il fragment ha una profondità inferiore al punto di osservazione della scena, il fragment non sarà visibile e quindi non sarà elaborato dal fragment processor.

Pensando con ottica general purpose, questa funzione permette di decidere velocemente attraverso una condizione basata su un valore scalare se eseguire il fragment program su di un dato.

Anche gli altri test potrebbero avere una qualche utilità nella programmazione general purpose, comunque limitata.

## 4.6 I modelli di programmazione

I modelli di programmazione prendono il nome di "Shader Model" e si suddividono in Vertex Shader e Pixel Shader in relazione ai due tipi di pipeline programmabili. Essi indicando i tipi di dati e le istruzioni utilizzabili

### 4.6.1 Shader Model 1.0

Progettato per eseguire piccoli programmi personalizzati di trasformazioni geometriche ed effetti luminosi. Contestualizzando, hanno introdotto un notevole livello di flessibilità. Al momento della presentazione, le unità grafiche con questo supporto erano programmabili solo in assembler, successivamente con le nuove versioni, qui di seguito, si sono creati diversi compilatori per altrettanti linguaggi detti di shading. Alcuni di loro prevedono la compilazione anche per questa versione. Dati i limiti imposti dall'architettura, le GPU che hanno solo questo supporto mal si prestano al calcolo general purpose. Vediamo comunque quali sono le possibilità offerte.

#### *Vertex Shader 1.0*

Il limite massimo di istruzioni per programma è di 128, si hanno a disposizione 96 registri read-only e 12 registri read-write. I floating-point sono in singola precisione. I costrutti condizionali ed i cicli sono permessi solo se possono essere risolti a tempo di compilazione e non si superi il limite di istruzioni. Non si possono gestire sottoprogrammi. Gli array non possono essere indicizzati con espressioni variabili, la gestione di matrici risulta onerosa. Non è possibile accedere alle texture.

#### *Pixel Shader 1.0*

Le operazioni possibili possono essere classificate tra istruzioni di texturing e istruzioni aritmetiche. Il limite massimo è di 4 operazioni di texturing ed 8

istruzioni aritmetiche. Inoltre le istruzioni di texturing non possono avere nessuna dipendenza dall'output di un'istruzione aritmetica, a meno che quest'ultima non sia espressa tramite “modificatori”, un set di otto operazioni aritmetiche specifiche che vengono svolte dall'hardware a tempo di esecuzione.

MODIFICATORE	ESPRESSIONE
instr_x2	$2 * x$
instr_x4	$4 * x$
instr_d2	$x / 2$
instr_sat	$\min(1, \max(0, x))$
reg_blas	$x - 0.5$
1-reg	$1 - x$
-reg	$-x$
rex_bx2	$2 * (x - 0.5)$

**Tabella 3 Modificatori**

Il floating-point è *signed clamped* cioè normalizzato nel range  $[-1, 1]$ .

Gli operatori booleani consentiti sono solo  $<$ ,  $<=$ ,  $>$ ,  $>=$ .

Non sono consentiti gli operatori bitwise.

Non è consentita la divisione a meno che il divisore non sia una costante.

Non è consentito il modulo.

I costrutti condizionali ed i cicli sono permessi solo se possono essere risolti a tempo di compilazione e non si superi il limite di istruzioni.

Gli array sono indicizzabili solo a tempo di compilazione.

## 4.6.2 Shader Model 2.0

Il Vertex Shader 2.0 introduce nelle GPU il concetto di programmabilità vera e propria. Tale possibilità è fortemente incentrata sulla grafica e il grado di libertà specie nel Vertex Shader è ancora relativamente basso.

### *Vertex Shader 2.0*

Il limite massimo di istruzioni per programma è di 256, si hanno a disposizione 256 registri read-only e 12-32 registri read-write. I floating-point sono in singola precisione.

I costrutti condizionali ed i cicli sono permessi solo se possono essere risolti a tempo di compilazione e non si superi il limite di istruzioni. Quello che cambia rispetto a prima oltre il numero di istruzioni è l'aggiunta del supporto alla gestione di sottoprogrammi e che gli array possono essere indicizzati con espressioni variabili. Le texture rimangono inaccessibili.

### *Pixel Shader 2.0*

Il limite massimo è di 32 operazioni di texturing ed 64 istruzioni aritmetiche, esiste un'estensione di questa versione che permette un limite massimo di istruzioni pari a 1024. Si hanno a disposizione 32 registri read-only e 12-32 registri read-write. I floating-point sono in singola precisione. I costrutti condizionali ed i cicli sono permessi solo se possono essere risolti a tempo di compilazione e non si superi il limite di istruzioni. Tutte le altre limitazioni sono state eliminate.

## **4.6.3 Shader Model 3.0**

Lo Shader Model 3.0 è la versione più recente, quindi la analizzeremo più in dettaglio. Sia il vertex processor che il fragment processor hanno una precisione floating-point a 32 bit, l'accesso alle texture e lo stesso set di istruzioni.



### *Vertex Shader 3.0*

Il numero di istruzioni per un vertex program è limitato, attualmente tale limite è di 512 istruzioni statiche e 65.536 istruzioni dinamiche. Il numero di istruzioni statiche rappresenta il numero di istruzioni in un programma così come viene compilato. Il numero di istruzioni dinamiche il numero di istruzioni attualmente eseguito. In pratica, il numero dinamico può essere più alto di quello statico grazie alla possibilità di utilizzare cicli e chiamate a sottoprogrammi, possibilità non prevista nelle GPU con il solo supporto allo Shader Model 2.0.

Vi sono poi 32 registri a 128 bit utilizzabili per variabili d'appoggio.

Un singolo vertex processor riesce ad eseguire una operazione di moltiplicazione ed addizione su una matrice 4x4 più una funzione scalare per ciclo di clock.

Le funzioni scalari sono:

- funzioni esponenziali: EXP, EXPP, LIT, LOG, LOGP
- funzioni reciproco : RCP, RSQ
- funzioni trigonometriche: SIN, COS

Le texture accessibili sono massimo quattro.

### *Pixel Shader 3.0*

Anche il numero delle istruzioni eseguibile dal fragment processor sono limitate, in questo caso sia il numero di istruzioni statiche che quelle dinamiche è pari a 65.535. Ci sono delle limitazioni a quanto tempo il sistema operativo aspetta che il fragment program sia eseguito, quindi un programma troppo lungo che lavora a pieno schermo potrebbe andare in time-out.

Occorre quindi tenere in forte considerazione la lunghezza del programma ed il numero di fragment renderizzati.

Ogni unità di calcolo può eseguire una somma ed un prodotto vettoriale con quattro termini più una moltiplicazione per clock. Inoltre se si riduce l'operazione a tre termini è possibile eseguire calcoli aggiuntivi sul quarto

operando. Le operazioni sono in virgola mobile e possono essere sia a 32 bit che a 16 bit, in alcuni casi si ha un miglioramento di prestazioni utilizzando la singola precisione. Dato che ogni processore dispone di due unità di calcolo si possono eseguire otto operazioni per ciclo di clock.

All'interno del fragment processor vi è anche una unità di normalizzazione e una che calcola il reciproco che lavorano in parallelo alle unità di calcolo, quindi si può normalizzare, portando un floating-point 32 bit a 16 bit e calcolare il reciproco senza perdita di prestazioni.

Inoltre si possono eseguire due operazioni indipendenti in parallelo, suddividendo i quattro operandi in due gruppi, uno per operazione. Ciò dà al compilatore l'opportunità di inserire più calcoli scalari in un unico calcolo vettoriale, guadagnando in prestazioni. Anche fragment processor può gestire cicli e sottoprogrammi.

Le istruzioni di controllo danno luogo ad overhead, cioè impiegano più cicli di clock per essere eseguite, esaminiamo in dettaglio il costo in termini di cicli di clock impiegati:

<b>Istruzione</b>	<b>Costo ( cicli di clock )</b>
if / end if	4
if / else / end if	6
call	2
ret	2
loop / end loop	4

**Tabella 4 Overhead delle istruzioni di controllo**

## 4.7 Profili

Dato che i linguaggi di shading sono dedicati, specifici per le GPU, la cui architettura è in rapida evoluzione, non tutto quello che si può scrivere è detto che sia eseguibile da una specifica scheda grafica. Da ciò la definizione del concetto di profilo hardware, che deve essere specificato al compilatore.

Ogni profilo corrisponde ad una specifica combinazione di architettura e API grafica, con una corrispondenza diretta con lo Shading Model nelle sue tre versioni, il concetto di profilo però è più specifico e va nel dettaglio delle differenze minime esistenti tra hardware, soprattutto se di produttori diversi.

Il codice dovrà oltre ad essere sintatticamente e concettualmente corretto, rispettare le restrizioni imposte dallo specifico profilo.

Tali limiti possono riguardare il numero e la dimensione delle texture utilizzabili o la presenza di cicli, il numero di iterazioni, e non ultima la dimensione del codice.

Si prevede che queste limitazioni in futuro saranno meno ingombranti ed i futuri profili comprenderanno intere generazioni di GPU, mantenendo riutilizzabile il codice scritto fino ad ora.

Per quanto fastidiosi, tali limiti sono un punto di forza del linguaggio, utilizzabile su GPU molto diverse tra loro.

Inoltre dare un limite alla dimensione e alle funzionalità di un codice porta ad una maggiore efficienza e velocità di esecuzione.

Va comunque sottolineato che tali restrizioni sono conseguenza diretta delle differenze architetturali delle GPU e non insite dei linguaggi.

Il linguaggio di shading sono abbastanza espressivi da poter definire algoritmi che ancora non potrebbero girare sull'hardware attuale.

## **4.8 La scelta del linguaggio**

Da quanto esposto si capisce che la scelta non è stata dettata dall'espressività dei linguaggi. A rafforzare questa tesi si pensi che il Cg della NVIDIA e il HLSL della Microsoft sono stati sviluppati come un unico linguaggio con una forte collaborazione, la differenza di nome sta solo nel fatto che le due società ad un certo punto hanno evidenziato politiche commerciali differenti e quindi hanno continuato a sviluppare indipendentemente quel linguaggio con diversi nomi. rimane comunque vero che la scelta del linguaggio di programmazione col quale scrivere un'applicazione è sempre una scelta delicata. Da essa dipendono moltissime conseguenze.

Prima tra tutte le prestazioni dell'applicazione stessa, dato che i compilatori non sono assolutamente perfetti e quindi da un codice concettualmente identico scritto in due linguaggi diversi, non è detto che si ottenga lo stesso risultato.

La portabilità del codice per i vari sistemi operativi e la compatibilità con le API ( Application Programming Interface ) grafiche è un altro fattore determinante al successo di un'applicazione e sui costi di sviluppo.

Nel caso dei linguaggi di shading non è poi da sottovalutare il supporto alle tre diverse versioni di Shader Model e quindi alla ampiezza del raggio di scelta dell'hardware sul quale si eseguirà il codice non che delle funzionalità disponibili e quelle mancanti. Di estrema importanza poi è la leggibilità del codice, la reperibilità di documentazione, il loro costo, la diffusione del linguaggio, l'ampiezza della comunità di programmatori, la predisposizione ad accogliere e formare nuovi programmatori, l'affidabilità degli sviluppatori e le loro politiche di mercato, le previsioni di crescita e di sopravvivenza del linguaggio stesso.

Per la nostra applicazione si è scelto il Cg della NVIDIA. Esponiamo di seguito una tabella molto dettagliata che evidenzia tutte queste differenze e molte altre in più tra HLSL, GLSL e Cg che motivano la scelta.

<b>Availability</b>	<b>HLSL</b>	<b>GLSL</b>	<b>Cg</b>
Installation/upgrade requirements	Part of DirectX 9; comes with XP or a free Windows updated	May need driver update from hardware vendor for ARB extensions or OpenGL 2.0	User level libraries so no driver upgrade typically required
Time of first release	March 2003, DirectX 9 ship	June 2003, ARB standards approved; implementations in late 2003	December 2002, 1.0 release
Current version	DirectX 9.0c	1.10	Cg 1.4
Standard maker	Microsoft	OpenGL Architectural Review Board	NVIDIA
Implementer	Microsoft	Each OpenGL driver vendor	NVIDIA

<b>Operating System Support</b>	<b>HLSL</b>	<b>GLSL</b>	<b>Cg</b>
Windows support (98/2000/XP)	Yes	Yes	Yes
Legacy Windows 95 support	No	Yes	Yes
Legacy Windows NT 4.0 support	No	Yes	Yes
Linux	No	Yes	Yes
Mac OS X	No	Yes	Yes

<b>3D Graphics API Support</b>	<b>HLSL</b>	<b>GLSL</b>	<b>Cg</b>
OpenGL	No	Yes	Yes
Direct3D	Yes	No	Yes
One shader can compile for either API	No	No	Yes

<b>Tools</b>	<b>HLSL</b>	<b>GLSL</b>	<b>Cg</b>
Virtual machine for CPU shader execution	Yes	No	Yes
Debugging	Visual Studio .NET Shader Debugger	No	No
Open source language parser	No	Yes	Yes

<b>OpenGL Specifics</b>	<b>HLSL</b>	<b>GLSL</b>	<b>Cg</b>
OpenGL 1.x support	n/a	Needs ARB GLSL extensions	Needs ARB or NV low-level assembly extensions
Multi-vendor ARB_vertex_program support	n/a		Yes
Multi-vendor ARB_fragment_program support	n/a	No	Yes
NVIDIA OpenGL extension support	n/a	No	Yes, profiles for fp20, vp20, fp30, vp30, fp40, and

<b>OpenGL Specifics</b>	<b>HLSL</b>	<b>GLSL</b>	<b>Cg</b>
			vp40
Relationship to OpenGL standard	n/a	Part of core OpenGL 2.0 standard	Layered upon ARB-approved assembly extensions
Access to OpenGL state settings	n/a	Yes	Yes
Open Source OpenGL rendering support (via Mesa)	n/a	No Mesa support yet	Yes, no changes required
Language tied to OpenGL	n/a	Yes	No, API-independent

<b>Direct3D Specifics</b>	<b>HLSL</b>	<b>GLSL</b>	<b>Cg</b>
DirectX 8 support	Requires DirectX 9 upgrade but supports DirectX 8-class hardware profiles	n/a	Yes
DirectX 9 support	Yes	n/a	Yes

<b>GPU Hardware Support</b>	<b>HLSL</b>	<b>GLSL</b>	<b>Cg</b>
NVIDIA DirectX 9-class GPUs	Yes	Yes	Yes
ATI DirectX 9-class GPUs	Yes	Yes	Yes
3Dlabs DirectX 9-class GPUs	Yes	Yes	Yes
DirectX 8-class GPUs	Yes, with ps1.x and vs1.1 profiles	No	Yes, fp20 and vp20 profiles

<b>Language Details</b>	<b>HLSL</b>	<b>GLSL</b>	<b>Cg</b>
Fixed-point data type	No	No, has fixed reserved word	Yes, fixed
Matrix arrangement	<b>Column major by default</b>	Column major	Row major by default
Non-square matrices	Yes	No	Yes
Shader outputs readable	Yes	No	Yes

<b>Language Details</b>	<b>HLSL</b>	<b>GLSL</b>	<b>Cg</b>
Vector write masking	Yes	No	Yes
Vector component and matrix swizzling	Yes	No	Yes
Vector ?: operator	Yes	No, Boolean only	Yes
Vector comparison operators	Yes	No, must use lessThan, etc. standard library routines	Yes
Semantic qualifiers	Yes	No	Yes
Array dimensionality	Multi-dimensional	1D only	Multi-dimensional
Un-sized arrays (dynamic sizing)	No	No	Yes

<b>Documentation</b>			
Specification or definitive documentation	MSDN Direct3D 9 documentation	The OpenGL Shading Language specification	The Cg Language Specification
Tutorial book	Various books	OpenGL Shading Language (Rost)	The Cg Tutorial (Fernando & Kilgard)
User's manual	MSDN Direct3D 9 documentation	No	Cg User's Manual

## 4.9 L'ambiente grafico

I linguaggi di shading come il Cg sono solo uno dei componenti dell'intera infrastruttura software ed hardware per renderizzare in tempo reale complesse scene tridimensionali con GPU programmabili.

Agli inizi della grafica tridimensionale su personal computer, prima dell'avvento delle GPU, le CPU gestivano tutte le trasformazioni e l'elaborazione dei pixel richieste per renderizzare una scena tridimensionale.

L'hardware grafico forniva esclusivamente il supporto del buffer per i pixel che la scheda visualizzava sul monitor. I programmatori dovevano implementare i loro algoritmi di rendering grafico tridimensionale via software che la CPU eseguiva serialmente.

Allo stato attuale come abbiamo visto esiste hardware dedicato, ma molto è stato fatto a livello software. Tra gli elementi software indispensabili per la grafica tridimensionale troviamo le OpenGL e Direct3D.

#### **4.9.1 OpenGL e Direct3D**

OpenGL e Direct3D sono equivalenti e consistono in una libreria specifica per la grafica che mette a disposizione del programmatore un piccolo set per dichiarare le primitive geometriche quali punti, linee, poligoni, immagini, e bitmap. Un set di comandi permette di aggregare le primitive per la definizione di oggetti 2D e 3D, ed infine una serie di comandi controllano come questi oggetti vengono renderizzati sul frame buffer.

La libreria fornisce strumenti di controllo diretto sui fondamentali operatori 2D e 3D tra cui matrici di trasformazione, coefficienti di illuminazione, metodi di antialiasing, ed operatori per l'aggiornamento dei pixel.

La differenza sta ancora una volta nella portabilità. OpenGL è sviluppata da Silicon Graphics in collaborazione con un'organizzazione chiamata OpenGL Architecture Review Board ( ARB ) che comprende tutte le maggiori aziende produttrici di sistemi grafici.

Originariamente OpenGL girava solo su potenti workstation grafiche UNIX. Microsoft come membro della ARB implementò la propria versione per i sistemi Windows.

OpenGL quindi non si limita ad un singolo sistema operativo o ad un singolo windowing system e supporta anche Apple di Macintosh e Linux.

Direct3D è la diretta evoluzione delle DirectX, sviluppate da Microsoft soprattutto per i videogiochi attualmente include anche HLSL omologo del Cg e gira solo su sistemi Microsoft.



### 4.9.2 Il compilatore

La GPU non può eseguire direttamente i vertex program ed i fragment program nella loro forma testuale. I programmi vanno compilati scegliendo se si utilizzerà l'interfaccia OpenGL o Direct3D.

Successivamente tramite comandi delle suddette librerie si inviano i programmi tradotti alla GPU. Infine i driver OpenGL o Direct3D eseguiranno la traduzione finale nella forma eseguibile che la GPU richiede.

I dettagli di questa traduzione dipende dalla combinazione del tipo di GPU e dell'interfaccia 3D scelta tra le suddette.

Può accadere che uno specifico programma non possa essere compilato perché contiene istruzioni non eseguibili dalla scheda presente nel sistema.

### 4.9.3 La compilazione dinamica

Quando si compila un programma scritto in un linguaggio convenzionale come il C o il C++, la compilazione è una procedura offline. Diciamo che la compilazione è statica. Il compilatore genera un programma eseguibile e lo esegue direttamente nella CPU. Una volta compilato il programma non ha bisogno di ricompilazioni a meno che non si modifichi il codice.

I linguaggi di shading sono diversi ed è preferibile la compilazione dinamica, anche se quella statica è ugualmente supportata. Il compilatore non è un programma separato, ma una parte di una libreria che nel caso del Cg si chiama Cg-runtime.

L'applicazione che usa la compilazione dinamica, chiama una routine che compila e manipola il vertex program o il fragment program. La compilazione dinamica permette di ottimizzare i kernel grafici per la specifica GPU installata sulla macchina in uso.

Tale libreria dispone di funzioni che ritornano il tipo di scheda a disposizione, parametri quali il numero massimo di texture indirizzabili, la loro dimensione massima, inoltre vi è un insieme di funzioni specifiche che leggono da file o da

stringa uno kernel, lo compilano inviando il risultato alla scheda e lo attivano, disattivano o distruggono.

```
*context = cgCreateContext();

*fragmentProfile = cgGLGetLatestProfile(CG_GL_FRAGMENT);

cgGLSetOptimalOptions(*fragmentProfile);

/*cgCreateProgramFromFile - generate a new program object from a file*/
*fragmentProgram = cgCreateProgramFromFile(*context, CG_SOURCE,
"FragmentProgram.cg", *fragmentProfile, "FragmentProgram", 0);

//static load (precompiled)
/**fragmentProgram = cgCreateProgramFromFile(*context, CG_OBJECT,
"FragmentProgram.obj", *fragmentProfile, "FragmentProgram", 0);

if(*fragmentProgram == 0)
{
    CGError Error = cgGetError();
    fprintf(stderr, "%s \n", cgGetErrorString(Error));
    exit(EXIT_FAILURE);
}

//move program to GPU (shader)
/*cgGLLoadProgram - prepares a program for binding*/
cgGLLoadProgram(*fragmentProgram);
```

#### 4.9.4 Il window system

Parlando di applicazioni grafiche non si può fare a meno di un supporto che si occupi di gestire le finestre e tutti gli eventi relativi ad esse. Nel caso di Microsoft Windows il sistema è unico e esiste un'opportuna libreria. Nel caso di sistemi con più gestori del window system, in particolare Linux, abbiamo il supporto di GLUT.

GLUT (OpenGL Utility Toolkit) è un toolkit indipendente, per la gestione del window system. Utile nella scrittura di applicazioni grafiche con interfaccia a finestre, è portabile su varie piattaforme e sistemi operativi.



## Capitolo 5

### Applicazione

Per valutare le prestazioni dell'architettura GPU nell'ambito del calcolo General Purpose ed in particolare analizzare la possibilità di includerlo come nuovo strumento nell'ambito del High Performance Computing, si è realizzato un applicativo per il calcolo del prodotto matrice per matrice e lo si confronterà con una tra le maggiori soluzioni alternative che la ricerca scientifica mondiale ha proposto.

#### 5.1 Le matrici

La nozione di matrice si incontra spesso nella matematica e in tutte le sue applicazioni: statistica, fisica, chimica, elettrotecnica, ingegneria delle strutture, economia, scienze gestionali, sociologia, biologia molecolare, scienze naturali. Le matrici hanno grande importanza nella pratica computazionale e sono prese in precisa considerazione da tutti i linguaggi di programmazione procedurali, da quelli di livello medio-basso a partire dal Fortran, a quelli di livello alto come Matlab, ai linguaggi per le elaborazioni simboliche come Mathematica e Maple.

Operazioni basilari sulle matrici quali somma e prodotto sono concettualmente semplici, descrivibili in termini di un numero finito di somme e prodotti, ma nel caso di matrici dalle dimensioni notevoli, anche i più moderni elaboratori sono sottoposti ad una mole di calcoli così impressionante, che i tempi di attesa per i risultati sono talmente lunghi da rendere spesso inutile la realizzazione di applicativi che ne necessitano.

L'High Performance Computing ha come obiettivo quello di sfruttare al massimo le risorse a disposizione e ridurre i tempi d'attesa nel calcolo.

Le strade percorse sono diverse: l'utilizzo del SSE (Streaming SIMD Extensions), la ricerca di nuovi algoritmi, l'ottimizzazione del codice, l'utilizzo di architetture alternative quali cluster di computer. Dovendo fare un confronto per valutare le prestazioni, si è scelto di lavorare su di una singola macchina che monti una singola GPU e di far girare sulla stessa sia il nostro applicativo GPGPU, sia quella che si può considerare una delle maggiori soluzioni, anche per processore singolo, attualmente in circolazione, le librerie ATLAS, come ha dimostrato L. Benini [BEN04]. Tale scelta permette di fare un confronto che esuli da questioni di programmazione parallela quali la comunicazione e la distribuzione dei dati e del calcolo tra più unità.

## 5.2 Il prodotto matrice per matrice

Date due matrici, la prima  $A$  con  $m$  righe ed  $n$  colonne, i cui elementi indicheremo  $a_{i,j}$  e la seconda con  $n$  righe e  $p$  colonne, i cui elementi indicheremo con  $b_{i,j}$ . Definiamo il prodotto della matrice  $A$  per la matrice  $B$  come la matrice  $C$  di  $m$  righe e  $k$  colonne, i cui elementi indicheremo con  $c_{ij}$  con valore pari a:

$$c_{ij} = \text{sum} ( a_{ik} \times b_{ki} ) \text{ con } k = 1 \dots n$$

### 5.2.1 Il prodotto matrice per matrice “ijk”

Per eseguire il prodotto tra matrici con un computer vi sono diversi algoritmi, la versione più intuitiva è detta “ijk” e si compone di tre cicli. I primi due più esterni indicano quale elemento della matrice  $C$  andremo a calcolare e l'ultimo itera la sommatoria indicata sopra.

```

void matrixmatrix(float* checkImage1, float* checkImage2, float* checkImage3,
                  int m, int n, int q, int argc, char** argv)
{
    int i=0, j=0, k=0;
    for (i=0; i<m; i++)
        for (j=0; j<q; j++)
            for (k=0; k<n; k++)

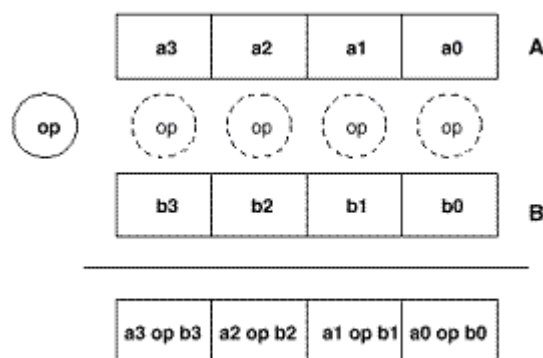
                checkImage3[i*q+j] += (checkImage1[i*n+k] * checkImage2[k*q+j]);
}

```

**Tabella 5 Prodotto matrice per matrice “ijk”**

## 5.2.2 Il prodotto matrice per matrice con SSE

Con il supporto SSE le operazioni di prodotto e somma possono essere eseguite per gruppi di quattro operandi per quattro operandi in un unico ciclo di clock secondo lo schema sottostante.



**Figura 16 Struttura di una istruzione SSE**

in questo caso volendo riscrivere il codice liberamente per dare una definizione intuitiva del risparmio di tempo nell’esecuzione otterremmo qualcosa di simile al codice sottostante.

```

for (i=0; i<m; i++) {
    for (j=0; j<p; j++) {
        sum = 0.0;
        for (k=0; k<n/4; k+=4)
            sum += a[i][k]*b[k][j]+
                   a[i][k+1]*b[k+1][j]+
                   a[i][k+2]*b[k+2][j]+
                   a[i][k+3]*b[k+3][j];
        c[i][j] = sum;
    }
}

```

Figura 17 Prodotto matriciale "ijg" SSE

### 5.2.3 Il prodotto matrice per matrice con ATLAS

ATLAS è un'implementazione della libreria BLAS (Basic Linear Algebra Subprograms) che consiste in un insieme di routines che eseguono le tradizionali operazioni su vettori e matrici organizzandole a blocchi.

Blas si suddivide in tre livelli:

- Livello 1 - operazioni scalari, vettoriali
- Livello 2 - operazioni tra vettori e matrici
- Livello 3 - operazioni tra matrici

Esistono diverse implementazioni, molto efficienti e specifiche per molti tipi di macchine compresi molti supercomputer.

L'obiettivo è di utilizzare in modo efficiente soprattutto su macchine parallele a memoria condivisa con processori vettoriali, le risorse rese disponibili dalla macchina che esegue il codice.

BLAS migliora le prestazioni riorganizzando gli algoritmi eseguendo operazioni a blocchi in modo che i tempi di latenza della memoria siano ridotti e l'utilizzo della cache sia ottimizzato. Queste operazioni possono essere ottimizzate in base all'architettura che eseguirà il calcolo e consente una

trasportabilità che mantiene alta l'efficienza su macchine molto diverse.

L'implementazione più completa di BLAS è ATLAS che sta per Automatically Tuned Linear Algebra Software. ATLAS è sia un progetto di ricerca che un pacchetto software. La versione corrente fornisce una copertura completa delle funzionalità di BLAS sia per C che per Fortran<sup>77</sup>.

Per rendersi conto dell'affidabilità e portabilità della soluzione si pensi che è usata in programmi come MAPLE, MATLAB, Mathematica, Octave è incluso anche in Absoft Pro Fortran e può girare su sistemi Debian Linux, FreeBSD, Mac OS 10, Scyld Beowulf, SuSE Linux e su Windows emulando Linux.

Per eseguire il prodotto matrice per matrice con l'utilizzo di ATLAS occorrerà quindi compilare preventivamente la libreria per la specifica macchina che si andrà ad utilizzare ed effettuare nel codice la chiamata alla procedura specifica della libreria, ovvero *cblas\_sgemv*, che calcola  $C = \alpha A B + \beta C$ .

```
void cblas_sgemv (
    const enum CBLAS_ORDER Order,          /* disposizione matrici in memoria */
    const enum CBLAS_TRANSPOSE TransA,      /* indica se A è trasposta */
    const enum CBLAS_TRANSPOSE TransB,      /* indica se B è trasposta */
    const int M,                            /* righe di A */
    const int N,                            /* colonne di B */
    const int K,                            /* colonne di A, righe di B */
    const float alpha,                      /* scalare */
    const float * A,                        /* puntatore ad A */
    const int lda,                          /* colonne di A */
    const float * B,                        /* puntatore a B */
    const int ldb,                          /* colonne di B */
    const float beta,                       /* scalare */
    float * C,                              /* puntatore a C */
    const int ldc                           /* colonne di C */
)
```



### 5.2.4 Il prodotto matrice per matrice su GPU

Per comprendere meglio come eseguire il prodotto su GPU introdurremo delle semplificazioni che al momento dell'implementazione verranno ottimizzate. Una matrice può essere rappresentata come una texture in scala di grigi, dove ogni pixel contiene un elemento della matrice. Le matrici possono essere visualizzate sullo schermo disegnando rettangoli  $m \times n$  di pixel con le texture applicate con coordinate (0,0) (0,n-1) (m-1,n-1) (m-1,0) ai vertici del rettangolo. Il primo vantaggio che si può notare è che per ottenere la trasposta della matrice basta applicare la texture al rettangolo invertendo le coordinate di adiacenza della texture. Il prodotto di matrice per matrice  $C=AB$  memorizzando le matrici A e B come texture può essere calcolato in n cicli di rendering.

Si può descrivere con un grosso livello di astrazione l'algoritmo come segue.

```
Pulisci lo schermo
Definisci un rettangolo tramite Quattro vertici
Carica la texture A
Carica la texture B
For (i=0; i<n; i++)
    Pixel=pixel+look_textureA_at_coord ( pixel_row , i )
    * look_textureB_at_coord( i , pixel_col )
```

Per ogni passo, la texture A replica la sua i-esima colonna su tutte le colonne e la texture B replica la sua i-esima riga su tutte le righe. Le due texture vengono applicate al rettangolo come prodotto, quindi il prodotto viene accumulato. Descriviamo le operazioni svolte dall'algoritmo graficamente.

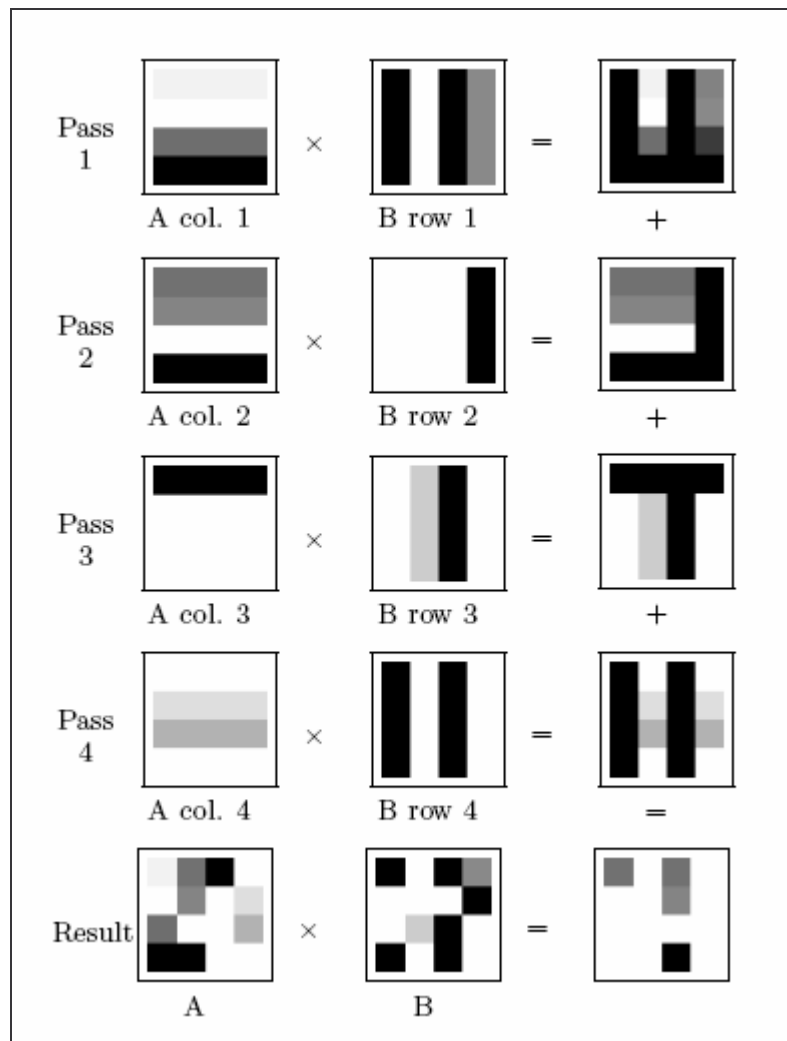


Figura 18 Prodotto matriciale su GPU visto graficamente

Si tenga conto che in realtà ogni pixel può gestire quattro canali di colore con precisione floating-point a 32 bit per i canali RGBA. Questo permette di ottimizzare ulteriormente l'algoritmo sopra descritto.

### 5.2.5 Il prodotto matrice per matrice a blocchi

Sia definita la matrice

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & \cdot & \cdot & a_{1,n-1} & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & \cdot & \cdot & a_{2,n-1} & a_{2,n} \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ a_{m-1,1} & a_{m-1,2} & a_{m-1,3} & a_{m-1,4} & \cdot & \cdot & a_{m-1,n-1} & a_{m-1,n} \\ a_{m,1} & a_{m,2} & a_{m,3} & a_{m,4} & \cdot & \cdot & a_{m,n-1} & a_{m,n} \end{pmatrix}$$

Possiamo definire matrice a blocchi in questo caso di dimensione due per due la matrice equivalente:

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdot & \cdot & A_{1,N} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ A_{M,1} & A_{M,2} & \cdot & \cdot & A_{M,N} \end{pmatrix}$$

con  $N=n/4$  ed  $M=m/4$  e di conseguenza

$$A_{1,1} = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix}$$

e più in generale

$$A_{ij} = \begin{pmatrix} a_{4i, 4j} & a_{4i, 4j+1} \\ a_{4i+1, 4j} & a_{4i+1, 4j+1} \end{pmatrix}$$

Le notazioni scelte suggeriscono che quando si ripartisce la matrice A in blocchi, si può pensare ad essa come ad una matrice i cui elementi sono i blocchi della ripartizione effettuata.

Siano

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ \cdots & \cdots & \cdots & \cdots \\ A_{m,1} & A_{m,2} & \cdots & A_{m,n} \end{pmatrix}$$

e

$$B = \begin{pmatrix} B_{1,1} & B_{1,2} & \cdots & B_{1,s} \\ \cdots & \cdots & \cdots & \cdots \\ B_{r,1} & B_{r,2} & \cdots & B_{r,s} \end{pmatrix}$$

Se  $r = n$  allora  $AB = C$  è la matrice a blocchi

$$C = \begin{pmatrix} C_{1,1} & C_{1,2} & \cdots & C_{1,s} \\ \cdots & \cdots & \cdots & \cdots \\ C_{m,1} & C_{m,2} & \cdots & C_{m,s} \end{pmatrix}$$

$$\text{con } C_{ij} = A_{i,1}B_{1,j} + A_{i,2}B_{2,j} + \dots + A_{i,n}B_{n,j} = \sum_{1 \leq k \leq n} A_{i,k} B_{k,j}$$

### 5.2.5 Il prodotto matrice per matrice su GPU in RGBA

Rappresentando i blocchi con i quattro canali colore RGBA tipici di un pixel possiamo eseguire il prodotto a blocchi su GPU.

Definiamo le operazioni per il singolo prodotto blocco per blocco.

Dati i blocchi

$$A_{1,1} = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} \quad \text{e} \quad B_{1,1} = \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{pmatrix}$$

memorizzati su due pixel nella seguente maniera:

R	G	B	A
$a_{1,1}$	$a_{1,2}$	$a_{2,1}$	$a_{2,2}$

r	g	b	a
$b_{1,1}$	$b_{1,2}$	$b_{2,1}$	$b_{2,2}$

Dato che

$$A_{1,1} B_{1,1} = \begin{pmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} & a_{1,1}b_{1,2} + a_{1,2}b_{2,2} \\ a_{2,1}b_{1,1} + a_{2,2}b_{2,1} & a_{2,1}b_{1,2} + a_{2,2}b_{2,2} \end{pmatrix}$$

che su pixel sarebbe:

Rr+Gb	Rg+Ga	Br+Ab	Bg+Aa
$a_{1,1}b_{1,1}+a_{1,2}b_{2,1}$	$a_{1,1}b_{1,2}+a_{1,2}b_{2,2}$	$a_{2,1}b_{1,1}+a_{2,2}b_{2,1}$	$a_{2,1}b_{1,2}+a_{2,2}b_{2,2}$

Se scomponessimo il risultato come somma di due pixel avremo:

Rr	Ga	Br	Aa
$a_{1,1}b_{1,1}$	$a_{1,1}b_{1,2}$	$a_{2,1}b_{1,1}$	$a_{2,2}b_{2,2}$
+	+	+	+
Gb	Rg	Ab	Bg
$a_{1,2}b_{2,1}$	$a_{1,2}b_{2,2}$	$a_{2,2}b_{2,1}$	$a_{2,1}b_{1,2}$
=	=	=	=
Rr+Gb	Rg+Ga	Br+Ab	Bg+Aa
$a_{1,1}b_{1,1}+a_{1,2}b_{2,1}$	$a_{1,1}b_{1,2}+a_{1,2}b_{2,2}$	$a_{2,1}b_{1,1}+a_{2,2}b_{2,1}$	$a_{2,1}b_{1,2}+a_{2,2}b_{2,2}$

Che a loro volta possono essere scomposti come prodotto tra due pixel, con l'accortezza di abbinare i canali al corrispondente prodotto.

R	G	B	A
$a_{1,1}b_{1,1}$	$a_{1,1}b_{1,2}$	$a_{2,1}b_{1,1}$	$a_{2,2}b_{2,2}$
x	x	x	x
r	a	r	a
$a_{1,1}b_{1,1}$	$a_{1,1}b_{1,2}$	$a_{2,1}b_{1,1}$	$a_{2,2}b_{2,2}$
=	=	=	=
Rr	Ga	Br	Aa
$a_{1,1}b_{1,1}$	$a_{1,1}b_{1,2}$	$a_{2,1}b_{1,1}$	$a_{2,2}b_{2,2}$

Da notare che i canali colore, nel secondo operando, non sono conformi alla rappresentazione del pixel. Questo non è un problema perché le GPU eseguono queste operazioni di permutazione o replica del canale senza perdita di prestazioni.

### 5.2.6 Il wrapper e l'unwrapper

Dovendo lavorare con delle matrici, ci si dovrà preoccupare anche della formattazione dei dati. Volendo mantenere una coerenza sintattica e formale con le soluzioni alternative di calcolo del prodotto, descritte nel capitolo precedente, si manterrà la stessa disposizione dei dati di input e di output. Le matrici quindi saranno memorizzate in modo contiguo, riga per riga.

Anche le texture sono memorizzate per righe di pixel e per ognuno in successione sono memorizzati i canali di colore RGBA.

La necessità di calcolare il prodotto a blocchi, dove ogni blocco  $2 \times 2$  è memorizzato su di un singolo pixel, contrasta con questo formalismo.

Ogni blocco copre due righe della matrice, mentre leggendo la matrice come texture, ogni pixel conterrebbe quattro elementi della stessa riga. Per fare in modo che il prodotto a blocchi sia possibile è necessario ridistribuire opportunamente i dati tramite un wrapper, altrettanto occorrerà fare per l'output.

Per capire meglio faremo un esempio. Sia la matrice  $A$  di dimensioni  $4 \times 4$ .

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

La matrice verrebbe memorizzata come vettore contiguo di righe.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------

Il vettore, così com'è, risulterebbe una texture di quattro pixel.

<table><tr><td><math>a_{0,0}</math></td><td><math>a_{0,1}</math></td><td><math>a_{0,2}</math></td><td><math>a_{0,3}</math></td></tr></table> <p>Pixel 0,0</p>	$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	<table><tr><td><math>a_{1,0}</math></td><td><math>a_{1,1}</math></td><td><math>a_{1,2}</math></td><td><math>a_{1,3}</math></td></tr></table> <p>Pixel 0,1</p>	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$						
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$						
<table><tr><td><math>a_{2,0}</math></td><td><math>a_{2,1}</math></td><td><math>a_{2,2}</math></td><td><math>a_{2,3}</math></td></tr></table> <p>Pixel 1,0</p>	$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	<table><tr><td><math>a_{3,0}</math></td><td><math>a_{3,1}</math></td><td><math>a_{3,2}</math></td><td><math>a_{3,3}</math></td></tr></table> <p>Pixel 1,1</p>	$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$						
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$						

Per memorizzare un blocco per ogni pixel la disposizione dovrebbe essere la seguente.

<table> <tr> <td><math>a_{0,0}</math></td><td><math>a_{0,1}</math></td></tr> <tr> <td><math>a_{1,0}</math></td><td><math>a_{1,1}</math></td></tr> </table> <p>Pixel 0,0</p>	$a_{0,0}$	$a_{0,1}$	$a_{1,0}$	$a_{1,1}$	<table> <tr> <td><math>a_{0,2}</math></td><td><math>a_{0,3}</math></td></tr> <tr> <td><math>a_{1,2}</math></td><td><math>a_{1,3}</math></td></tr> </table> <p>Pixel 0,1</p>	$a_{0,2}$	$a_{0,3}$	$a_{1,2}$	$a_{1,3}$
$a_{0,0}$	$a_{0,1}$								
$a_{1,0}$	$a_{1,1}$								
$a_{0,2}$	$a_{0,3}$								
$a_{1,2}$	$a_{1,3}$								
<table> <tr> <td><math>a_{2,0}</math></td><td><math>a_{2,1}</math></td></tr> <tr> <td><math>a_{3,0}</math></td><td><math>a_{3,1}</math></td></tr> </table> <p>Pixel 1,0</p>	$a_{2,0}$	$a_{2,1}$	$a_{3,0}$	$a_{3,1}$	<table> <tr> <td><math>a_{2,2}</math></td><td><math>a_{2,3}</math></td></tr> <tr> <td><math>a_{3,2}</math></td><td><math>a_{3,3}</math></td></tr> </table> <p>Pixel 1,1</p>	$a_{2,2}$	$a_{2,3}$	$a_{3,2}$	$a_{3,3}$
$a_{2,0}$	$a_{2,1}$								
$a_{3,0}$	$a_{3,1}$								
$a_{2,2}$	$a_{2,3}$								
$a_{3,2}$	$a_{3,3}$								

Tale configurazione sarebbe disposta in memoria in modo differente da quanto sopra.

<table> <tr> <td><math>a_{0,0}</math></td><td><math>a_{0,1}</math></td><td><math>a_{1,0}</math></td><td><math>a_{1,1}</math></td></tr> </table> <p>Pixel 0,0</p>	$a_{0,0}$	$a_{0,1}$	$a_{1,0}$	$a_{1,1}$	<table> <tr> <td><math>a_{0,2}</math></td><td><math>a_{0,3}</math></td><td><math>a_{1,2}</math></td><td><math>a_{1,3}</math></td></tr> </table> <p>Pixel 0,1</p>	$a_{0,2}$	$a_{0,3}$	$a_{1,2}$	$a_{1,3}$	<table> <tr> <td><math>a_{2,0}</math></td><td><math>a_{2,1}</math></td><td><math>a_{3,0}</math></td><td><math>a_{3,1}</math></td></tr> </table> <p>Pixel 1,0</p>	$a_{2,0}$	$a_{2,1}$	$a_{3,0}$	$a_{3,1}$	<table> <tr> <td><math>a_{2,2}</math></td><td><math>a_{2,3}</math></td><td><math>a_{3,2}</math></td><td><math>a_{3,3}</math></td></tr> </table> <p>Pixel 1,1</p>	$a_{2,2}$	$a_{2,3}$	$a_{3,2}$	$a_{3,3}$
$a_{0,0}$	$a_{0,1}$	$a_{1,0}$	$a_{1,1}$																
$a_{0,2}$	$a_{0,3}$	$a_{1,2}$	$a_{1,3}$																
$a_{2,0}$	$a_{2,1}$	$a_{3,0}$	$a_{3,1}$																
$a_{2,2}$	$a_{2,3}$	$a_{3,2}$	$a_{3,3}$																



Quindi bisogna trasformare la prima matrice perché appaia in memoria come descritto, prima del calcolo vero e proprio. L'operazione inversa andrà fatta sull'output.

## Capitolo 6

### Implementazione del prodotto matriciale su GPU

Per implementare il prodotto matrice per matrice come applicazione che giri su GPU occorre proiettare i paradigmi classici di programmazione alle risorse messe a disposizione dalla GPU.

Si introdurrà poi l'interfaccia di comunicazione tra il sistema e la GPU per il passaggio dei dati in input e per ritornare l'output. Infine analizzeremo il codice che girerà sulla GPU ed eseguirà il calcolo.

#### 6.1 Proiezione dei paradigmi di programmazione alla GPU

Da quanto fino ad ora descritto, occorre delineare delle analogie tra il modello di programmazione di una CPU e quello della GPU per poter scrivere dei kernel che possano eseguire algoritmi general purpose.

*GPU textures = CPU arrays:*

Le strutture dati fondamentali per gli array in una GPU sono le texture e gli array di vertici. Il fragment processor tende ad essere più utile nel calcolo general purpose su GPU rispetto al vertex processor, quindi, ogni volta che si vuole utilizzare un array, questo deve essere fatto nelle GPU sotto forma di texture.

*GPU shader programs = CPU inner loops body*

I processori in parallelo di una GPU sono il motore del calcolo, eseguito sul flusso di dati in input. Nelle CPU viene utilizzato un loop per iterare istruzioni contenute al suo interno su una serie di dati, magari memorizzati in un array. Il processo è eseguito sequenzialmente.

Nella GPU istruzioni simili vengono scritte all'interno dello shader program e automaticamente eseguite su tutti gli elementi dello stream.

L'ammontare del parallelismo dipende dal numero di processori nella GPU, ma molto anche da quanto si riesce a sfruttare l'aritmetica della GPU, ottimizzata per vettori a quattro dimensioni, come avviene per SSE.

*Render to texture = feedback*

Girando su una CPU un'applicazione può accedere alla memoria in qualsiasi momento ed eventualmente eseguire una lettura di feedback.

Attualmente è l'unico meccanismo che rende possibile il riutilizzo dell'output come input senza far transitare dati all'esterno della GPU è il render to texture che può essere immaginato come una scrittura su memoria write-only, eseguita come ultimo passo del rendering e riutilizzabile come input di un nuovo rendering.

*Geometry rasterization = computation invocation*

Descritte le analogie per la rappresentazione dei dati, il calcolo ed il feedback, non rimane altro che capire come eseguire il vertex shader program e il fragment shader program su di una GPU. Semplicemente occorre avviare la funzione di rendering del disegno.

Il vertex processor trasformerà i vertici, il rasterizer determinerà quanti pixel occorrono per coprire i triangoli generati dai vertici e il fragment shader program calcolerà il colore che il pixel dovrà avere.

La figura geometrica comunemente usata nella programmazione general-purpose su GPU è il rettangolo.

E' di fondamentale importanza settare la visuale generale della scena, il viewport, pari alla dimensione del rettangolo per avere un corrispondenza uno a uno tra indice dell'array e il valore del colore contenuto nella texture di input e del pixel di output.

## 6.2 Implementazione

L'applicazione è strutturata su due file principali:

- *gpumatrix.cpp*, che viene eseguito dalla CPU e funge da interfaccia;
- *FragmentProgram.cg*, che viene eseguito dal fragment processor;

Vi è poi il file *utility.h* in cui sono contenute funzioni di test, per la creazione delle matrici e la scrittura su file dei risultati.

Il *VertexProgram.cg* è stato scritto per testare i profili, ma nella fattispecie non esegue nessuna trasformazione, per questo non è caricato ed eseguito in fase di calcolo.

## 6.3 L'interfaccia

La realizzazione dell'interfaccia all'esecuzione dei programmi Cg riveste notevole importanza. E' in questa procedura che si eseguono le chiamate per indicare il formato e la quantità dei dati da elaborare, i parametri passati, si controlla la fruibilità delle risorse e le si attivano, si mandano in esecuzione gli shader program.

```

int matrixmatrixGPU(float* unwrapped_checkImage1 ,
                    float* unwrapped_checkImage2 ,
                    float* unwrapped_checkImage3 ,
                    int m,int n,int q,
                    int argc, char** argv)

{
int text_size,NWin;
CGcontext context = NULL;
CGprogram vertexProgram,fragmentProgram;
CGprofile vertexProfile,fragmentProfile;
GLuint fb;
CGparameter dim,kparz,matA,matB,matCk;
GLuint textId[4];
clock_t Start;
float diff;
int mid_m, mid_n, mid_q;

float* checkImage1=(float*)malloc(sizeof(float)*m*n);
float* checkImage2 =(float*)malloc(sizeof(float)*n*q);
float* checkImage3 =(float*)malloc(sizeof(float)*m*q);

wrapCPU(checkImage1,unwrapped_checkImage1,m,n);
wrapCPU(checkImage2,unwrapped_checkImage2,n,q);

NWin = glutCreateWindow(argv[0]);

glewInit();

(void)initCG(&context,&vertexProgram,&fragmentProgram,
            &vertexProfile,&fragmentProfile,
            &dim,&kparz,&matA,&matB,&matCk);

mid_m =m/2;
mid_n=n/2;
mid_q=q/2;

(void)initGL(&fb,textId,checkImage1,checkImage2,checkImage3,
            mid_m, mid_n, mid_q);

status=chkFBO();

(void)enableCG(&vertexProgram,&fragmentProgram,
            &vertexProfile,&fragmentProfile,
            &dim,&matA,&matB,textId,mid_n);

display(&fb,textId,checkImage3,
        &matCk,&kparz,
        mid_m, mid_n, mid_q);

glutDestroyWindow(NWin);

//free GPU memory
glDeleteFramebuffersEXT (1,&fb);
glDeleteTextures (1,&textId[0]);
glDeleteTextures (1,&textId[1]);
glDeleteTextures (1,&textId[2]);
glDeleteTextures (1,&textId[3]);
glDeleteTextures (1,&textId[4]);
cgDestroyContext(context);

unwrapCPU(unwrapped_checkImage3,checkImage3,m,q);
return 0;
}
// N.B. le procedure glBindTexture e glReadPixels sono dentro display

```

### **6.3.1 WrapCPU ed unwrapCPU**

Eseguono il wrapping e l'unwrapping delle matrici, come descritto nel capitolo precedente. Il suffisso CPU indica che tale operazione è eseguita dalla CPU, vi sarebbe infatti la possibilità di eseguire anche questa funzione in GPGPU.

### **6.3.2 GlutCreateWindow e glewinit**

Crea la finestra di lavoro. Essendo un'elaborazione grafica, necessita di una finestra che comunque resta invisibile. Questo implica che sulla macchina giri un window system ( Windows , Xserver ).

L'estensione GLEW delle OpenGL è necessaria per gestire il Framebuffer Object e per questo viene attivata.

### **6.3.3 InitCG**

I vertex program ed i fragment program sono ospitati all'interno di un contesto assieme ai relativi dati. Diciamo che il contesto è l'area di memoria riservata di questi programmi ed ai loro dati. Questa procedura crea il contesto che ospiterà i programmi Cg. Verifica quale scheda programmabile si andrà ad utilizzare settando i parametri ottimali per essa. Primi tra tutti il vertex profile ed il fragment profile, discriminanti del tipo di precisione utilizzabile e della possibilità di utilizzo di strutture di controllo quali cicli. Compila in run-time e crea il programmi shader dai file di testo .cg o dai precompilati .obj e li carica sulla GPU. Memorizza i puntatori di eventuali parametri aggiuntivi da passare agli shader oltre i loro input standard: vertici, colori, texture i cui puntatori all'interno della GPU sono indicati da delle costanti chiamate semantiche.

### 6.3.4 InitGL

Inizializza e alloca la memoria per la texture che memorizzerà l'output del render to texture e le texture di input che vengono caricate subito dopo l'allocazione. Ne specifica il formato numerico e la dimensione ed infine specifica il dirottamento del rendering sulla texture di output.

```
//-----  
// texture 1 : B  
//-----  
  
glActiveTexture(GL_TEXTURE1);  
glBindTexture(GL_TEXTURE_2D, textId[1]);  
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F_ARB, q, n, 0,  
             GL_RGBA, GL_FLOAT, checkImage2);  
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);  
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);  
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);  
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
```

### 6.3.5 EnableCG e ChkFBO

EnableCG attiva degli specifici shader tra i tanti possibili che possono essere stati creati, questi saranno attivi su ogni funzione di disegno fino alla loro disattivazione o distruzione.

ChkFBO controlla che la funzione di render to texture tramite un oggetto chiamato FBO framebuffer object sia possibile.

### 6.3.6 Display

Attiva le texture di input ed output, setta la viewport pari alla dimensione delle texture di input per non avere deformazioni delle texture stesse lasciando inalterata la corrispondenza uno ad uno dei pixel con le relative coordinate.

Definisce i vertici del quadrato da disegnare che rappresenta lo spazio degli indirizzi su cui gli shader lavoreranno. Incolla le texture al quadrato rispettando le proporzioni sempre per mantenere la corrispondenza degli indirizzi. Itera il rendering per il numero di volte necessario ed esegue il “ping pong”, vale a dire legge l'output del singolo passo e lo passa come input al rendering successivo come feedback. Scarica dalla GPU alla memoria centrale la texture contenente l'output.

```
void display(GLuint* fb,GLuint* textId, float* checkImage3,
             CGparameter* matCk, CGparameter* kparz,
             int m,int n, int q)
{
    int view[4];
    clock_t Start;
    float diff;

    glClear (GL_COLOR_BUFFER_BIT);

    //glActiveTexture - select server-side active texture unit
    //glBindTexture - bind a named texture to a texturing target

    //Matrix A
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, textId[0]);

    //Matrix B
    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_2D, textId[1]);

    glViewport(0,0,q, m);

    glBindFramebufferEXT(GL_FRAMEBUFFER_EXT,*fb);

    //Ping pong Framebuffer - Render to texture
    GLenum attachmentpoint[]={GL_COLOR_ATTACHMENT0_EXT, GL_COLOR_ATTACHMENT1_EXT};
    int write_to = 0;
    int read_from = 1;
    int swap;

    glDrawBuffer(attachmentpoint[write_to]);
    cgGLSetTextureParameter(*matCk,textId[read_from+2]);
    cgGLEnableTextureParameter(*matCk);

    for (int i=0;i<n;i++)
    {
        glDrawBuffer(attachmentpoint[write_to]);
        cgGLSetTextureParameter(*matCk,textId[read_from+2]);
        cgGLEnableTextureParameter(*matCk);

        cgGLSetParameter1f(*kparz,i);

        // put 2 textures on a quad (square): define 4 coordinates
        glBegin (GL_QUADS);
            //il range di definizione delle texture è [0,1]
            //il range di definizione del quadrato da disegnare è [-1,+1]
            glTexCoord2f(0.0,0.0);
            glVertex3f(-1. 0,-1. 0,0.0);

            glTexCoord2f(1.0,0.0);
            glVertex3f(1.0,-1. 0,0.0);
```



```

        glTexCoord2f(1.0,1.0);
        glVertex3f(1.0,1.0,0.0);

        glTexCoord2f(0.0,1.0);
        glVertex3f(-1.0,1.0,0.0);
    gland();

    swap=write_to;
    write_to=read_from;
    read_from=swap;

}
// get result from GPU frame buffer to CPU memory
glReadBuffer(attachmentpoint[read_from]);
glReadPixels(0,0,q,m,GL_RGBA,GL_FLOAT,checkImage3);

glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

}

```

## 6.4 Il prodotto matriciale come shader

L'inner loop del prodotto matrice per matrice a blocchi viene eseguito dal fragment program del file `FragmentProgram.cg`. Ricordiamo che l'iterazione per riga e per colonna è automaticamente eseguita dal rendering dei pixel che sono indipendenti e elaborati parallelamente.

Nel calcolo del prodotto come in molte altre applicazioni GPGPU il Vertex Program non riveste notevole importanza, sia per la difficoltà nella rappresentazione ed indirizzamento dei dati, sia per il ridotto numero di unità di calcolo rispetto a quelle disponibili per il Fragment Program.

Come anticipato le due matrici in input A e B vengono divise a blocchi due per due e passate alla GPU sotto forma di singolo pixel RGBA all'interno delle texture di dimensione (  $n \times n$  ) dove i valori del blocco  $A_{i,j}$  e  $B_{i,j}$  sono passati nella forma:

$R = a_{i,j}$	$G = a_{i,j+1}$
$B = a_{i+1,j}$	$A = a_{i+1,j+1}$

Il calcolo avverrà come rendering su un quadrato delle stesse dimensioni, specificate nel viewport, dato che nella definizione dei vertici si utilizzano coordinate normalizzate.

```
glViewport(0,0,q, m);
glBegin (GL_QUADS);
//il range di definizione delle texture è [0,1]
//il range di definizione del quadrato da disegnare è [-1,+1]
    glTexCoord2f(0.0,0.0);
    glVertex3f(-1.0,-1.0,0.0);

    glTexCoord2f(1.0,0.0);
    glVertex3f(1.0,-1.0,0.0);

    glTexCoord2f(1.0,1.0);
    glVertex3f(1.0,1.0,0.0);

    glTexCoord2f(0.0,1.0);
    glVertex3f(-1.0,1.0,0.0);
glEnd();
```

Ogni pixel è indipendente dagli altri, quindi ogni operazione è riferita al singolo e specifico pixel. Il fragment program è eseguito per ogni pixel, ovvero ogni blocco due per due della matrice C.

Ad ogni passo di rendering viene calcolata la matrice parziale  $C_k$  per k che va da 1 a n è  $C_k(i,j) = a(i,k) b(k,j)$  ed accumulata fino ad ottenere la matrice  $C=AB$ .

Nonostante ogni pixel di output dal rendering è elaborato indipendentemente dagli altri e le sue coordinate sono costanti, possiamo tramite Cg applicarvi il colore della texture desiderata alle coordinate che vogliamo, oppure una qualsiasi combinazione matematica di più texture a diverse coordinate.

Quindi se nel fragment program poniamo come valore di output

$$\text{pixel}(i,j)=\text{texture0}(i,k)*\text{texture1}(k,j)$$

otteniamo proprio  $C_k(i,j)$ .

Questo viene automaticamente calcolato per tutti i pixel di output per ogni  $1 \leq i \leq n$ ,  $1 \leq j \leq n$ , in parallelo per quante sono le unità del vertex processor.

Le matrici parziali  $C_k$  vengono accumulate per  $k$  che va da 0 ad  $n$  e la sommatoria parziale viene passata di nuovo come input alla pipeline al successivo passo di rendering, sfruttando la possibilità di feedback attraverso l'operazione di render to texture resa possibile dal framebuffer object.

```
struct F_Out {
    float4 color : COLOR ;
};

F_Out FragmentProgram (uniform float dimcgparam,
    uniform float cgkparz,
    float2 texCoord : TEXCOORD,
    uniform sampler2D cgA ,
    uniform sampler2D cgB,
    uniform sampler2D cgCk): COLOR
{
    F_Out OUT;

    float2 texCoordA;
    float2 texCoordB;
    float4 blockA;
    float4 blockB;

    texCoordA.y=texCoord.y;
    texCoordB.x=texCoord.x;
    texCoordA.x=(cgkparz+0.5)/dimcgparam;
    texCoordB.y=texCoordA.x;
    blockA=tex2D(cgA, texCoordA);
    blockB=tex2D(cgB, texCoordB);
    OUT.color = tex2D (cgCk, texCoord)+( blockA.xyzw * blockB.xwxw ) +(
    blockA.yxwz * blockB.zyzy );
}
```

### **FragmentProgram.cg**

## **Capitolo 7**

### **Analisi delle prestazioni**

Per valutare le prestazioni dell'applicazione, abbiamo effettuato un confronto con le soluzioni alternative che comunemente sono utilizzate. Non si può prescindere comunque dalla macchina che eseguirà il codice, nonché i compilatori e le librerie usate ovvero la piattaforma di lavoro.

#### **7.1 Piattaforma di lavoro**

La macchina utilizzata è un Pentium 4 a 3.00GHz con 1024 Kb di cache, scheda madre con PCI-Express 16X.

La GPU utilizzata è una NVIDIA 7800 GT con 256Mb di RAM.

Il sistema operativo della macchina è la distribuzione Linux Gentoo 3.4.4 con kernel ricompilato.

Le librerie Atlas sono della versione 3.6.0. Il codice dell'interfaccia è stato sviluppato in C++ e compilato con gcc 3.4.4.

Il Cg del fragment program è compilato con profilo -fp40 ( Shader Model 3.0) con il compilatore cgc versione 1.3.0001, sviluppato da NVIDIA.

## 7.2 Test comparativi

Per i test si sono misurate le prestazioni in termini di tempo, nell'esecuzione di due tipi di prodotto matrice matrice:

- Prodotto tra matrici quadrate di dimensioni variabili da 64 a 2048.
- Prodotto tra una matrice rettangolare di 512 colonne ed numero variabile di righe da 63 a 8192 per una matrice quadrata 512 x 512 .

Eseguita da:

- CPU senza nessun supporto
- CPU con compilazione ottimizzata O3
- CPU con supporto SSE
- CPU utilizzando le librerie ATLAS
- GPU

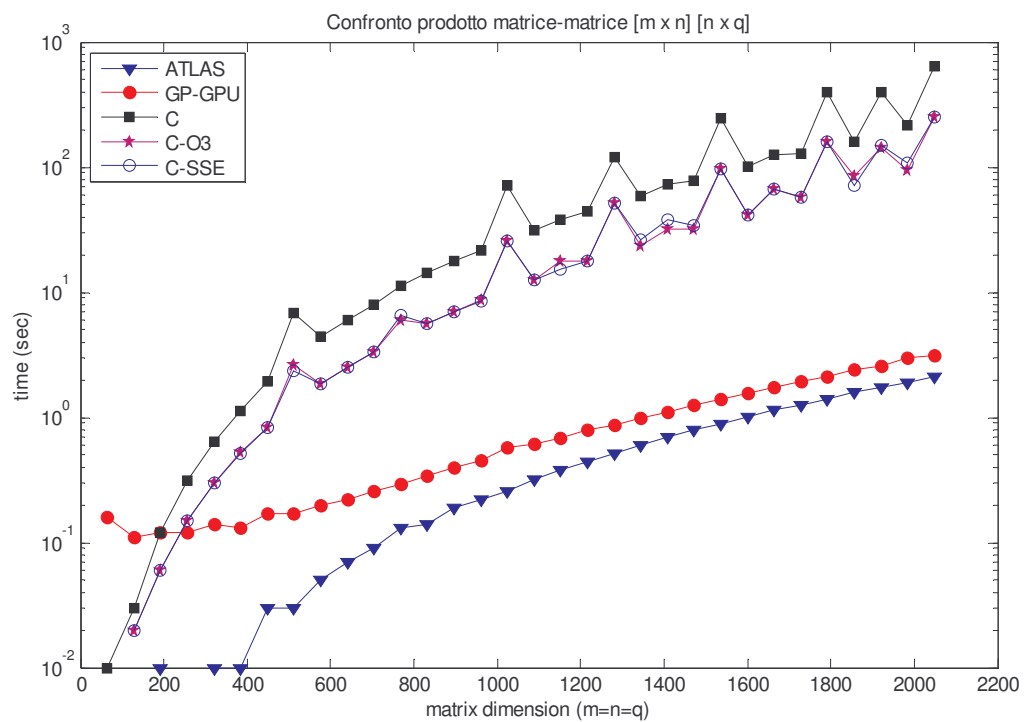
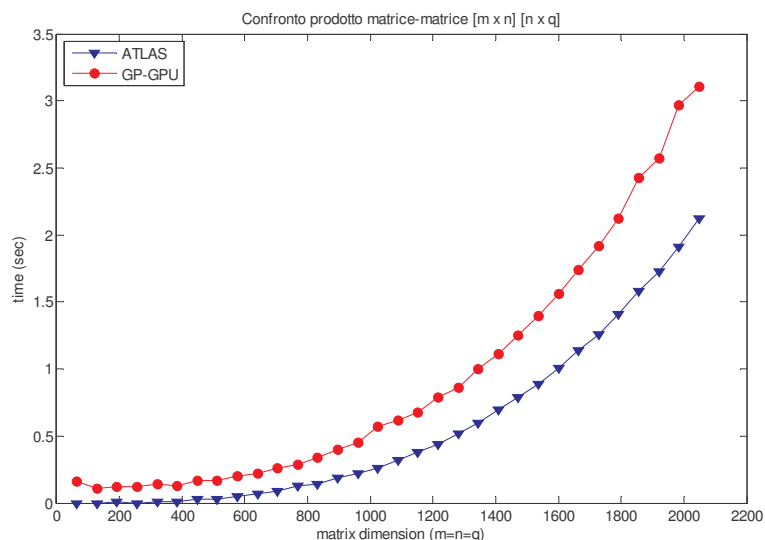


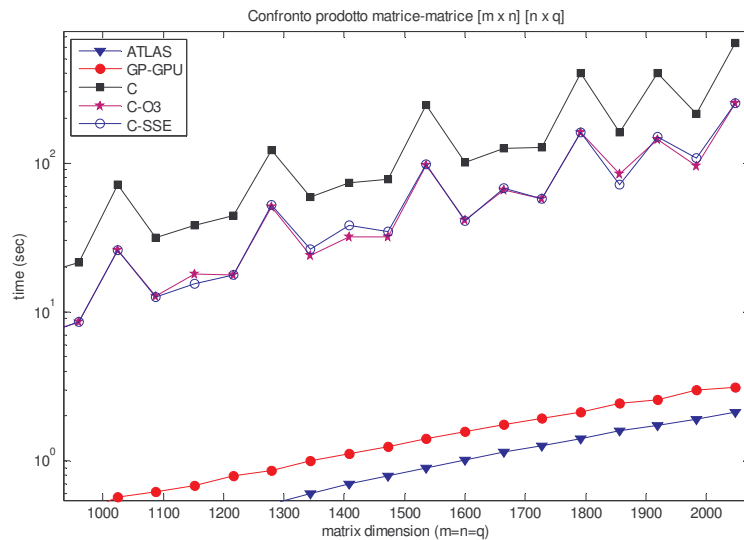
Tabella 6

La tabella 6 dimostra con una vista d'insieme, su scala logaritmica, come i risultati sono pienamente soddisfacenti e quanto ci siamo avvicinati alla soluzione fino ad oggi ottima, nella classe single processor.



**Tabella 7**

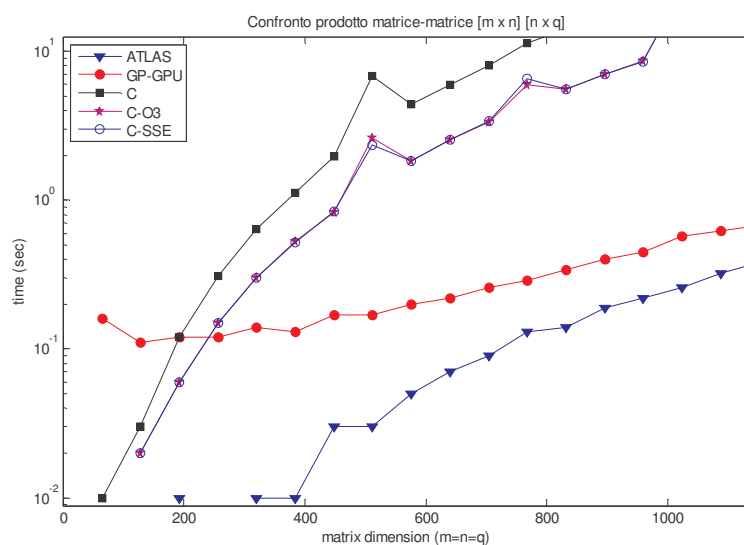
Analizzando in dettaglio il confronto tra le due soluzioni, ora su scala lineare, si può notare come si è riusciti a seguire l'andamento di quella che è la soluzione principe nel calcolo scientifico, con uno scarto pressoché costante, inferiore al secondo, nel peggiore dei casi. Si tenga conto che la libreria ATLAS è stata sviluppata da un team che annovera numerosi programmatori di altissimo livello. Una libreria che massimizza in modo esasperato l'utilizzo delle risorse. Si pensi che la libreria deve essere preventivamente compilata per la specifica macchina. Non è da sottovalutare che, all'interno del set di funzioni, strettamente definite, fornite da ATLAS, il prodotto matrice per matrice è quello che riesce meglio a sfruttare la cache e le altre ottimizzazioni. L'applicazione GPGPU è stata scritta liberamente e propone un modello di programmazione. Con la programmazione GPGPU si può uscire dal set di funzioni di ATLAS e scrivere una propria funzione che soddisfi specifiche esigenze. A tutto ciò si può aggiungere che nel prodotto suddetto il confronto è pienamente soddisfacente.



**Tabella 8**

Le valutazioni fatte sul livello di ottimizzazione nell'utilizzo delle risorse è evidenziato dalla tabella 8 che stringe il dettaglio sul fenomeno del cache trashing che avviene quando gli stessi blocchi vengono ripetutamente trasferiti da e verso la cache.

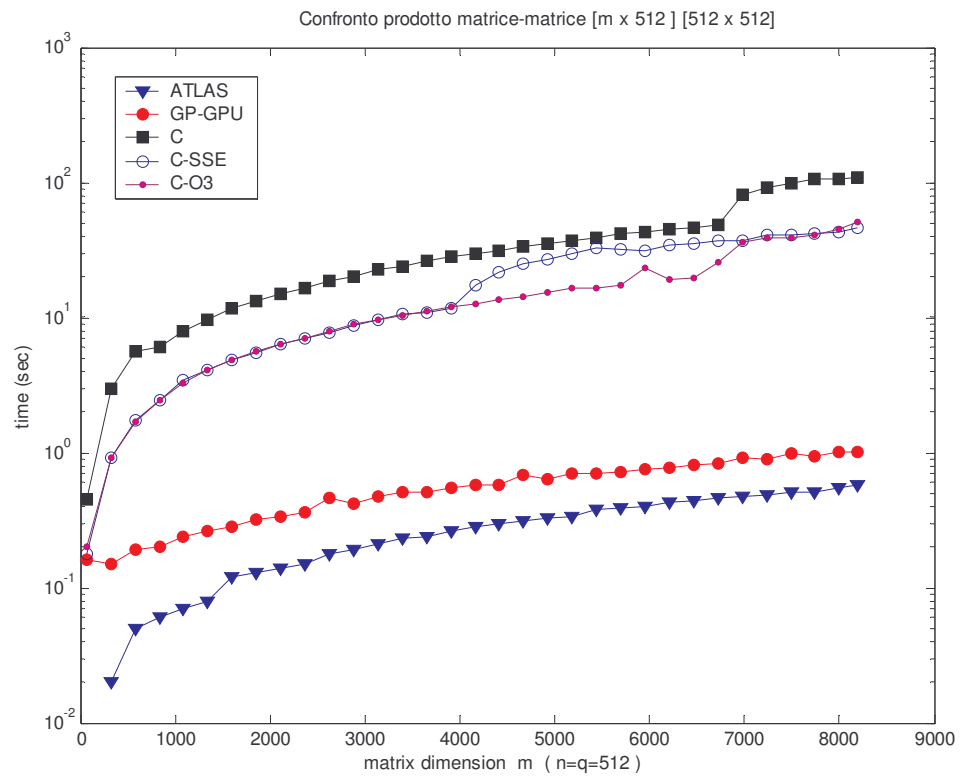
Si può notare che rispetto alle altre soluzioni, ATLAS e l'applicazione GPGPU hanno un effetto di cache trashing talmente ridotto che non si riesce ad evidenziare, perché il range di oscillazione è troppo piccolo in proporzione.



**Tabella 9**

La tabella 9 va a porre l'attenzione su una caratteristica specifica e insita nella programmazione GPGPU, il tempo di start-up.

Le applicazioni GPGPU sono eseguite dall'unità grafica, questo impone di inizializzare e trasferire dei parametri, caricare ed eventualmente compilare gli shader program. Queste operazioni, in parte, vengono eseguite in un tempo costante che è indipendente dalla dimensione del calcolo.



**Tabella 10**

La tabella 10 mostra i risultati del secondo test, quello con matrici rettangolari. Si può vedere come i risultati siano in linea con i precedenti. Da notare che è scomparso il grossolano effetto di cache trashing, dato che la matrice B in questo caso ha dimensioni ridotte (  $512 \times 512$  ).



### 7.3 Confronto numerico diretto

Analizzate graficamente le prestazioni e compresi i risultati dei test effettuati diamo un confronto numerico tra le varie soluzioni.

Modello di programmazione	Tempo di esecuzione (sec) Prodotto matrice – matrice ( $m = n = q = 2048$ )	$\Delta$ time rispetto alla versione GPGPU	$\Delta$ prestazioni rispetto alla versione GPGPU
C	651,36	648,25	20844,05 %
C – O3	282,36	279,25	8979,10 %
C – SSE	290,52	287,41	9241,48 %
C – SSE2	253,42	250,31	8048,55 %
C – SSE3	286,82	250,31	9122,51 %
ATLAS	2,14	-0,97	-31,19 %
GPGPU	3,11	0,00	0,00 %

**Tabella 11**

Quanto visto in precedenza, possiamo esprimerlo numericamente, confrontando le applicazioni su una dimensione del problema notevole ma non al limite. Si intuisce subito che le soluzioni hanno prestazioni con differenze spropositate, da una parte troviamo ATLAS e GPGPU e dall'altra le restanti

## 7.4 Analisi interna dell'applicazione GPGPU

Abbiamo visto nel paragrafo 6.3 che l'applicazione è costituita da un interfaccia che si occupa di eseguire quanto necessario perchè il calcolo sia eseguito dall'unità grafica.

<b>Procedura in esecuzione</b>	<b>Tempo di esecuzione m=n=q=1024</b>	<b>Tempo di esecuzione m=n=q=1024</b>	<b>Tempo di esecuzione m=n=q=4096</b>	<b>Descrizione procedura</b>
wrap ck1	0,02	0,04	0,18	Wrapping di A
wrap ck2	0,00	0,04	0,18	Wrapping di B
glutCreateWindow	0,02	0,03	0,02	Crea la finestra di lavoro
glewInit	0,01	0,01	0,01	Inizializza l'estensione GLEX per il Framebuffer Object
initCG	0,10	0,09	0,10	Inizializza l'ambiente Cg,
InitGL	0,04	0,13	0,50	Inizializza le texture e le carica su GPU
EnableCG	0,00	0,00	0,00	Attivazione degli shader program
glBindTexture	0,01	0,04	0,24	Attiva le texture

Display	0,02	0,03	13,34	Esegue gli shader program
glReadPixels	0,36	2,78	12,19	Scarica la matrice C
unwrapCPU	0,00	0,03	0,13	Unwrap della matrice C
TOTALE	0,59	3,23	26,89	Tempo di esecuzione totale C = A B

**Tabella 12**

Questa tabella evidenzia un punto fondamentale: l'incidenza del download dei dati dalla GPU alla CPU sul tempo di esecuzione totale. Tema trattato in tutte le pubblicazioni scientifiche che si occupano di GPGPU. Rappresenta il collo di bottiglia di tutta l'architettura. Si tenga conto che diversamente del caso studiato, il download dell'intero Framebuffer non è sempre necessario, come nel caso in cui i risultati dell'elaborazione hanno come destinazione finale di output proprio la visualizzazione a video, come nel lavoro di E.Zangheri [ZAN04] oppure quando il dato finale è dimensionalmente molto ridotto rispetto all'input, si pensi ad un prodotto tra due vettori monodimensionali. Con l'avvento del BUS PCI-Express 16 le cose sono notevolmente migliorate ed il trend di sviluppo è promettente.

## 7.5 Conclusioni

In conclusione si può affermare con totale tranquillità che l'avvento delle unità grafiche programmabili ( GPU ) ha aperto una nuova e promettente strada nel campo del High Performance Computing.

Il trend di crescita tecnologico dei dispositivi va oltre la legge di Moore e tale crescita è fortemente incentivata dal mercato dei videogames, target principale per le GPU. I programmi sono multiplatforma e portabili. La tecnologia è scalabile. Gli strumenti di lavoro e la documentazione sono molto curati.

La programmazione GPGPU è molto recente, i primi tentativi risalgono al 2002 ma tutto fa pensare che questa tecnica di programmazione avrà un roseo futuro.

In questo studio si è voluto analizzare le potenzialità dello strumento GPU e lo si è analizzato isolandolo e valutandone le prestazioni. Ciò non toglie che tra gli utilizzi che se ne possono fare, sicuramente è da considerare quella di utilizzarlo come coprocessore matematico, utilizzato come supporto e non in sostituzione della CPU. Interessante sarà capire che vantaggi potrà portare l'utilizzo del GPGPU in campi di studio come le Support Vector Machine dove fino ad ora sono state proposte soluzioni basate su cluster, come quella di F.Turroni [TUR05], oppure nel Advanced Machine Learning, tema trattato da M.Roffilli [ROF06]



## BIBLIOGRAFIA

- [1] VV.AA. "GPUGems 2"
- [2] Randima Fernando, Mark J. Kilgard "The Cg Tutorial"
- [3] OpenGL Architecture Review Board, Dave Shreiner "OpenGL Reference Manual"
- [4] OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis "OpenGL Programming Guide"
- [5] Jesse D. Hall, Nathan A. Carr, John C. Hart "Cache and bandwidth aware matrix multiplication on GPU"
- [6] S. E. Larson, D. McAllister "Fast matrix multiplies using graphics hardware. Super Computing"
- [7] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware."
- [8] Ian Buck Tim Foley Daniel Horn Jeremy Sugerman Kayvon Fatahalian Mike Houston Pat Hanrahan "Brook for GPUs: Stream Computing on Graphics Hardware"
- [9] Fernando, Harris, Wloka and Zeller "Programming Graphics Hardware (Eurographics 2004 Tutorial)"
- [10] Ian Buck "GPU Computation Strategies and Tricks" SIGGRAPH 2004 GPGPU Tutorial Presentations

- [11] Aaron Lefohn "GPU Data Formatting and Addressing" SIGGRAPH 2004 GPGPU Tutorial Presentations
- [12] Cliff Woolley "Efficient Data Parallel Computing on GPUs" SIGGRAPH 2004 GPGPU Tutorial Presentations
- [13] M. Roffilli. "Advanced Machine Learning Techniques for Digital Mammography." PhD thesis, University of Bologna, Department of Computer Science, 2006.
- [14] O.Schiaratura "Progettazione ed implementazione di un sistema di calcolo ibrido multithread-multiprocesso per HPC: applicazione alla mammografia", Scienze dell'Informazione, 2003-2004
- [15] C.Zoffoli "Progettazione, realizzazione ed ottimizzazione di un cluster ibrido 32/64 bit per HPC altamente affidabile", Scienze dell'Informazione, 2004-2005
- [16] L.Benini: "Ottimizzazioni microarchitetturali per high performance computing", Scienze dell'Informazione, 2003-2004
- [17] E.Zangheri: "Progettazione e realizzazione di una gui multi-piattaforma per applicazioni mediche in 2D", Scienze dell'Informazione, 2003-2004
- [18] F.Turroni: "Studio delle prestazioni del training parallelo di support vector machine su cluster ibrido 32/64 bit", Scienze dell'Informazione, 2004-2005
- [19] NVIDIA site  
<http://www.nvidia.com>
- [20] NeHe Productions: OpenGL Lessons  
<http://nehe.gamedev.net>

[21] GPGPU.org SITE

<http://www.gpgpu.org>

[22] Dominik Goddeke. “GPGPU::Basic Math Tutorial.”

<http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html>